Aplikace spravující stavební projekty demonstrující doporučené postupy při tvorbě klient-server aplikací

Filip Řehák

Bakalářská práce
2018

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2017/2018

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Filip Řehák**
Osobní číslo:	**A15065**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Forma studia:	**prezenční**
Téma práce:	**Aplikace spravující stavební projekty demonstrující doporučené postupy při tvorbě klient-server aplikací**
Téma anglicky:	**A Client-server Application for Demonstrating the Best Practices Approach When Creating Client-Server Applications for Construction Projects**

Zásady pro vypracování:

1. **Popište vybrané technologie a doporučené postupy pro tvorbu multiplatformních klient-server a webových aplikací.**
2. **Analyzujte daný problém a navrhněte vhodnou architekturu pro aplikaci spravující stavební projekty s důrazem na multiplatformní řešení.**
3. **Navrhněte implementaci aplikace s využitím vhodných technologií.**
4. **Vytvořte ukázkovou aplikaci demonstrující klíčové prvky řešení.**
5. **Demonstrujte výsledky a formulujte závěr.**

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **NAGEL, Christian. Professional C# 6 and .Net Core 1.0. Indianapolis, IN: John Wiley, 2016. ISBN 9781119096603.**
2. **J. PRICE, Mark. C# 6 and .NET Core 1.0: Modern Cross-Platform Development. Birmingham: Packt Publishing, 2016. ISBN 9781785285691.**
3. **SINGLETON, James. ASP.NET Core 1.0 High performance. Birmingham: Packt Publishing, 2016. ISBN 9781785881893.**
4. **PETZOLD, CHARLES. Creating Mobile Apps with Xamarin.Forms [online]. Redmond, Washington: Microsoft Press, 2016 [cit. 2017-01-10]. ISBN ISBN: 978-1-5093-0297-0. Dostupné z: https://blogs.msdn.microsoft.com/microsoft_press/2016/03/31/free-ebook-creating-mobile-apps-with-xamarin-forms/.**
5. **Xamarin.Forms: Cross-Platform User Interfaces with Xamarin.Forms [online]. Xamarin, 2017 [cit. 2017-01-10]. Dostupné z: https://developer.xamarin.com/guides/xamarin-forms/.**

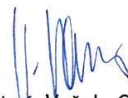
Vedoucí bakalářské práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **15. prosince 2017**

Termín odevzdání bakalářské práce: **25. května 2018**

Ve Zlíně dne 15. prosince 2017

doc. Mgr. Milan Adámek, Ph.D.
děkan



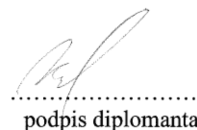
prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.



Ve Zlíně, dne 23.5.2018

....
podpis diplomanta

ABSTRAKT

Cílem práce je vytvoření ukázkové aplikace pro správu stavebních projektů, demonstrující nejnovější doporučené postupy při tvorbě klient-server aplikací. Výstupem práce bude návrh základních částí aplikace, které umožní správu jednotlivých projektů. Například jejich časové náročnosti, náklady a stavy. V teoretické části budou popsány vybrané technologie, doporučené postupy, návrhové vzory a techniky pro tvorbu klient-server aplikací. V praktické části potom budou doporučené postupy demonstrovány na vybraných částech ukázkové aplikace.

Klíčová slova: klient-server, návrhové vzory, multiplatformní vývoj, .NET Core, IoC, web API, SOLID, REST, Xamarin.Forms, OAuth, Dependency Injection

ABSTRACT

The aim of the bachelor thesis is creation of a sample application for management of the building projects, which demonstrates latest recommended practices for creating client-server application. The output of the thesis will be draft of the fundamental parts of the application, which will enable the management of the particular projects. For example, their estimated time, costs and progress state. Selected technologies, recommended practices, design patterns and techniques for creating client-server model applications will be described in the theoretical part. The practical part deals with recommended practices demonstrated on selected parts of the sample application.

Keywords: Client-Server, Design Patterns, multiplatform development, .NET Core, IoC, Web API, SOLID, REST, Xamarin.Forms, OAuth, Dependency Injection

Tímto děkuji mému vedoucímu práce, panu Ing. et Ing. Eriku Králi, Ph.D. za poskytnuté rady, věcné připomínky a věnovaný čas při konzultacích této práce. Dále také mé rodině a přítelkyni za motivaci a důvěru, kterou ve mne vkládají.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická, nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD ... **8**

I TEORETICKÁ ČÁST ... **9**

1 TECHNOLOGIE A DOPORUČENÉ POSTUPY PRO TVORBU MULTIPLATFORMNÍCH KLIENT-SERVER A WEBOVÝCH APLIKACÍ ... **10**

1.1 KLIENT-SERVER ... 10

1.2 FRAMEWORK ... 10

1.2.1 .NET Framework ... 11

1.2.2 ASP.NET ... 11

1.2.3 REST a web API ... 12

1.2.4 Entity Framework a Objektově Relační Mapování (ORM) ... 13

1.3 DOPORUČENÉ POSTUPY ... 14

1.3.1 Návrhové principy ... 14

1.3.1.1 SOLID ... 14

1.3.1.2 Law of Demeter ... 15

1.3.1.3 Don't Repeat Yourself ... 16

1.3.2 Architektura ... 16

1.3.2.1 MVC ... 16

1.3.2.2 MVP ... 17

1.3.2.3 MVVM ... 19

1.3.3 Testování ... 19

1.3.3.1 Jednotkové testy ... 20

1.3.3.2 Integrační testy ... 20

1.3.3.3 Test-Driven-Development ... 20

1.4 VZOR REPOSITORY ... 20

1.5 VZOR FACTORY ... 21

1.6 INVERSION OF CONTROL A DEPENDENCY INJECTION ... 21

1.6.1 Dependency Injection ... 22

1.6.1.1 Constructor Injection ... 22

1.6.1.2 Setter Injection ... 22

1.6.1.3 Interface Injection ... 23

1.7 XAMARIN.FORMS ... 23

II PRAKTICKÁ ČÁST ... **24**

2 NÁVRH ZÁKLADNÍ ARCHITEKURY CELÉHO SYSTÉMU ... **25**

2.1 ANALÝZA PROBLÉMU A POPIS APLIKACE ... 25

2.2 ZÁKLADNÍ ARCHITEKTURA APLIKACE ... 25

2.2.1 Klienti v ukázkové aplikaci ... 25

Web ... 25

Mobilní aplikace ... 26

2.2.2 Server ... 26

3 NÁVRH IMPLEMENTACE ... **27**

3.1 WEBOVÁ APLIKACE ... 27

3.1.1 Technologie ... 27

3.1.2 Implementace ... 27

3.2 Mobilní aplikace ... 29
3.2.1 Technologie ... 29
3.2.2 Implementace ... 29
3.3 Webová služba ... 29
4 VYTVOŘENÍ UKÁZKOVÉ APLIKACE ... 32
4.1 API 32
4.1.1 Zabezpečení ... 32
4.1.2 Dependency Injection (DI) ... 34
4.1.3 Dokumentace ... 36
4.1.4 Ukázka zpracování GET požadavku ... 38
4.1.5 Ukázka zpracování PUT a DELETE požadavku ... 42
4.2 Web ... 43
4.2.1 Seskupení do oblastí ... 43
4.2.2 Ukázka oblasti Admin ... 43
4.2.3 Ukázka oblasti Employee ... 47
4.2.4 Ukázka registrace ... 48
4.3 Mobilní Aplikace ... 50
4.3.1 Přihlášení ... 51
4.3.2 Přehled smluv a detail smlouvy ... 54
ZÁVĚR ... 59
SEZNAM POUŽITÉ LITERATURY ... 61
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 64
SEZNAM OBRÁZKŮ ... 65
SEZNAM PŘÍLOH ... 66

ÚVOD

Webové technologie v současné době nabírají na velké popularitě. Doba, kdy klienti požadovali desktopové aplikace, se pomalu vytrácí. Samozřejmě ne vše jde přesunout na web, ale moderní technologie umožňují dříve nepředstavitelné. Při vývoji kvalitních a robustních řešení je potřeba udržovat přehledný, co nejméně provázaný, čistý kód. Neméně důležitý je i výběr vhodných technologií s vhodnou architekturou.

Pro pochopení problematiky jsou nutné základní znalosti programování a objektově orientovaného přístupu. Pro zasvěcení do problematiky, je zde teoretická část. Kde jsou vybrány doporučené postupy při vývoji. Jedná se o návrhové vzory, návrhové principy a techniky pro tvorbu aplikací. Dále také základní informace, týkající se vybraných technologií. V této části práce jsou také rozebrány základní architektury, které se využívají při vývoji softwaru.

V praktické části práce, je provedena analýza problému a navržení základní architektury celého systému. Dále je zde návrh implementace ukázkové aplikace, která na vybraných částech demonstruje doporučené postupy při tvorbě klient-server aplikací. Poslední kapitolou praktické části, je vytvoření samotné ukázkové demo aplikace, na které jsou předvedeny vybrané techniky použité při vývoji. Tato demo aplikace má umožnit základní práci s celým systémem.

Důvodů pro zvolení tohoto tématu práce je hned několik. Jedním z důvodů, je zmíněná rostoucí popularita webových technologií a má snaha v tomto oboru působit. Dalším podnětem ke zvolení tohoto tématu byl návrh na vytvoření jednoduché přehledné aplikace, která bude nezávislá na třetích stranách (např. Google) a bude umožňovat základní přehled nad chodem firmy.

Cílem práce je shrnutí nejnovějších technologií a doporučených postupů při tvorbě klient-server aplikací a demonstrace klíčových prvků řešení na vytvořené ukázkové aplikaci.

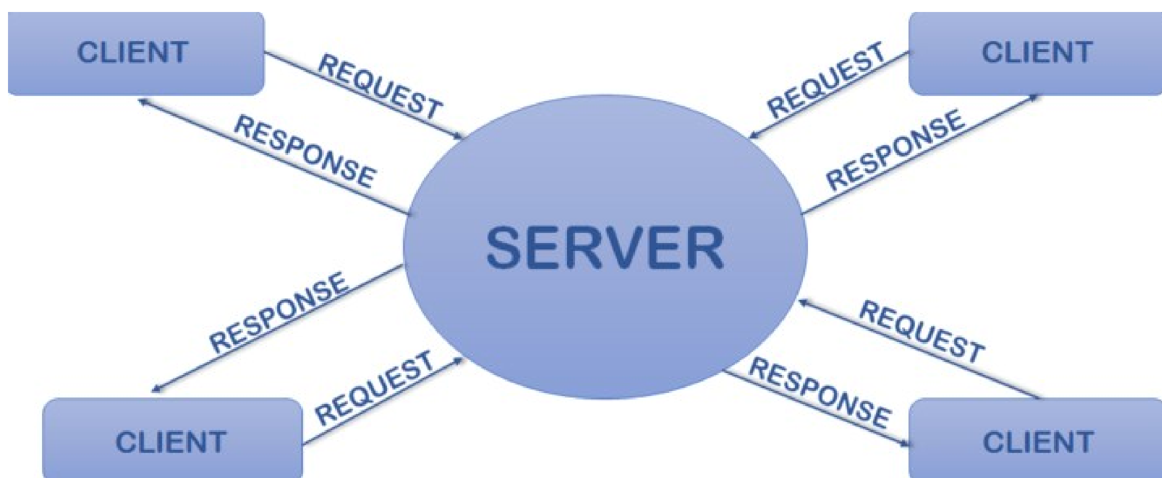
I. TEORETICKÁ ČÁST

1 TECHNOLOGIE A DOPORUČENÉ POSTUPY PRO TVORBU MULTIPLATFORMNÍCH KLIENT-SERVER A WEBOVÝCH APLIKACÍ

V této části je nejprve vysvětlena problematika klient-server. Dále je zde vysvětlen pojem framework a shrnuty informace týkající se vybraných technologií. Je zde také rozebrána architektura REST a doporučené postupy při vývoji aplikací. Konec této části je věnován multiplatformnímu vývoji.

1.1 Klient-Server

Architektura, popisující vztah mezi dvěma programy. Odděluje roli klienta, kde se nachází uživatelské rozhraní. Na straně klienta se vytvoří požadavek a odešle se na server, kde se požadavek zpracovává. Výsledné data se odesílají zpět na stranu klienta, kde se zobrazí. Příkladem může být webový prohlížeč (klient), pomocí kterého uživatel posílá dotazy na server a odpověď serveru se zde zobrazí. Tento způsob komunikace je jedním ze základních způsobů komunikace po síti, využívá jej například protokol HTTP.



Obrázek 1 Princip architektury Klient-Server [1]

Na jeden server se může dotazovat více klientů a stejně tak jeden klient se může dotazovat na více serverů. [1]

1.2 Framework

Frameworky poskytují vývojářům základy, na kterých lze vyvíjet efektivněji, hlavně z hlediska úspory času. Vývojář není nucen v každé aplikaci psát základní prvky pořád dokola. Frameworky většinou řeší problémy týkající se bezpečnosti, obsluhu zařízení, ale třeba také

komunikaci po síti, nebo autorizaci. Toho jsou schopny docílit díky předem vytvořeným knihovnám.

1.2.1 .NET Framework

Tento framework byl vyvinut společností Microsoft v roce 2002. Pro tento framework byl vytvořen zcela nový jazyk C#. Tento framework obsahuje dvě hlavní části. CLR (Common Language Runtime). CLR lze přirovnat k JVM (Java Virtual machine). Tedy zdrojový kód je zkompilován do přechodového jazyka a Just-In-Time kompilátor, generuje nativní kód, který se stará o běh aplikace. [2] CLR se mimo to stará o bezpečnost, typovou kontrolu a debugger. Také ovšem obsahuje garbage collector (GC), který se stará o uvolňování paměti. Což je velmi pohodlné a programátorovi v mnohém usnadňuje práci. Druhou části je obrovská knihovna tříd FCL (Framework Class Library). V době vzniku první verze tohoto frameworku, už tato knihovna obsahovala 3000 tříd. Třídy jsou organizovány do jmenných prostorů, jenž usnadňují navigaci mezi seskupenými třídami, ale také je umožněno použít stejné názvy tříd, jestliže se nachází v jiném jmenném prostoru. Díky tomuto frameworku, jsme schopni vytvářet webové aplikace (ASP.NET), desktopové aplikace pro Windows (Windows Forms), ale také konzolové aplikace. [3]

1.2.2 ASP.NET

ASP.NET se využívá při vývoji webových aplikací, nebo web API. Je následovníkem Active Server Pages (ASP). Novější verzí je ASP.NET MVC, jenž implementuje návrhový vzor MVC. Poslední verzí je ASP.NET Core 2.0. Který je nově open source a multiplatformní. Tato verze je zatím nejrychlejší z této rodiny. Framework se dělí na velké množství tzv. NuGet balíčků. Používá modulární přístup, tedy podle potřeby aplikace stačí požadovaný modul stáhnout a začít používat. [3]

- **Cross-platform:**

  Umožňuje implementovat funkcionality aplikace, bez ohledu na cílovou platormu. Současně podporuje tři největší cílové platformy (Windows, Linux a MacOS. Napsaná aplikace běží beze změn na všech podporovaných systémech.

- **Open source:**

  .NET Core jako open source podporuje transparentnější vývojový proces a udržuje aktivní komunitu. Je k dispozici na GitHubu.

- **Modularita:**

  .NET Core je modulární, je uvolňován prostřednictvím „NuGet“ v menších balíčcích. Spíše než jedna velká sestava, která obsahuje většinu základních funkcí. Je tedy k dispozici po menších částech, které jsou zaměřeny na určité funkce. To umožňuje zejména optimalizovat aplikaci tak, aby obsahovala pouze potřebné části. [4]

1.2.3 REST a web API

Roy Fielding v roce 2000 uvedl Representational State Transfer neboli REST, ve své dizertační práci. Byl zároveň spoluautorem, jednoho z nejznámějších protokolů, a to protokolu HTTP (Hypertext Transfer Protocol). REST architektura je orientována datově, například oproti architektuře SOAP (Simple Object Access Protocol). Jednou z hlavních výhod této architektury je to, že používá otevřené standardy a nezavádí tedy žádné specifické implementace. Lze tomu rozumět tak, že API lze napsat například v .NET Core a klientskou stranu v jakémkoli jazyce, který je schopen generovat HTTP požadavky a zpracovávat odpovědi ze serveru. Odpovědi jsou posílány v počítačově čitelném formátu, obvykle JSON (Javascript Object Notation), nebo XML (eXtensible Markup Language). Tento protokol implementuje čtyři základní operace s daty CRUD (CREATE, READ, UPDATE, DELETE), pomocí metod protokolu HTTP. [5]
Základní metody, které implementuje většina webových služeb:

- **POST (CREATE)**
  Metoda vytvářející nový zdroj na specifikované URI. Tělo požadavku nese informace o vytvářeném zdroji. Navíc touto metodou lze i spouštět operace, které žádné zdroje nevytváří, například pokud nechceme parametry předávat přes URI.

- **GET (READ)**
  Základní metodou pro přístup ke zdrojům je metoda GET. Pomocí této metody HTTP protokolu, lze získat data ze zdroje.

- **PUT (UPDATE)**
  Vytvoří nebo nahradí zdroj na požadované URI. Jako u metody POST, tělo požadavku nese informace o vytvářeném, nebo aktualizovaném zdroji.
- **DELETE**
  Smaže zdroj na definované URI.

Důležitým principem funkčnosti tohoto protokolu jsou bezstavové požadavky. Tedy jednotlivé dotazy na službu spolu nesouvisí a můžou přicházet v libovolném pořadí. Webová služba, která implementuje REST, by také měla vracet stavový kód znázorňující, jestli požadavek byl či nebyl úspěšně vyřízen. [5]

Základní stavové kódy jsou:

- **200** OK – úspěšně vyřízený požadavek.
- **400** Bad Request – požadavek nebyl vyřízen, jelikož byl syntakticky nesprávný.
- **401** Unauthorized – pro přístup ke zdrojům je nutná autorizace.
- **403** Forbidden – tento stavový kód může webová služba vrátit, pokud byl požadavek v pořádku, ale vývojář přistupuje ke zdroji, který server odmítá předat.
- **404** NotFound – zdroj nenalezen.
- **500** Internal Server Error – obecná chybová hláška znázorňující, že nastala neočekávaná chyba na straně serveru. [6]

1.2.4 Entity Framework a Objektově Relační Mapování (ORM)

Zprostředkovává konverzi dat z relační databáze do objektů objektově orientovaného jazyka. Vývojář tedy poté pracuje s daty jako se specifickými objekty. Jedním z takových mapperů, je například Entity Framework, který .NET framework obsahuje.

Entity Framework

Umožňuje využití přístupu Code First, kdy se databázový model popisuje pomocí tříd jazyka C#. Popsání modelu má svou konvenci, která je uvedena v dokumentaci tohoto frameworku. Například jako primární klíč výsledné tabulky je zvolena vlastnost třídy s názvem ID. Z takto napsaných tříd dokáže tento framework vytvořit celou databázi, či doplnit tabulky do databáze stávající. Eliminuje množství vývojářem napsaných SQL dotazů na minimum, avšak tyto dotazy generuje na pozadí. Stěžejní třídou pro práci s tímto frameworkem, je třída

DbContext, která nese informace o datových objektech a zprostředkovává interakci s databází. [7]

1.3 Doporučené postupy

1.3.1 Návrhové principy

Návrhové principy popisují, jak by měl programátor pracovat v objektově orientovaných jazycích. Popisují doporučené postupy objektového návrhu. Cílem je vytvořit návrh, který nebude vykazovat vlastnosti jako ztuhlost, křehkost či špatnou znovupoužitelnost. Tedy vlastnosti, které si vlastně protiřečí s filozofií objektově orientovaného programování (OOP). Hlavním cílem je omezit závislosti mezi jednotlivými částmi na maximum. Tedy co nejvíce je izolovat od zbytku kódu. Vývojář se poté nedostane do situace, kdy je potřeba použít určitou část znovu, ale díky závislostem a špatnému designu, tuto část radši napíše celou znovu. Tím se dostává do konfliktu s jedním ze základních návrhových principů a to „Don't Repeat Yourself" (DRY). Dodržováním těchto principů, jednoznačně vede k mnohem čistšímu kódu. [8]

1.3.1.1 SOLID

Autorem těchto principů je Robert "Uncle Bob" C. Martin. Definoval pět principů, kterými by se měl programátor jak při vývoji, tak při návrhu řídit. [8]

Single Responsibility Principe – SRP

Princip jedné odpovědnosti, třída by měla mít jen jednu zodpovědnost, tedy jediný důvod ke změně. Tato zodpovědnost by měla být jasně vystižena názvem třídy. Dodržováním tohoto principu, tedy dostaneme software složený z většího množství jednodušších tříd, které jsou přehlednější a při úpravě umožní rychleji najít odpovědné místo za danou situaci. Existuje pomůcka, pokud nelze popsat zodpovědnost třídy jednoduchou větou, pravděpodobně porušuje tento zákon. [8]

Open-Closed Principle – OCP

Princip otevřenosti a uzavřenosti. Třída by měla být otevřena pro rozšíření, ale uzavřena pro změny. Tudíž vývojář, by měl být schopen kód pouze přidat, nikoliv upravovat kód stávající. Neměla by nastat situace, kdy při úpravě, programátor neúmyslně poškodí funkčnost jiné komponenty. Podle autora těchto principů, je tento princip nejdůležitější ze všech, ovšem ne

úplně vždy je možné ho přesně dodržet. Najdou se situace, kdy je programátor nucen změnit existující kód. [8]

Liskov Substitution Principle – LSP

Odvozené třídy nesmí nikdy vyžadovat více a poskytovat méně než bázová třída. Tedy pokud nějaká komponenta používá bázovou třídu, musí fungovat i v případě použití jakékoliv její podtřídy. [8]

Interface Segregation Principle – ISP

Tento princip říká, že více specifických rozhraní je lepší než jedno univerzální. Pokud je definováno jedno univerzální rozhraní, ztrácí se přehlednost. Změna v tomto rozhraní, by vedla k případné úpravě všech uživatelů tohoto rozhraní. Je tedy lepší části rozhraní separovat a pro každého uživatele vytvořit rozhraní vlastní. [8]

Dependency Inversion Principle – DIP

Tento princip říká, že závislost konkrétního by měla být vždy na abstraktním, ne naopak. Závislosti tříd by tedy měli vést od konkrétních k abstraktním. Závislosti by měli být směřovány k rozhraní (Interface) a abstraktní třídě, ne ke konkrétní implementaci. Implementace se mění častěji než rozhraní, tedy závislost by měla být vůči stabilnější části. Pokud je třída závislá na abstraktním rozhraní, je pro vývojáře jednodušší konkrétní implementaci vyměnit za jinou. Dodržování tohoto principu, vede k výrazně lepší flexibilitě kódu.

Název SOLID je akronymem prvních písmen těchto principů. [8]

1.3.1.2 Law of Demeter

Deméteřin zákon definuje s jakými objekty bychom měli komunikovat a s kterými naopak ne. Opět se dodržením tohoto zákona více izoluje samotný objekt čili výsledný kód bude o to méně provázán. Metoda třídy by měla volat pouze:

1.) Metody vlastní třídy
2.) Metody objektů, které sama vytvořila
3.) Metody objektů, které byly předány jako argument
4.) Metody objektů, které jsou vlastnictvím třídy

Tudíž metoda by nikdy neměla komunikovat s objekty, ke kterým získá přístup přes prostředníka. Hlavní výhodou je samozřejmě menší provázanost tříd mezi sebou, pokud je zákon dodržen, třída má plnou zodpovědnost za svůj vnitřní stav. Nevýhodou můžou být rozsáhlejší rozhraní, které musí obsahovat mnohem více metod. Ovšem o to lepší je poté zapouzdření a následné použití této komponenty. Mohou nastat situace, kdy tento zákon musí vývojář porušit, ovšem vždy musí předem zvážit, jestli je to opravdu nutné. [9]

1.3.1.3 Don’t Repeat Yourself

Jak název napovídá, vývojář by nikdy neměl psát tu samou část znova, tedy netvořit duplicitní kód. Ovšem nejedná se jen o kód, ale o všechny informace napříč celým projektem. Nedodržováním se do aplikace dostává mnoho prostoru pro chyby. Při odhalení chyby a nedodržování DRY, se vývojář vystavuje riziku, že při opravě chyby neopraví všechny místa, na kterých se tato chyba vyskytuje čili samotný zdrojový kód je poté velmi těžké opravovat nebo upravovat. Při úpravě kódu, nedodržujícího tento princip, je dokonce schopen roznést neupravením všech míst do celého systému nové chyby. Ovšem ne každá duplicita je porušením, nemusíme vždy extrahovat určité části kódu do nové metody, pokud spolu například logicky nesouvisí. Vždy se vývojář musí zamyslet nad tím, zda se jedná o porušení DRY, nebo nikoliv. Přílišné kladení důrazu na toto pravidlo se může projevit zvyšující se provázanost kódu, což je zajisté naopak nevýhodou. [10]

1.3.2 Architektura

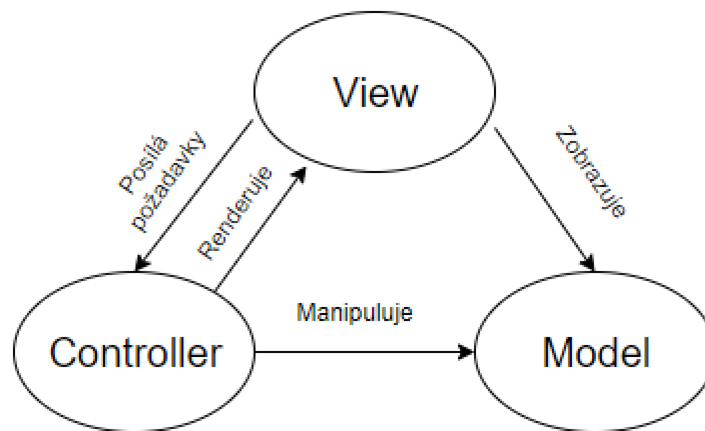
1.3.2.1 MVC

Tento vzor je odlehčený, dobře testovatelný a hraje důležitou roli u různých frameworků (např. ASP.NET MVC). Obsahuje tři hlavní komponenty. Model, View a Controller.

Model je objekt, reprezentující data a business logiku. Nenese ovšem žádnou informaci o tom, jak mají být uživateli zobrazena.

View je reprezentací modelu v očích uživatele. View se stará jen o zobrazení informací, tudíž funguje jako uživatelské rozhraní.

Controller zpracovává požadavky uživatele, pracuje s modelem a aktualizuje View. Má na starosti aplikační logiku a tok události v aplikaci.



Obrázek 2 Schéma znázorňující architekturu MVC

Tento návrhový vzor tedy pomáhá separovat rozdílné aspekty aplikace. Vstupní logiku (Controller), business logiku (Model) a logiku uživatelského rozhraní (View), ovšem poskytuje propojení mezi nimi. [11]

Při tvorbě aplikací je to velmi výhodné, jelikož se vývojář například může zaměřit na view bez závislosti na business logice. Naopak při práci s modelem ho zajímá například komunikace s databázi, nebo business logika a může se na to plně soustředit a nemusí ho v tu chvíli zajímat samotná prezentace dat.

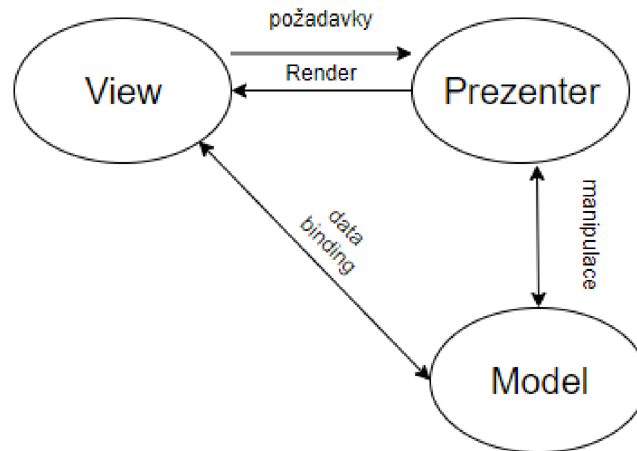
Mezi hlavní výhody aplikací založených na MVC vzoru patří:

1.) Plná kontrola nad chováním aplikace.
2.) Mnohem lepší použitelnost při testování.
3.) Vhodné pro webové aplikace (v této oblasti se také využívá nejčastěji), které jsou vyvíjeny velkými týmy programátorů. [11]

1.3.2.2 MVP

Tato architektura Model-View-Presenter je velmi podobná architektuře MVC. Také obsahuje tři hlavní komponenty a to Model, který je totožný s modelem ve vzoru MVC. View, které nyní navíc dokáže zpracovávat zásahy uživatele. Ovšem View stále nenese žádnou aplikační logiku, pouze zpracuje událost a zavolá metodu další komponentu. Komponenta Presenter, která v sobě nese obslužnou logiku a aktualizuje samotný Model. Aktualizaci View zajistí buď systém notifikací, nebo také Presenter přímo aktualizuje View. Podle toho, jak velkou roli hraje samotné View, Martin Fowler rozdělil tuto architekturu na dva druhy. [12]

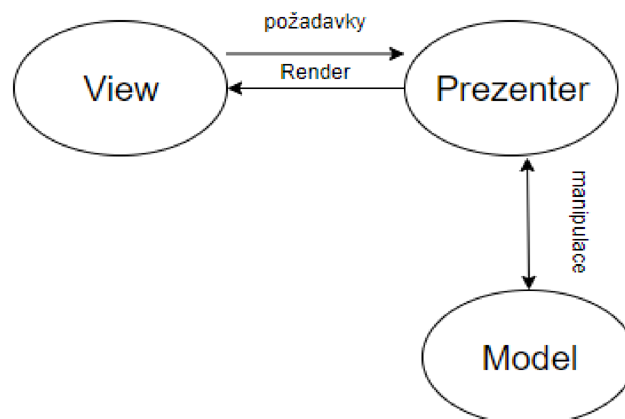
Supervising Controller



Obrázek 3 Schéma návrhového vzoru MVP Supervising Controller

Martin Fowler nejspíše slovo Controller zvolil trochu předčasně, protože se opravdu jedná spíše o Prezenter. Prezentační logika je rozdělena na dvě části. Controller (Prezenter) a View. View je odpovědné za zobrazení dat, které nese Model, pomocí data bindingu. Prezenter zasahuje pouze tehdy, pokud je potřeba složitější logika View, s kterým dokáže manipulovat dle potřeby. [12]

Passive View



Obrázek 4 Schéma návrhového vzoru MVP Passive View

Zmizela vazba mezi View a Modelem. View je nyní v naprosto pasivní formě a je tedy otrokem Prezenteru. Nemůže aktualizovat samo sebe přímo z modelu. Všechna UI logika je obsažena v Prezenteru, který nyní kromě prezentační logiky řeší také práci s modelem. Passive View se snaží funkce uživatelského rozhraní dostat na minimum. Výhodou je psaní automatických testů, které se píší mnohem snáze. Jelikož testování je jeden ze základních a stěžejních kroků při vývoji, toto usnadnění je velmi důležité. [13]

Tento návrhový vzor se nejčastěji využívá pro vývoj desktopových aplikací.

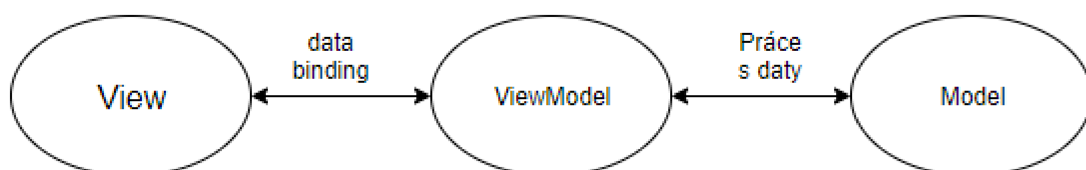
1.3.2.3 MVVM

Tento návrhový vzor Model-View-ViewModel je určen primárně pro vývoj WPF aplikací. Odděluje logiku aplikace od UI. WPF bylo vytvořeno tak, aby použití MVVM bylo snadné a rychlé. Opět je snaha separovat logiku od uživatelského rozhraní. [14]

Model je opět stejný, obsahuje jak data, tak i business logiku.

View je odpovědné za vzhled uživatelského rozhraní. Definováno za pomocí XAML. Data k zobrazení, View získává data k zobrazení pomocí volání metod ViewModelu, nebo obousměrného data bindingu.

ViewModel je třída uchovávající si stav aplikace. Této třídy se dotazuje UI a podle toho vykresluje patřičné prvky uživateli. Tato třída je nejdůležitější. Poskytuje data v takových strukturách, které jsou schopny vyvolat událost při změně. Tím je dosaženo okamžitého zobrazení změněných dat uživateli. [14]



Obrázek 5 Schéma návrhového vzoru MVVM

1.3.3 Testování

Mezi doporučené postupy při vývoji neodmyslitelně patří testování. Je zásadním pro kvalitní a funkční software. Při rozsahu dnešních aplikací a velikostí týmů, jenž se podílí na vývoji,

je zajisté potřeba například prověřit, jestli nově přidaná funkcionalita nikterak neovlivní stávající stav aplikace. [2]

1.3.3.1 Jednotkové testy

Tyto testy se také nazývají Unit testy neboli testy jednotek. Testují metody tříd jednu po druhé. Předají jednotce vstupy a zjišťují, jestli výstup je v pořádku. Dobrým zvykem je, aby vývojář k dané funkcionalitě napsal jednotkový test, který ověří její chování. Pokud metoda komunikuje s jinou komponentou, používá se tzv. Mock objekt. Tento pomocný objekt simuluje chování objektu reálného. Otázkou je, na co všechno tyto testy psát. Psaní těchto testů, zabere nějaký čas, proto je vhodné předem si definovat, na co vše se tyto testy budou psát. [15] [16]

1.3.3.2 Integrační testy

Tyto testy dohlíží na komunikaci komponent, vyvíjených separovaně, mezi sebou. Testují, zda komponenty správně komunikují a vše funguje, jak má. Například místo Mock objektu, se použije objekt reálný. [17]

1.3.3.3 Test-Driven-Development

Vývoj řízeny testy TTD, je technika použití automatizovaných testů, jenž řídí návrh softwaru. Výsledkem takového postupu je hodně komplexní sada testů, které po spuštění poskytnou zpětnou vazbu o stavu systému. Tato metodika je jednou z agilních metodik vývoje. Základním motem při tomto postupu vývoje, je „Red/Green/Refactor“. [18]

Red Tvorba testu

Green Vývoj kódu, bez ohledu na kvalitu, hlavním cílem je, aby test úspěšně prošel.

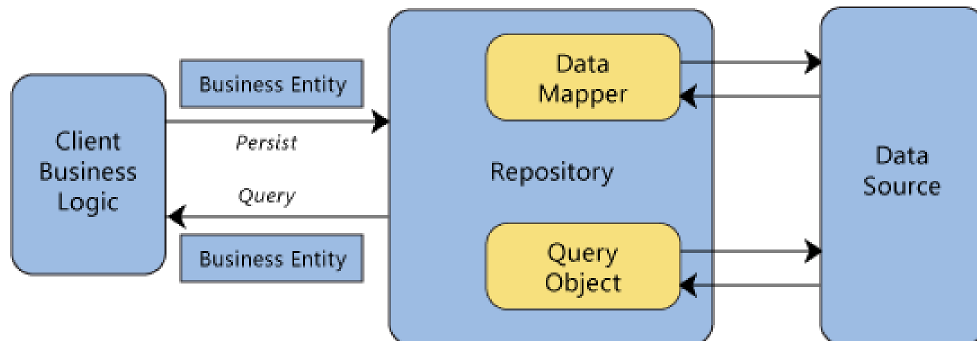
Refactor Úprava kódu do elegantnější a čisté formy, ale zároveň kontrola, zda test pořád prochází.

Tento cyklus se neustále opakuje pro každou novou funkcionalitu. Výhodou je neustálá zpětná vazba od systému. [18]

1.4 Vzor Repository

Používá se k oddělení business logiky a přímého získání a mapování dat. Použití je vhodné především pokud vývojář chce mít co největší množství kódu testovatelného a izolovaného od datové vrstvy. Dále je zde místo pro implementaci „cachování“ dat,

které je vhodné využít, pokud se dotaz často opakuje a data zůstávají povětšinou stejná. Využití tohoto vzoru tedy vedle k lepší testovatelnosti, přehlednosti a udržitelnosti kódu. [19]



Obrázek 6 Vzor Repository [19]

1.5 Vzor Factory

Tento návrhový vzor definuje rozhraní pro vytváření objektů. Pokud by programátor v rozsáhle aplikaci vytvářel jednotlivé instance pomocí "new" a následně například do konstruktoru přidal další parametr, tak na všech místech aplikace, bude donucen tuto změnu zanést. Čím větší aplikace, tím větší komplikace by tato změna do celého systému zanesla. Například pokud se v aplikaci pracuje s databází, tak při změně adresy serveru, na kterém se databáze nachází, by se musela tato změna zavést do všech repositářů, které s touto databází pracují. Proto je vhodné zavést vzor Factory, který se postará o vytvoření instance třídy umožňující připojení a samotné otvírání a zavírání transakcí už se přenechá danému repositáři. Tento vzor se tedy zabývá samotným procesem vytvoření instance a umožňuje rychlejší práci, zejména v rozvíjejícím se systému. [20]

Obrázek 7 Vzor Factory [20]

1.6 Inversion of Control a Dependency Injection

Inversion of Control (IoC), je technika, která mění běh programu. Umožňuje vytvářet instance tříd, bez nutnosti existence reference na instanci požadované třídy. Jedna z implementací tohoto vzoru je Dependency Injection.

1.6.1 Dependency Injection

Neboli vkládání závislostí, má za cíl dosažení volného spojení mezi objekty. Nejčastěji třídy vyjadřují své závislosti pomocí konstruktoru. Toto je také známo jako Constructor Injection. [21]

1.6.1.1 Constructor Injection

Podstatou je odebrat třídám zodpovědnost za získávání instancí objektů, potřebných k činnosti, místo toho jim tyto instance předávat při vytváření.

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly IDbConnectionFactory _dbConnectionHandler;

    public EmployeeRepository(IDbConnectionFactory connection)
    {
        _dbConnectionHandler = connection;
    }
}
```

Ukázka kódu, kde třída poskytující repositář zaměstnanců potřebuje přistupovat do databáze. Objekt zprostředkovávající přístup do databáze, je předán této třídě v konstruktoru zvenčí. Veškeré další případné závislosti se tedy vloží jako parametry konstruktoru. Toto vkládání závislostí je asi nejčastější metodou. [22]

1.6.1.2 Setter Injection

Další častou variantou vkládání závislostí, je Setter Injection. Vkládání závislostí pomocí setteru. Jedná se o tu samou situaci, jen je závislost předána v setteru a ne v konstruktoru třídy. [22]

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly IDbConnectionFactory _dbConnectionHandler;

    public void SetDbConnection(IDbConnectionFactory connection)
    {
        _dbConnectionHandler = connection;
    }
}
```

1.6.1.3 Interface Injection

Tento postup není tak častý, jedná se o stejnou metodiku jako setter Injection. Jen metoda pro vložení závislostí je získána z rozhraní. Ukázka jednoduchého kódu je níže.

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly IDbConnectionFactory _dbConnectionHandler;

    public void InjectDependencies(IDbConnectionFactory connection)
    {
        _dbConnectionHandler = connection;
    }
}

public interface InjectDependencies
{
    void InjectDependencies (IDbConnectionFactory connection)
}
```

Jednotlivé závislosti stačí přidat do metody, která se stará o vložení závislostí (InjectDependencies). [22]

1.7 Xamarin.Forms

Xamarin.Forms je framework umožňující rychle vytvářet multiplatformní uživatelské rozhraní. Poskytuje vlastní abstrakci pro UI, které je poté vykresleno pomocí nativních ovládacích prvků jednotlivých platforem (iOS, Android, Windows, Windows Phone). [23]

Aplikace dokážou sdílet velkou část kódu svého uživatelského rozhraní a zachovávají si vzhled dané platformy. Jelikož jsou aplikace stále nativní, nemají různá omezení, nebo špatný výkon. Tyto aplikace umožňují využívat API a ostatní vlastnosti cílené platformy.

Aplikace vytvořená pomocí Xamarin.Forms disponuje možností, že část aplikace je tvořena tímto frameworkem a část nativním UI toolkitem. [23]

Nejběžnější je klasický přístup k vývoji multiplatformních aplikací. Tedy použití přenosných knihoven, či sdílených projektů, které obsahují sdílený kód, jenž je poté využíván v aplikaci vytvořené pro specifickou platformu. [23]

II. PRAKTICKÁ ČÁST

2 NÁVRH ZÁKLADNÍ ARCHITEKURY CELÉHO SYSTÉMU

Tato část bakalářské práce, se zaměřuje na návrh celého systému, s důrazem na multiplatformní řešení. Nejdříve je navržena architektura systému jako celku, poté implementace aplikace za použití moderních technologií a doporučených postupů. Výběr zvolených technologií je odůvodněn. Následně je zde popis vytvoření samotné ukázkové aplikaci, která demonstruje tyto postupy na klíčových prvcích řešení.

2.1 Analýza problému a popis aplikace

Ukázková aplikace, demonstrující doporučené postupy při tvorbě aplikací, by měla umožňovat přehled nadřízenému nad jednotlivými projekty v malé firmě, která nechce být závislá na službách třetích stran a chce pouhý přehled nad výkonem zaměstnanců, bez zbytečných funkcionalit navíc. Zaměstnanci by zde měli vidět úkoly, které jim jsou přiřazeny a zapisovat jak strávený čas nad úkolem, tak procentuální stav, ve kterém se úkol nachází, dále by měla poskytovat report pro nadřízeného, který by měl být schopen základní správy nad zaměstnanci a projekty.

2.2 Základní architektura aplikace

Vzhledem k multiplatformnímu řešení, je celý systém rozdělen na jednotlivé a logicky seskupené celky. Použita je architektura klient-server, která obsahuje dva klienty. Jeden z těchto klientů, web, je napsán v ASP.NET Core MVC. Další klientskou stranou celého systému je mobilní multiplatformní aplikace s architekturou MVVM, napsána v Xamarin.Forms. Stěžejní části je webová služba napsána v .NET CORE a funguje jako server pro výše zmíněné klientské strany. Tato služba bude zprostředkovávat přístup k databázi a celou business logiku daného problému, klientské strany se budou připojovat na dané přístupové body, z kterých budou získávat potřebná data, pro zobrazení uživateli.

2.2.1 Klienti v ukázkové aplikaci

Web

Celý web, je napsán v ASP.NET Core 2.0 MVC, je sestrojen tak, aby sloužil jako hlavní rozhraní pro sběr dat od zaměstnanců a jejich reprezentaci nadřízenému, který zde vidí nákladovost projektů (jednotlivých smluv), průměrný procentuální stav zakázky, procentuální stavy dílčích úkolů, přiřazených zaměstnancům a také jednoduchou správu zaměstnanců,

kde je schopen zaměstnance vymazat, nebo mu povolit přístup do celého systému. Kromě toho zprostředkovává registraci do systému.

Mobilní aplikace

Tato klientská strana s architekturou MVVM, je napsána v Xamarin.Forms, který je multiplatformní. Je zaměřena hlavně pro nadřízeného, který zde vidí přehled smluv. Může tedy přistupovat k datům i bez nutnosti otevření prohlížeče.

2.2.2 Server

Server je vytvořen pomocí REST web API frameworku .NET Core. Služba je připojena k databázi, která nese data o uživatelích aplikace, jednotlivých projektech a úkolech, které jsou přiřazeny zaměstnancům. Úkol, který je přiřazen zaměstnanci, zároveň nese informaci o projektu, pod který spadá. Databáze u uživatele uchovává pouze tyto citlivá data:

- Jméno a příjmení
- Email, který musí být unikátní

Aplikace má demonstrovat, doporučené postupy při vývoji a moderní technologie, v aplikaci je poté pro ukázku využito mapování pomocí Entity Framework, tak přístup s mapperem Dapper. Dapper je ve srovnání s Entity Frameworkem rychlejší, avšak za cenu nutnosti ručního psaní databázových procedur.

3 NÁVRH IMPLEMENTACE

Tato část práce se zabývá rozebráním návrhu implementace jednotlivých částí a zdůvodnění výběru použitých technologií.

3.1 Webová aplikace

3.1.1 Technologie

Pro tuto část, je zvolena technologie ASP.NET Core MVC 2.0, tato verze frameworku, mimo jiné, obsahuje základní implementaci Dependency Injection a umožňuje velmi snadné přihlášení pomocí třetích stran. Mezi podporované poskytovatele autentizace patří Google, Facebook, Twitter a Microsoft. Což velmi usnadňuje práci na počátku vývoje i vzhledem k přicházející směrnici GDPR. Samotná autorizace, lze kdykoliv vyměnit za vlastní implementaci. Další z řady vylepšení oproti předchůdci (.NET 4.7), kvůli kterým jsem vybral tuto technologii, je rychlost. Starší verzi překonává v rychlosti operací nad kolekcemi, v rychlosti provádění matematických funkcí, serializaci a deserializaci dat, ale i v počtu vyřízených požadavků za sekundu. Samozřejmě záleží i na HW konfiguraci serveru, na kterém běží, ale při testech na stejném HW, disponoval v těchto a mnoho dalších ohledech větší rychlostí. Dalším důvodem je zmíněná modularita a možnost vystavení aplikace na server s operačním systémem Windows, ale i Linux, jelikož je „cross-platform".

https://blogs.msdn.microsoft.com/dotnet/2017/06/07/performance-improvements-in-net-core/

3.1.2 Implementace

Aplikace se chová jinak pro zaměstnance a jinak pro nadřízeného. Pro větší přehlednost je využito seskupení do dvou logických oblastí (Area).

Area Employee

Jedna nezávislá oblast je oblast zaměstnance, obsahující dva controllery EmployeeController a TaskController.

EmployeeController

- Zobrazení Indexu (titulní strany pro zaměstnance)
- Odstranění zaměstnance z databáze (implementováno vzhledem k GDPR)

TaskController

- Zobrazení detailu přiřazeného úkolu
- Přidání času do úkolu
- Odstranění času z úkolu

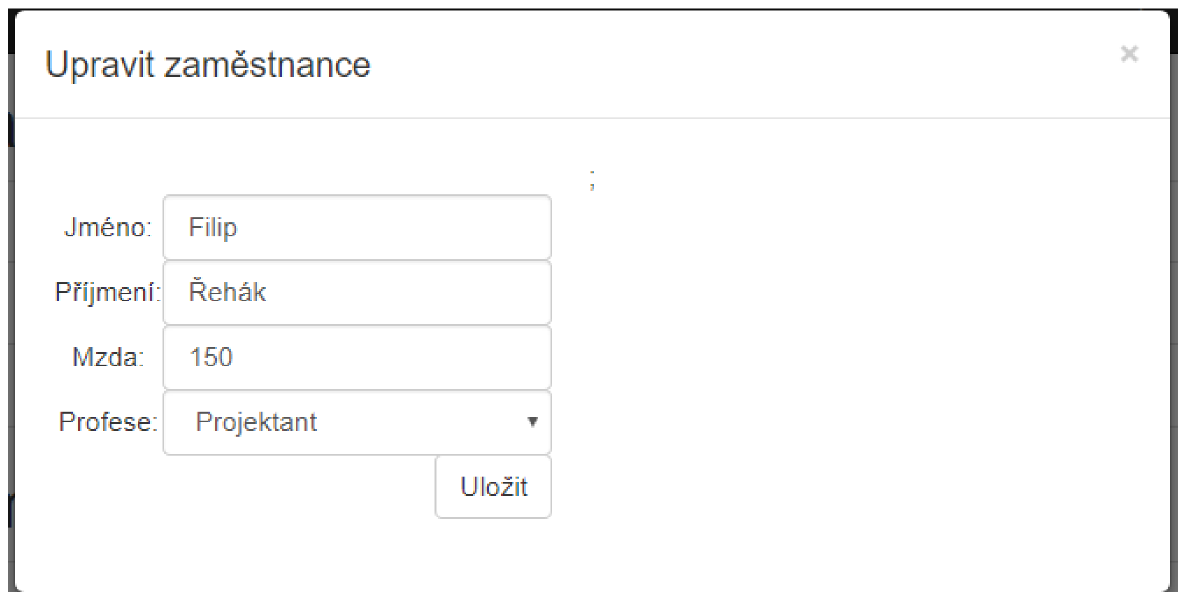
Controllery nevolají webovou službu přímo, ale logika volání služby je přesunuta do vrstvy service, pojmenovanou EmployeeService. V controlleru se tedy jen volá požadovaná metoda této třídy a popřípadě z vrácených dat sestaví ViewModel.

Tato oblast, dále obsahuje sadu tříd (ViewModelů), které slouží k řízení a plnění View.

Area Admin

Tato oblast obsahuje také dva controllery. AdminController, který obsluhuje události jako:

- Aktivace/ Deaktivace zaměstnance
- Smazání zaměstnance
- Úprava údajů zaměstnance (nastavení mzdy, úprava jména a přidělení profese)



Obrázek 8 Náhled na editaci zaměstnance

Dále tato oblast obsahuje ContractController, obsluhující události týkající se zakázek a přidělování úkolů.

Metody těchto controllerů opět volají webovou službu přes metody třídy AdminService. Je tu opět několik ViewModelů, jež jsou seskupeny do jmenných prostorů podle entity s kterou pracují.

Dále je v aplikaci implementována třída ViewService, která obsluhuje View. Je společná pro obě oblasti. Mimo jiné zprostředkovává data pro zajištění správného zobrazení v rozložení stránky, například tlačítka s patřičnými odkazy pro daného uživatele (dle role v aplikaci), nebo zobrazení tlačítka pro odhlášení z aplikace. Dále také tvoří například jednotky u nákladů a mezd pro lehkou modifikaci.

3.2 Mobilní aplikace

3.2.1 Technologie

Vzhledem k požadovanému multiplatformnímu řešení, je použit framework Xamarin.Forms s architekturou MVVM, která disponuje dvoucestným vytěžováním dat (two way data binding). Tento framework umožňuje napsat jeden kód v jazyce C#, který je poté sdílený napříč platformami. Je vhodný zejména pokud daná aplikace potřebuje co nejméně specifických funkcionalit dané platformy.

3.2.2 Implementace

Mobilní aplikace obsahuje následující stránky:

- LoginPage – slouží k přihlášení do aplikace
- MainPage – obsahuje navigaci (Layout)
- ItemsPage – obsahuje přehlede smluv
- ItemDetailPage – obsahuje detail smlouvy

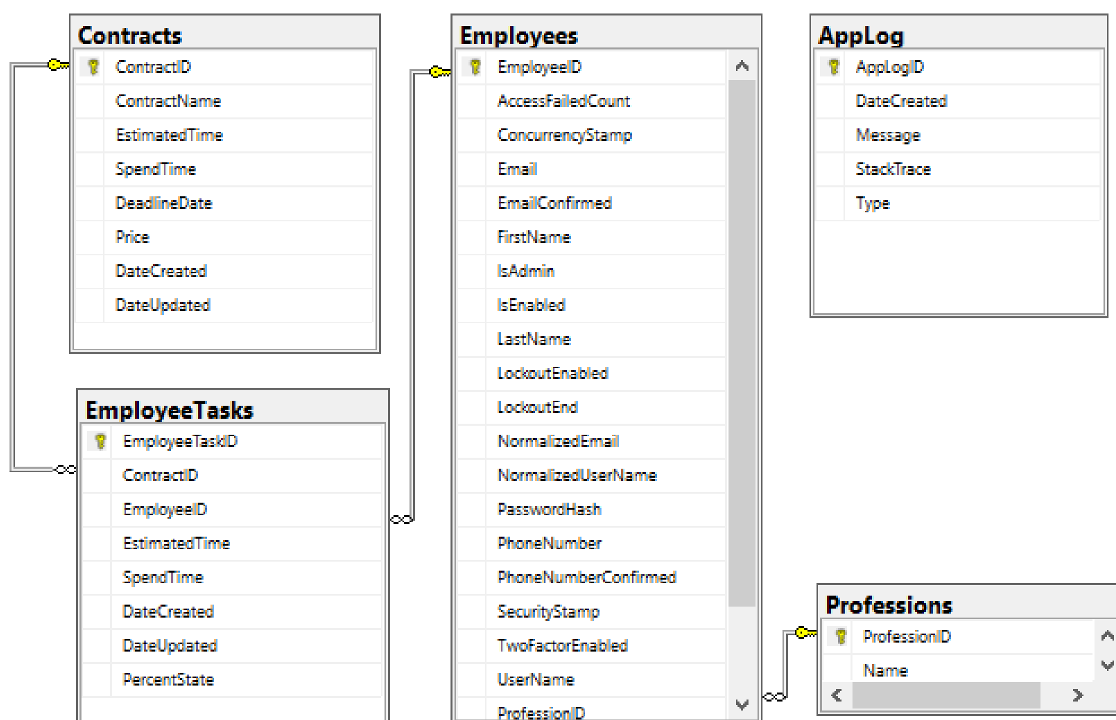
Stránky mají své ViewModely, které se starají o obsluhu události vyvolaných uživatelem. Tyto ViewModely volají metody objektu, který zprostředkovává komunikaci s webovou službou, pomocí které aplikace získává data pro zobrazení. Výsledná aplikace je díky Xamarin.Forms multiplatformní a v celkovém systému klient-server zastává roli klienta.

3.3 Webová služba

Pro vývoj je opět použita technologie ASP.NET Core 2.0. Webová služba je rozdělena na jednotlivé, logicky oddělené vrstvy. Každá entita, kterou služba disponuje, má vlastní Controller. Obsahuje tyto Controllery:

- **EmployeeController** – pomocí EmployeeRepository provádí operace nad zaměstnancem.
- **ContractController** – pomocí ContractRepository provádí operace nad smlouvami.
- **EmployeeTaskController** – pomocí EmployeeRepository provádí operace nad úkoly.
- **ContractDetailController** – pomocí ContractDetailProvider zprostředkovává detail smlouvy.
- **ContractTaskController** – pomocí ContractRepository zprostředkovává a přidává jednotlivé úkoly pod smlouvou.
- **ProfessionController** – pomocí ProfessionRepository zprostředkovává a přidává jednotlivé profese.

Tyto controllery mohou přistupovat k datům přes vrstvu repository, která se stará o načtení a mapování dat. V případě potřeby provedení výpočtu nad daty controller nevolá přímo repositář, ale vrstvu service, která se stará o získání dat přes vrstvu repository a následný výpočet nad získanými daty. Webová služba je RESTFull, data vrací ve formátu JSON. Data jsou poté v klientských stranách deserializovány, a to pomocí frameworku JSON.NET od společnosti Newtonsoft.



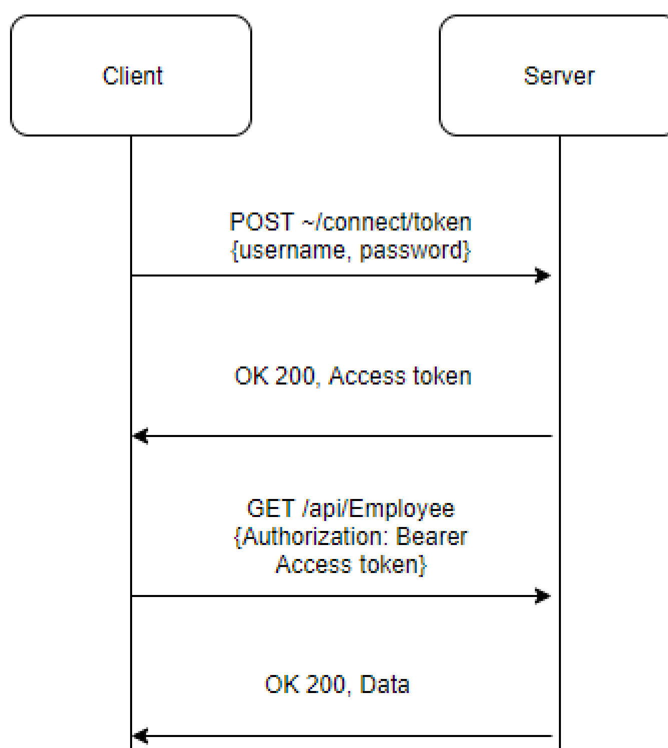
Obrázek 9 Databázový diagram

4 VYTVOŘENÍ UKÁZKOVÉ APLIKACE

V této části je rozebrána implementace samotné aplikace, na níž jsou demonstrovány doporučené postupy při vývoji multiplatformních klient-server aplikací. Jsou zde také vybrány techniky, jež byli zmíněny v teoretické části práce.

4.1 API

4.1.1 Zabezpečení



Obrázek 10 Autorizační proces

Na obrázku je znázorněn autorizační proces, který je v ukázkové aplikaci implementován. Klient pošle POST metodou na specifický koncový bod jméno a heslo zadané uživatelem. Na tomto koncovém bodě proběhne zpracování dat. Pokud je vše v pořádku, vrátí přístupový token. Poté klient s každým dalším požadavkem na server posílá tento přístupový token v autorizační hlavičce požadavku, tím je docíleno bezstavovosti, o které se píše v teoretické části této práce. Pokud se klient bez přístupového tokenu dotazuje na koncové body vyžadující autorizaci, server vrátí stavový kód 401. Tímto server informuje o tom, že pro získání dat musí být provedena autorizace.

Je zde implementován OpenID Connect, což je autentizační protokol založený na OAuth2 specifikaci. Lze ho použít v každé .NET Core aplikaci. Umožňuje přístup klientům jak z prohlížeče, tak z mobilních aplikací. Implementace začíná stažením potřebných nuggetů. [24]

Jedná se o následující moduly

- OpenIddict (verze 2.0.0)
- OpenIddict.EntityFrameworkCore (verze 2.0.0)
- OpenIddict.Mvc (verze 2.0.0)

Následně je nutné provést konfiguraci v metodě ConfigureServices třídy Startup. Je zde nakonfigurováno použití Entity Frameworku.

```
services.AddDbContext<ApplicationDbContext>(options =>
{
    // Configure the context to use SQL Server.
    options.UseSqlServer(Configuration.GetConnectionString("db"));

    options.UseOpenIddict();
});
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

Poté se nastaví autentifikace a konfigurují vyžadované claimy

```
services.AddAuthentication().AddOAuthValidation();
services.Configure<IdentityOptions>(options =>
{
    options.ClaimsIdentity.UserIdClaimType = OpenIdConnectCon-
stants.Claims.Subject;
    options.ClaimsIdentity.UserNameClaimType = OpenIdConnectCon-
stants.Claims.Name;
    options.ClaimsIdentity.RoleClaimType = OpenIdConnectCon-
stants.Claims.Role;
});
```

Nakonec se registruje a konfiguruje samotné OpenID

```
services.AddOpenIddict(options =>
{
    options.AddEntityFrameworkCoreStores<ApplicationDbContext>();
    options.AddMvcBinders();
    options.EnableTokenEndpoint("/connect/token");
    options.AllowPasswordFlow();
    //development purposes
```

```
    options.DisableHttpsRequirement();
});
```

Zde je nastaven DbContext pro tento framework, dále je zde specifikován koncový bod poskytující samotný token a pro vývojové účely je zde vypnut požadavek na zabezpečený HTTPS protokol.

Dále je vytvořen AuthorizationController, který provádí Generování tokenu. O hledání uživatele v databázi a kontrolu hesla se stará výše konfigurovaný Entity Framework. Hledání uživatele podle jména je provedeno pomocí UserManageru, který je vlastností třídy.

```
var user = await _userManager.FindByNameAsync(request.Username);
if (user == null)
{
    return BadRequest(new OpenIdConnectResponse
    {
        Error = OpenIdConnectConstants.Errors.InvalidGrant,
        ErrorDescription = "The username/password is invalid."
    });
}
```

Pokud je uživatel nalezen, následuje kontrola hesla

```
var result = await _signInManager.CheckPasswordSignInAsync(user,
request.Password, false);
if (!result.Succeeded)
{
    return BadRequest(new OpenIdConnectResponse
    {
        Error = OpenIdConnectConstants.Errors.InvalidGrant,
        ErrorDescription = "The username/password couple is invalid."
    });
}
```

Pokud heslo patří danému uživateli, následuje vytvoření tokenu a SignInResultu, který se vrací ve formátu JSON klientovi.

4.1.2 Dependency Injection (DI)

Registrace

Registrování vlastních servis do kontejneru probíhá v souboru Startup.cs. Pro základní práci a ukázku využití postačuje implementace od Microsoftu, která je součástí frameworku .Net Core. Pokud je potřeba pokročilých funkcionalit, jako podmíněné registrace, je možno použít například implementaci DI od společnosti Autofac, která se využívá při rozsáhlých projektech.

Registrace se provádí v metodě ConfigureServices třídy Startup. Pro registraci vlastních tříd je vytvořena privátní metoda RegisterAppServices, která jako parametr přebírá kolekci registrovaných služeb. Tato kolekce obsahuje metody pro registraci dalších tříd do kontejneru. Lze je registrovat například jako Transient nebo Singleton. Tím se definuje, kdy se má instance vytvářet a kdy zanikat. Čili život (lifetime) samotné instance.

- Transient – instance jsou vytvářeny pokaždé, kdy jsou požadovány. Vhodné pro jednoduché bez stavové třídy.
- Singleton – instance je vytvořena jednou, a každý další požadavek používá stejnou instanci.

```
private void RegisterAppServices(IServiceCollection serviceCollection)
{
 serviceCollection.AddSingleton<IDbConnectionFactory,
                              DbConnectionFactory>();
    serviceCollection.AddTransient<IEmployeeRepository,
                              EmployeeRepository>();
    serviceCollection.AddTransient<IAdminRepository,
                              AdminRepository>();
    serviceCollection.AddTransient<IContractRepository,
                              ContractRepository>();
    serviceCollection.AddTransient<IContractDetailProvider,
                              ContractDetailProvider>();
}
```

Metody AddTransient a AddSingleton jsou generické. První se definuje interface dané třídy, poté samotný typ kterého instance bude kontejnerem vytvořena.

Ukázka vložení závislosti (Inject)

```
public class ContractDetailProvider : IContractDetailProvider
{
    private readonly IContractRepository _contractRepository;

    public ContractDetailProvider(IContractRepository contractRepository)
    {
        _contractRepository = contractRepository;
    }
}
```

Třída ContractDetailProvider je závislá na existenci instance třídy ContractRepository. Své závislosti si definuje v konstruktoru. Jelikož je třída ContractRepository zaregistrována, vytvoří se její instance a předá se do konstruktoru, kde se uloží jako privátní vlastnost třídy.

Tímto je vytvoření instance třídy ContractRepository hotové a je možné využívat její metody. V případě potřeby se další závislosti definují opět v konstruktoru jako další parametry a ve třídě se vytvoří další privátní vlastnosti do kterých se vytvořené instance předají.

V případě, že by služba nebyla registrována instance nebude vytvořena a bude vyvolána výjimka, která informuje, že daný typ nebyl v kontejneru nalezen.

4.1.3 Dokumentace

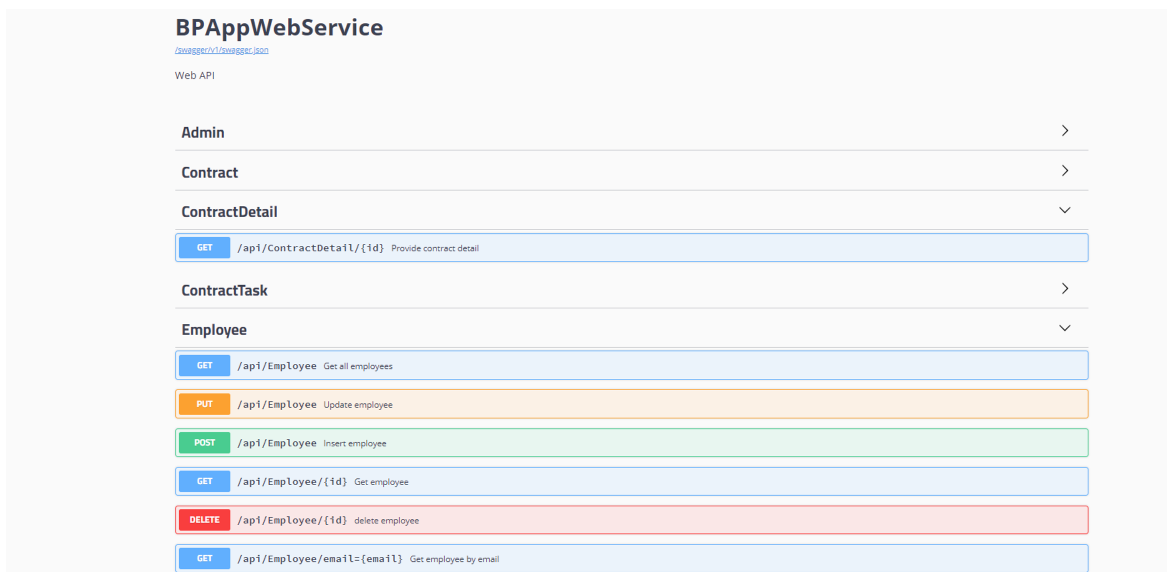
Pro tvorbu dokumentace API tedy popis jednotlivých koncových bodů, je použito frameworku Swagger. Pomocí Nugget Package Manageru stačí stáhnout a v metodě třídy Startup ho konfigurovat.

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info { Title = "BPAppWebService", Description =
    "Web API" });
    var path =
      $"{System.AppDomain.CurrentDomain.BaseDirectory}BPAppWebService.xml";
    c.IncludeXmlComments(path);
});
```

Poté nastavit použití Swaggeru v metodě Configure třídy Startup.

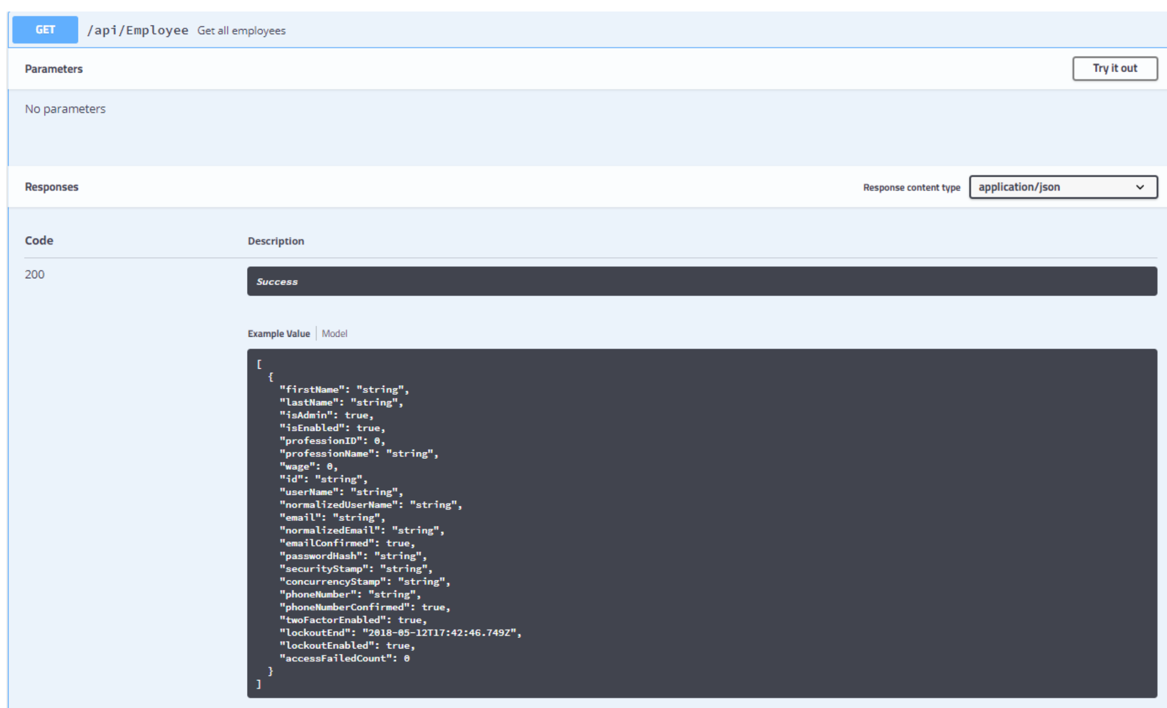
```
app.UseSwagger();
app.UseSwaggerUI(c =>
c.SwaggerEndpoint("/swagger/v1/swagger.json", "BpAppWebService"));
```

Swagger vytváří dokumentací z XML komentářů nad jednotlivými metodami. Vygenerovaná dokumentace může vypadat následovně.



Obrázek 11 Ukázka dokumentace koncových bodů pomocí Swaggeru

Kromě jednotlivých přístupových bodů a popisu jejich funkčností, také při otevření přístupového bodu dokumentují jeho vstupní parametry a výstupní data.



Obrázek 12 Ukázka dokumentace konkrétního přístupového bodu

Z dokumentace lze vidět, že tento přístupový bod nepožaduje žádný parametr a předává všechny uživatele v aplikaci. Tento přístupový bod slouží pro zobrazení všech zaměstnanců majiteli firmy.

4.1.4 Ukázka zpracování GET požadavku

Na zprostředkování detailu smlouvy je předvedeno přijmutí GET požadavku s parametrem a rozdělení do jednotlivých vrstev. Dále Single Responsibility Principe, Dependency Inversion Principle z principů SOLID, které byly zmíněny v teoretické části této práce. Dále je zde předveden vzor Factory, vzor Repository a mapování pomocí technologie Dapper.

Detail smlouvy obsahuje:

- Celkové náklady, které jsou závislé na nákladovosti (mzdě) zaměstnanců a jejich odpracovaných hodinách na dílčích úkolech.
- Jednotlivé úkoly zaměstnanců, které obsahují informace o odpracované době, přidělené době a stavu v kterém se úkol nachází.
- Všechny zaměstnance, pracující na této smlouvě a jejich profese pro přehlednost.
- Průměrný stav zpracování smlouvy (průměr stavu dílčích zaměstnaneckých úkolů)
- Celkový strávený čas na dané smlouvě.
- Odhadovaný čas potřebný k dokončení smlouvy.

ContractDetailController

```
[Produces("application/json")]
[Route("api/ContractDetail")]
[Authorize(AuthenticationSchemes = OAuthValidationDefaults.Authentication-
Scheme)]
[EnableCors("CorsPolicy")]
public class ContractDetailController : Controller
{
    private readonly IContractDetailProvider _contractDetailProvider;

    public ContractDetailController(
      IContractDetailProvider contractDetailProvider)
    {
        _contractDetailProvider = contractDetailProvider;
    }
    /// <summary>
    /// Provide contract detail
    /// </summary>
    /// <param name="id"> contractID</param>
    /// <returns></returns>
```

```
    [HttpGet("{id}")]
    public ContractDetail Get(int id)
    {
        return _contractDetailProvider.GetContractDetail(id);
    }
}
```

Atributy

- Authorize – definuje, že pro přístup k jednotlivým metodám je potřeba autentifikace.
- Produces – definuje výstup (Javascript Object Notation)
- Route – definuje směrování z URI do příslušného controlleru.
- EnableCors – spolu s nastavením ve Startup umožňuje controlleru zpracovávat požadavky z jiných domén.

Tento controller zprostředkovává detail smlouvy GET požadavkem. Parametrem je ID smlouvy, který se předá metodě GetContractDetail objektu ContractDetailProvider.

```
public class ContractDetailProvider : IContractDetailProvider
{
    private readonly IContractRepository _contractRepository;
    public ContractDetailProvider(IContractRepository contractRepository)
    {
        _contractRepository = contractRepository;
    }

    public ContractDetail GetContractDetail(int contractID)
    {
        var detail = new ContractDetail
        {
            Contract = _contractRepository.GetContractByID(contractID),
            ContractExecutors =
                    _contractRepository.GetContractTasks(contractID),
        };
        if (detail.ContractExecutors == null) return detail;
      detail.TotalTime =
                  detail.ContractExecutors.Sum(x => x?.Task?.SpendTime);
      detail.TotalCosts = CalculateTotalCosts(detail.ContractExecutors);
      detail.AveragePercentState =
                  CalculateAveragePercentState(detail.ContractExecutors);
        return detail;
    }
```

Tato metoda pomocí další vrstvy repository získá data. Pokud smlouva nemá přidělené žádné zaměstnance, kteří na ní pracují, neprovádějí se žádné kalkulace a ihned se vrací získaný detail. Pokud se na smlouvě pracuje, provedou se potřebné kalkulace pomocí dalších privátních metod vlastní třídy.

Tato implementace je demonstrací principů SOLID, konkrétně:

- SRP – třída má jedinou zodpovědnost a jednoznačný název, tudíž jediný důvod ke změně.
- DIP – závislost třídy je vedena k rozhraní (Interface). Tudíž od konkrétní implementace k abstraktnímu rozhraní.

Vzor Repository a Factory

Samotné získání a mapování dat na objekt se provádí ve vrstvě Repository. Pomocí vzoru Factory se získá objekt, zprostředkovávající připojení k databázi. ConnectionString z kterého je vytvořené spojení, je uložen v souboru appsettings.json.

```
public class DbConnectionFactory : IDbConnectionFactory
{
    private readonly IConfiguration _configuration;

    public DbConnectionFactory(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public IDbConnection GetConnection()
    {
        return new SqlConnection(
_configuration.GetSection("ConnectionStrings").GetSection("db").Value);
    }
}
```

Kontrola nad otevíráním spojení je přesunuta zpět do vrstvy Repository. Uzavření otevřeného spojení je zajištěno pomocí statementu using, které po skončení scopu uzavře spojení.

```
public MyContract GetContractByID(int contractID)

{
    using (var con = _dbConnectionFactory.GetConnection())
    {
        return con.Query<MyContract>("dbo.ProcContractByID",
        new { ContractID = contractID },
        commandType: CommandType.StoredProcedure).FirstOrDefault();
    }
}
```

Pro získání informací o smlouvě je napsána procedura v MS-SQL, která jako parametr přebírá ID smlouvy.

Pro získání jednotlivých úkolů spadajících pod smlouvu a jejich vykonavatelů, je napsána také procedura, která pro ukázku práce s Dapperem vrací více tabulek.

Dapper je schopen tyto tabulky postupně namapovat na patřičné objekty pomocí metody QueryMultiple a Read.

Metoda QueryMultiple uloží výsledek procedury do objektu GridReader. Metoda Read poté prochází data a postupně je mapuje na definované objekty.

```
public List<ContractTask> GetContractTasks(int contractID)
{
    var contractTasks = new List<ContractTask>();
    var employees = new List<Employee>();
    var tasks = new List<EmployeeTask>();

    using (var con = _dbConnectionFactory.GetConnection())
    {
        using (var multi =
        con.QueryMultiple("dbo.ProcContractExecutors",
        new { ContractID = contractID },
        commandType: CommandType.StoredProcedure))
        {
            employees = multi.Read<Employee>().ToList();
            tasks = multi.Read<EmployeeTask>().ToList();
        }
    }

    employees.ForEach(x =>
    {
        contractTasks.Add(new ContractTask {
            Employee = x,
            Task = tasks
            .Where(y => y.EmployeeID.Equals(x.Id)).FirstOrDefault() });
    });

    return contractTasks;
}
```

V případě, že procedura vrací data, používá se generická metoda Query, které se definuje na jaký objekt má získaná data zkusit namapovat. Pokud mapování selže, vyhodí výjimku. Pokud procedura provádí operace jako smazaní nebo nastavení příznaku, lze použít metodu Execute pro jednoduché provedení procedury bez výstupu.

4.1.5 Ukázka zpracování PUT a DELETE požadavku

Na smazání uživatele z databáze a upravení stávajícího uživatele, je předvedeno zpracování PUT a DELETE požadavku. Tyto operace se provádí v EmployeeControlleru a jelikož není potřeba provádět další operace nad daty, controller nepřistupuje k vrstvě service, ale přímo k vrstvě repository.

```
[Produces("application/json")]
[Route("api/Employee")]
[Authorize(AuthenticationSchemes = OAuthValidationDefaults.Authentication-
Scheme)]
[EnableCors("CorsPolicy")]
public class EmployeeController : Controller
{
    private IEmployeeRepository _employeeRepository;

    public EmployeeController(IEmployeeRepository employeeRepository)
    {
        _employeeRepository = employeeRepository;
    }

    /// <summary>
    /// Update employee
    /// </summary>
    /// <param name="employee">Employee for update</param>
    [HttpPut]
    public void Put([FromBody]Employee employee)
    {
        _employeeRepository.UpdateEmployee(employee);
    }

    /// <summary>
    /// delete employee
    /// </summary>
    /// <param name="id">employeeID</param>
    [HttpDelete("{id}")]
    public void Delete(string id)
    {
        _employeeRepository.DeleteEmployee(id);
    }
```

Metoda PUT se stará o aktualizaci stávajícího uživatele, majitel firmy je díky tomu schopen nastavit zaměstnanci mzdu nebo profesní zaměření. Z těla příchozího požadavku se vytvoří objekt typu Employee, který je poté předán repositáři, který provede samotný update.

Metoda DELETE zajišťuje odebrání uživatele z databáze, přijímá parametr ID, které se opět předá metodě DeleteEmployee, provádějící samotné smazání.

4.2 Web

V této části je rozebrána samotná implementace webu, pomocí technologie .Net Core 2.0 MVC. Jsou zde vybrány a popsány základní stavební kameny webu. Registrace, přihlášení a rozdělení do oblastí.

4.2.1 Seskupení do oblastí

Framework .Net umožňuje vytvářet oblasti (Areas). Na webu se nachází dvě oblasti pro autentifikované uživatele

- Admin
- Employee

Pro přidání oblasti stačí vytvořit novou oblast a poté nastavit směrování v metodě Configure třídy Startup. Následujícím způsobem se vytvoří směrování pro jednotlivé oblasti:

```
app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");

            routes.MapRoute(
                name: "Employee",
     template: "{area:exists}/{controller=Employee}/{action=Index}/{id?}");

            routes.MapRoute(
                name: "Admin",
     template: "{area:exists}/{controller=Admin}/{action=Index}/{id?}");
        });
```

Každá oblast poté obsahuje vlastní sady specifických controllerů, view a viewmodelů.

4.2.2 Ukázka oblasti Admin

Na následující ukázce je předvedeno zobrazení detailu smlouvy pro majitele firmy. Data pro tento detail poskytuje webová služba. Postup získání dat je popsán výše.

Majitel firmy se nachází v oblasti Admin. Po kliknutí na jméno smlouvy je přesměrován do metody ContractDetail ContractControlleru v této oblasti. Metoda přijímá parametr contractID.

```
[Area("Admin")]
[Route("Admin/[controller]")]
[Authorize(Roles = "Admin")]
public class ContractController : Controller
{
    private readonly IAdminService _adminService;

    public ContractController(IAdminService adminService)
    {
        _adminService = adminService;
    }

    public IActionResult ContractDetail(int contractID)
    {
        var detail = _adminService.GetContractDetail(contractID);
        var vm = new ContractDetailViewModel
        {
            Contract = detail.Contract,
            ContractExecutors = detail.ContractExecutors,
            TotalCosts = detail.TotalCosts,
            TotalTime = detail.TotalTime,
            AveragePercentState = detail.AveragePercentState
        };
        return View(vm);
    }
```

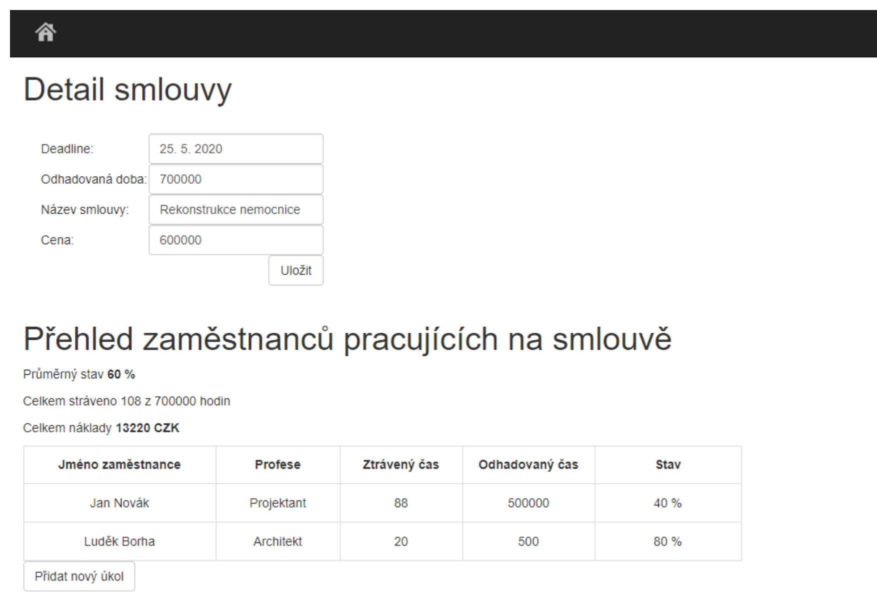
Lze vidět, že pro přístup do tohoto controlleru musí být uživatel v roli Admina, to je docíleno Authorize atributem s požadovanou rolí nad celým controllerem. Nastavení této role probíhá po autorizaci. Role a ID uživatele je uloženo jako cookies na serveru. Platnost této cookies je jedna hodina, poté dojde k odhlášení uživatele.

Detail smlouvy se získá z metody GetContractDetail objektu AdminService, na němž je instance tohoto Controlleru závislá. Vložení závislosti je zajištěno pomocí DI, stejně jako na straně webové služby. Tato metoda poté volá metodu třídy ServiceDataProvider, která provádí samotné vytvoření požadavku.

```
public ContractDetail GetContractDetail(int contractID)
{
    using (var client = new HttpClient())
    {
```

```
        client.DefaultRequestHeaders.Add(
            "Authorization",
            "Bearer " + _accesToken);


        var result = client.GetAsync(
            _baseURI + $"ContractDetail/{contractID}").Result;

        if (!result.StatusCode.Equals(HttpStatusCode.OK))
        {
            return new ContractDetail();
        }
        return JsonConvert.DeserializeObject<ContractDetail>
                                  (result.Content.ReadAsStringAsync().Result);
    }
}
```

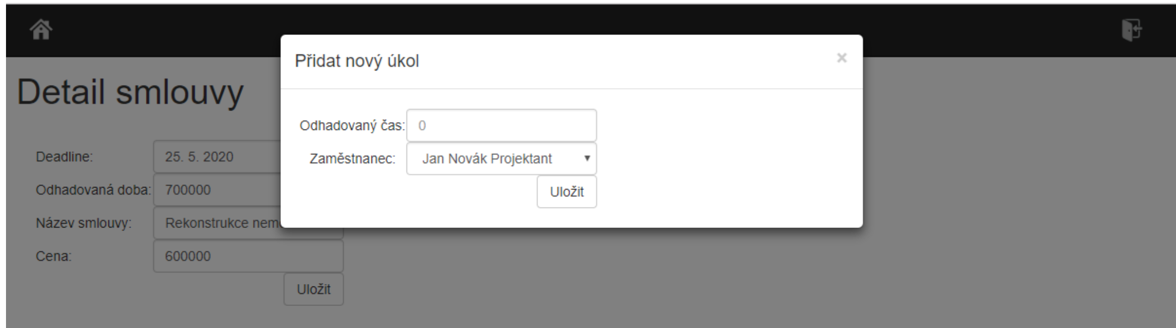
Web (klient) se tedy dotáže serveru na definovaný přístupový bod s patřičným parametrem. Výsledek požadavku je ve formátu JSON čili typu string. Tento string je poté deserializovány pomocí frameworku JSON.NET. Metoda DeserializeObject je opět generická, definuje se typ, na který se mají data deserializovat. Pokud tento typ odpovídá příchozím datům vše proběhne v pořádku. Pokud ne, vyhodí výjimku. Proto se tato deserializaci provádí pouze pokud byl požadavek úspěšně vyřízen (OK).



Obrázek 13 Náhled detailu smlouvy

V detailu smlouvy, kromě přehledu nad jednotlivými úkoly, může majitel firmy změnit Deadline, odhadovanou dobu, název smlouvy a fakturovanou cenu. Nebo přidat další úkol

k této smlouvě. Přidání dalšího úkolu se provede kliknutím na patřičné tlačítko, zobrazí se následující modální okno, kde proběhne výběr požadovaného zaměstnance a přidělí se mu dobu na daném úkolu.



Obrázek 14 Náhled na přidání nového úkolu pod smlouvou

V obou případech se data z formuláře metodou POST pošlou do patřičné metody Contract-Controlleru v oblasti Admin. Poté se objekt pošle ke zpracování webové službě. Názorně ukážu postup přidělení nového úkolu.

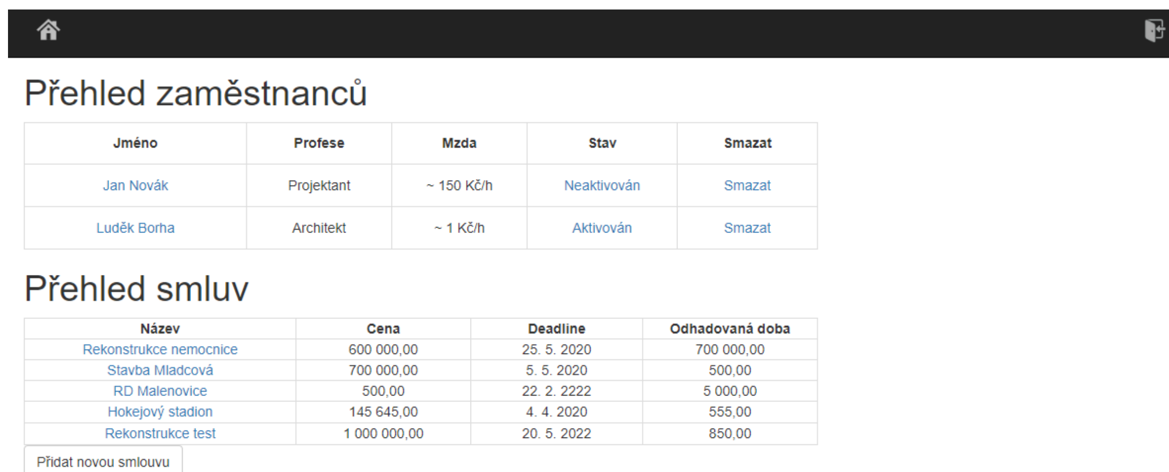
Metoda controlleru přijme uživatelem zadané hodnoty namapovány na objekt Insert-TaskViewModel, který v sobě nese uživatelem naplněný objekt EmployeeTask.

```
[HttpPost]
[Route("AddEmployeeTask")]
public IActionResult AddEmployeeTask(InsertTaskViewModel data)
{
    _adminService.InsertEmployeeTask(data.NewEmployeeTask);
    return RedirectToAction("Index", "Admin");
}
```

Ten se poté pošle ke zpracování webové službě.

```
public void InsertEmployeeTask(EmployeeTask task)
{
    using (var client = new HttpClient())
    {
        client.DefaultRequestHeaders.Add(
            "Authorization", "Bearer " + _accesToken);

        var data = JsonConvert.SerializeObject(task);
        client.PutAsync(_baseURI + "EmployeeTask",
            new StringContent(data, Encoding.UTF8, "application/json"));
    }
}
```

Po úspěšném vložení úkolu proběhne přesměrování na domovskou stránku admina.

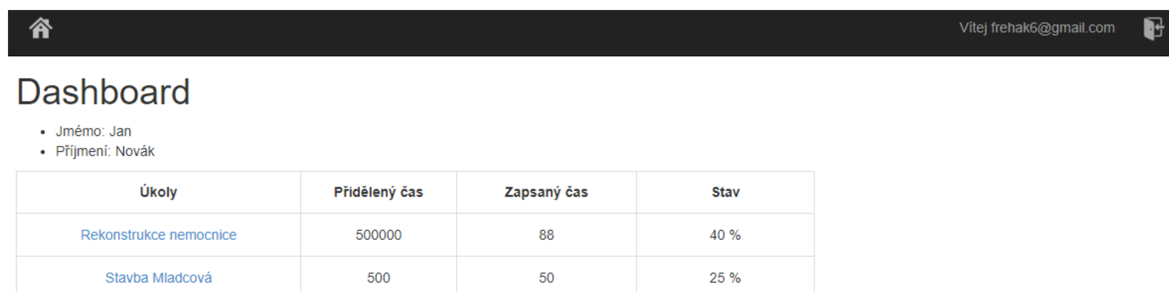


Obrázek 15 Náhled na domovskou stránku Admina

Admin zde může takto procházet jednotlivé smlouvy a editovat jednotlivé zaměstnance. Schvalovat jim přístup do aplikace, nebo je mazat z databáze. Komunikaci s databází poskytuje tomuto webu výše popsaná webová služba.

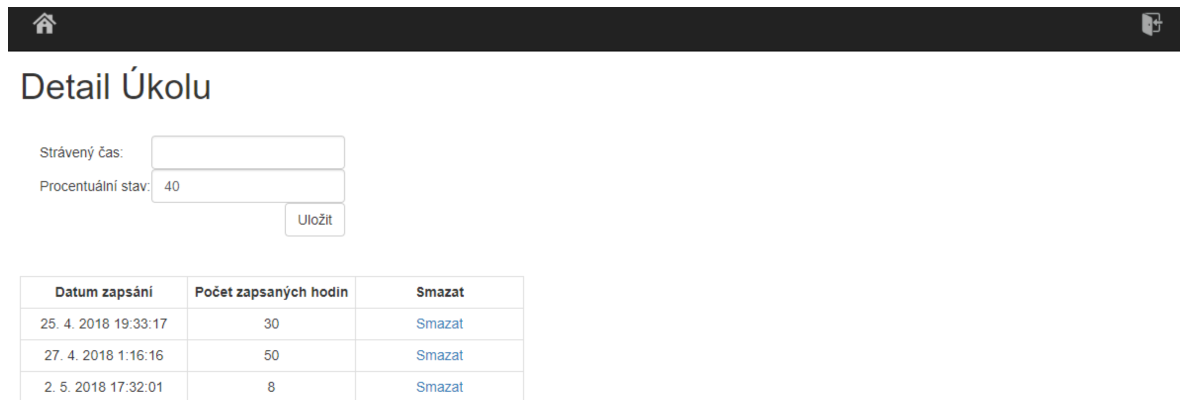
4.2.3 Ukázka oblasti Employee

Zaměstnanec po autentifikaci je přesměrován do své oblasti (Employee Area). Zobrazí se mu následující stránka, kde vidí přidělené úkoly na jednotlivých projektech (smlouvách). Jediné, co zde může dělat, je zapisování a odebírání času z jednotlivých úkolů.



Obrázek 16 Náhled na domovskou stránku zaměstnance

Po kliknutí na jméno úkolu se zaměstnanci zobrazí následující stránka, sloužící pro editaci úkolu.



Obrázek 17 Náhled na detailu úkolu

Jednoduchým formulářem se opět posbírají data a metodou POST pošlou do metody AddTime TaskControlleru v oblasti Employee.

```
[HttpPost]
public IActionResult AddTime(DetailViewModel data)
{
    if (ModelState.IsValid)
    {
        _employeeService.AddSpendTime(data.Task);
        return RedirectToAction("Index", "Employee");
    }
    else
    {
        return View("TaskDetail", data);
    }
}
```

V případě, že jsou přijatá data validní, se přes EmployeeService zavolá metoda AddSpendTime ve třídě ServiceDataProvider s parametrem EmployeeTask, který nese zaměstnancem vyplněné hodnoty. Metodou POST se pošlou data webové službě, která je zpracuje a zapíše do databáze. V případě, že vše proběhne v pořádku, je zaměstnanec přesměrován zpět na svou domovskou stránku, kde vidí přehled úkolů. Zaměstnanec v případě zadání špatného počtu hodin, či procentuálního stavu úkolu, může svůj čas z úkolu odstranit.

4.2.4 Ukázka registrace

Pro registraci do aplikace stačí uživateli vyplnit následující formulář. Uživatelem vyplněná data se pošlou metodou POST do metody Register v AccountControlleru na webu, kde se zkontroluje stav vyplnění formuláře. Pokud je formulář v pořádku, pošle se k validaci webové službě. V případě úspěšné validace (unikátnost emailu), je uživatel vložen do databáze.

Registrace

Vytvořit účet

Email

Heslo

Potvrdit heslo

Jméno

Příjmení

Registrovat

Obrázek 18 Náhled registračního formuláře na webu

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(
    RegisterViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var result = _serviceDataProvider.CreateUser(model);
        if (result)
        {
            return await RedirectToLocal(returnUrl,model.Email,
                                                   model.Password);
        }
    }
    return View(model);
}
```

V případě úspěšného vytvoření se uživatel přesměruje na domovskou stránku v oblasti Employee. Pokud ovšem webová služba odmítne uživatele vložit, proběhne přesměrování zpět na registrační formulář.

Webová služba je nastavená na validaci identity následovně, v případě potřeby lze kdykoliv nakonfigurovat potřebné vlastnosti. Konfigurace opět probíhá ve třídě Startup.

```
services.Configure<IdentityOptions>(options =>
{
    options.Password.RequireDigit = false;
    options.Password.RequiredLength = 8;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = false;
    options.Password.RequireLowercase = false;
    options.Password.RequiredUniqueChars = 6;

    options.SignIn.RequireConfirmedEmail = false;
    options.SignIn.RequireConfirmedPhoneNumber = false;
    options.Lockout.MaxFailedAccessAttempts = 10;
    options.Lockout.AllowedForNewUsers = true;

    options.User.RequireUniqueEmail = true;
});
```

4.3 Mobilní Aplikace

V této části je rozebrána implementace multiplatformní mobilní aplikace. Jedná se o ukázkovou aplikaci, která slouží pro přehled nad výkonem zaměstnanců nadřízenému. Jsou zde vybrány základní části, které umožňují přihlášení do aplikace a proklik na detail smlouvy, sloužící pro základní přehled o procentuálním stavu, ve kterém se smlouva nachází. Dále zde nadřízený vidí náklady a odpracované hodiny na dané smlouvě. Funkcionality jsou demonstrovány na dvou nejrozšířenějších platformách Android a iOS. Jelikož není potřeba žádných specifik platformy (například přístup k úložišti), je možné napsat jeden kód, který se poté přeloží do nativního jazyka dané platformy. Tedy je docíleno multiplatformního řešení bez nutnosti zásahu do kódu jednotlivých platforem. V případě potřeby práce s úložištěm se musí implementovat třída, obsluhující tuto činnost pro každou platformu zvlášť.

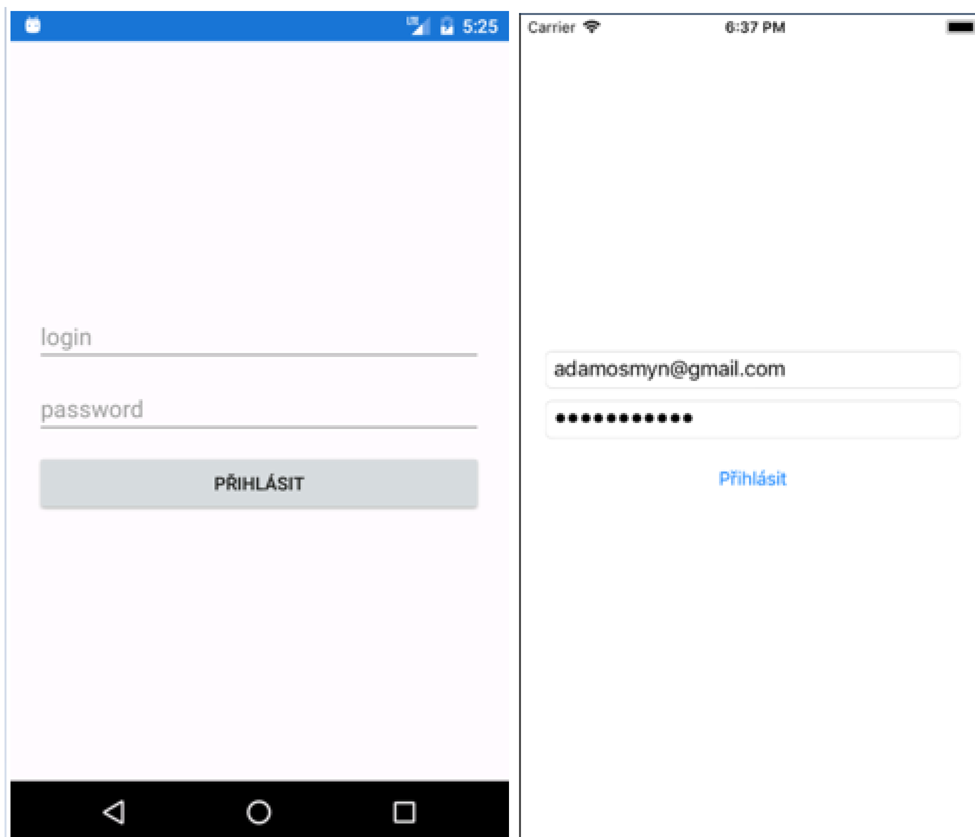
4.3.1 Přihlášení

Po otevření aplikace se zobrazí přihlašovací stránka. V Xamarin.Forms, lze UI psát jak v XAMLu tak v C#. V této ukázkové aplikaci předvedu obě možnosti. Přihlašovací stránka je napsána v C# a její implementace vypadá následovně:

```
public class LoginPage : ContentPage
{
    public LoginPage(LoginViewModel context)
    {
        this.BindingContext = context;

        var loginNameEntry = new Entry()
        {
            Placeholder = "login",
            HorizontalOptions = LayoutOptions.FillAndExpand
        };
        var passwordNameEntry = new Entry()
        {
            Placeholder = "password",
            IsPassword = true,
            HorizontalOptions = LayoutOptions.FillAndExpand
        };
        var button = new Button() { Text = "Přihlásit" };

        loginNameEntry.SetBinding(
            Entry.TextProperty,
            new Binding("Login")
        );
        passwordNameEntry.SetBinding(
            Entry.TextProperty,
            new Binding("Password")
        );
        button.SetBinding(
            Button.CommandProperty,
            new Binding("LoginCommand")
        );

        Content = new StackLayout
        {
            Children = {
                loginNameEntry,
                passwordNameEntry,
                button
            },
            Orientation = StackOrientation.Vertical,
            HorizontalOptions = LayoutOptions.FillAndExpand,
            VerticalOptions = LayoutOptions.CenterAndExpand,
```

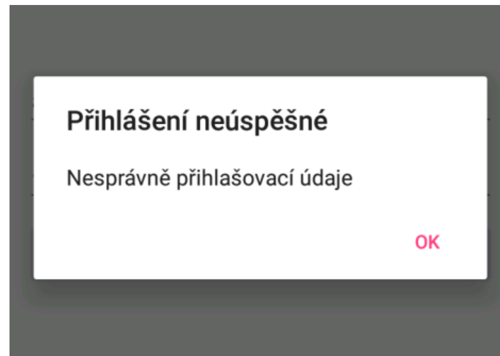
```
                Spacing = 10,
                Padding = new Thickness(20, 0)
            };
        }
    }
```

Lze v kódu vidět, jak probíhá nastavení binding contextu a bindování jednotlivých vstupů na ViewModel. V kódu je také předvedeno nastavení jednotlivých placeholderů a obsahu celé stránky. Vstup pro heslo lze jednoduše ošetřit, aby schovával zadané znaky pomocí příznaku IsPassword.



Obrázek 19 Přihlašovací stránka Android a iOS

Na této stránce uživatel vyplní email a heslo pod kterým je veden v databázi. V případě, že validace na straně webové služby proběhne v pořádku, uloží se přístupový token do vlastnosti třídy App.xaml, která je přístupná po celou dobu běhu aplikace. V opačném případě se zobrazí následující chybová hláška.



Obrázek 20 Chybová hláška Android

Zpracování požadavku na přihlášení, obsluhuje následující metoda, která je díky bindování spuštěna po kliknutí na tlačítko přihlásit.

```
void LoginCommandExecute()
{
    if (_provider.GetBearerToken(Login, Password))
    {
        App.Current.MainPage =
      new NavigationPage(ServiceLocator.Current.GetInstance<MainPage>());
    }
    else
    {
        Page.DisplayAlert("Přihlášení neúspěšné",
                          "Nesprávné přihlašovací údaje",
                          "Ok");
    }
}
```

Získání přístupového tokenu probíhá naprosto stejně jako na webu, zprostředkovává ho metoda GetBearerToken s parametry Login, Password. Metoda vypadá následovně:

```
public bool GetBearerToken(string username, string password)
{
    var content = new List<KeyValuePair<string, string>> {
        new KeyValuePair<string, string>("username", username),
        new KeyValuePair<string, string>("password", password),
        new KeyValuePair<string, string>("grant_type", "password")
    };
    using (var client = new HttpClient())
    {
        var response = client.PostAsync(_authURI,
                                new FormUrlEncodedContent(content)).Result;

        if (!response.IsSuccessStatusCode) return false;

      var deserializedResponse =
```

```
        JsonConvert.DeserializeObject<AuthorizationResponse>
                        (response.Content.ReadAsStringAsync().Result);

        ((App)Xamarin.Forms.Application.Current).accesToken =
                                        deserializedResponse.access_token;
    }
    return true;
}
```

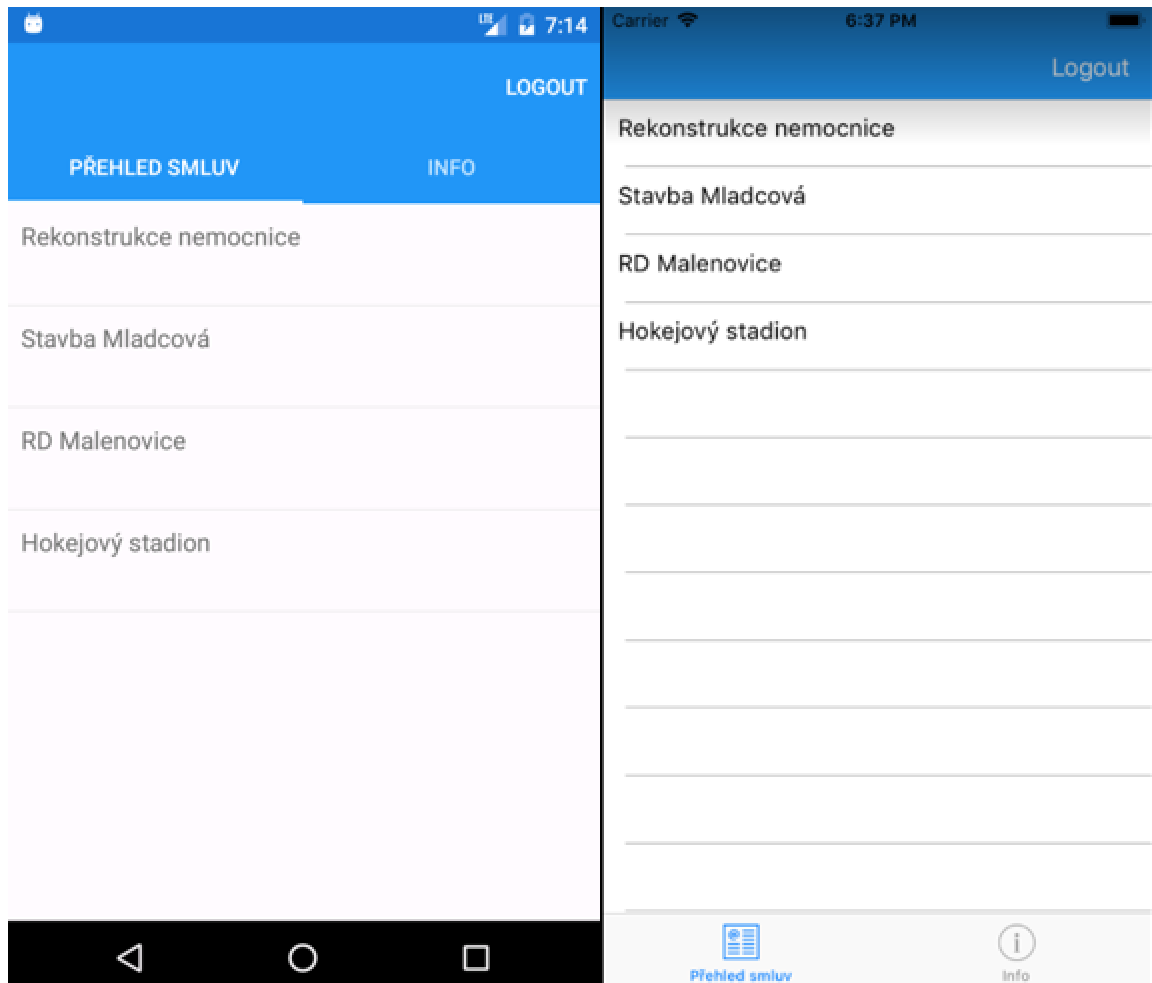
Získání instance třídy ServiceDataProvider, ve ViewModelu je provedeno pomocí Constructor Injection. Xamarin.Forms ovšem nemá vlastní implementaci Dependency Injection jako je tomu u .Net Core. Je proto použit framework Unity, který lze jednoduše stáhnout pomocí Nugget Package Manageru. Nastavení probíhá ve třídě App.xaml následovně:

```
public App()
{
    InitializeComponent();
    UnityContainer unityContainer = new UnityContainer();
    RegisterAppServices(unityContainer);
    ServiceLocator.SetLocatorProvider(() =>
                        new UnityServiceLocator(unityContainer));
    MainPage = ServiceLocator.Current.GetInstance<LoginPage>();
}

private void RegisterAppServices(UnityContainer container)
{
    container.RegisterType<IServiceDataProvider, ServiceDataProvider>();
}
```

Proběhne zde zaregistrování vlastních typů a nastavení objektu ServiceLocator. Poté lze instanci získat jak pomocí Constructor Injection, tak ručně pomocí nastaveného objektu ServiceLocator.

4.3.2 Přehled smluv a detail smlouvy

Po přihlášení do aplikace se zobrazí stránka, která v sobě nese všechny smlouvy uložené v systému. V hlavičce stránky je také tlačítko pro odhlášení. Pomocí kterého se uživatel okamžitě odhlásí z aplikace a jeho přístupový token se vymaže.



Obrázek 21 Přehled smluv Android a iOS

Tato stránka je vytvořena pomocí XAMLu pro ilustraci rozdílu oproti C#. Kód vypadá následovně:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BPAppMobileApp.Views.ItemsPage"
             Title="{Binding Title}"
             x:Name="BrowseItemsPage"
             NavigationPage.HasNavigationBar="False"
             >
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Logout" Command="{Binding LogoutCommand}"/>
    </ContentPage.ToolbarItems>
    <ContentPage.Content>
        <StackLayout>
            <ListView x:Name="ItemsListView"
                 ItemsSource="{Binding Items}"
                 VerticalOptions="FillAndExpand"
```

```
                 HasUnevenRows="true"
                 RefreshCommand="{Binding LoadItemsCommand}"
                 IsPullToRefreshEnabled="true"
                 IsRefreshing="{Binding IsBusy, Mode=OneWay}"
                 CachingStrategy="RecycleElement"
                 SelectedItem="{Binding SelectedItem}"
                 ItemSelected="OnItemSelected">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <StackLayout Padding="10">
                                <Label Text="{Binding Name}"
                       LineBreakMode="NoWrap"
                       Style="{DynamicResource ListItemTextStyle}"
                       FontSize="16" />
                                <Label Text="{Binding Description}"
                       LineBreakMode="NoWrap"
                       Style="{DynamicResource ListItemDetailTextStyle}"
                       FontSize="13" />
                            </StackLayout>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Na první pohled vypadá tento kód zcela odlišně, avšak funguje úplně stejně jako kód napsaný v C#. Pomocí tagů se vytvářejí instance daných objektů a definuje jejich nastavení. Například vytvoření instance třídy ListView a nastavení bindingu vypadá následovně:

```
 <ListView x:Name="ItemsListView"
      ItemsSource="{Binding Items}"
```

Tímto se definuje objektu ListView, kterou vlastnost ViewModelu má brát jako zdroj pro zobrazení. Načtení dat probíhá ve ViewModelu, volání webové služby s přístupovým tokenem už bylo demonstrováno, je zde tedy předvedena pouze práce samotného ViewModelu.

```
async Task ExecuteLoadItemsCommand()
{
    if (IsBusy)
        return;
    IsBusy = true;

    try
```

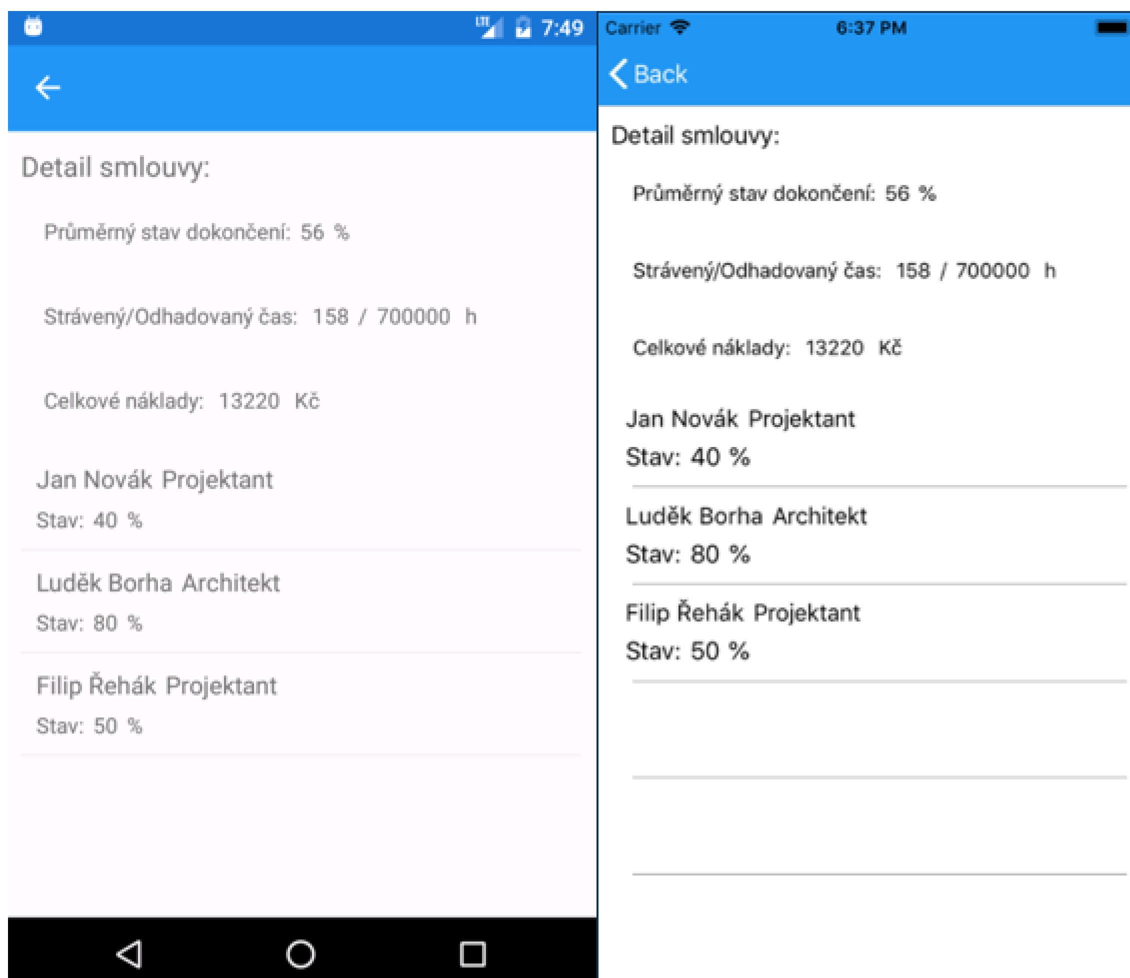
```
    {
        Items.Clear();
        var items = _provider.GetContracts();
        foreach (var item in items)
        {
            Items.Add(item);
        }
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
    }
    finally
    {
        IsBusy = false;
    }
}
```

Tato metoda se spustí při každém načtení, či obnovení stránky. Dotáže se webové služby a získaná data přidá do vlastnosti Items. Spouštění metody je opět nastaveno pomocí bindingu na ViewModel. Nastavení bindingu je provedeno následovně:

```
RefreshCommand="{Binding LoadItemsCommand}"
```

Postup zobrazení detailu probíhá obdobně. V aplikaci je vytvořena stránka pro zobrazení detailu smlouvy. Po kliknutí na danou smlouvu se vytvoří instance stránky a daného ViewModelu. Poté se spustí metoda GetDetailData s parametrem vybrané smlouvy. Logika získání dat se odehrává v tomto ViewModelu. Samotná data se poté získávají opět pomocí webové služby. Postup získání detailu smlouvy už v této práci byla popsán, proto je v této části rozebrána pouze implementace ItemDetailViewModelu.

```
public class ItemDetailViewModel : BaseViewModel
{
    public ContractDetail Item { get; set; }
    private readonly IServiceDataProvider _provider;
    public ItemDetailViewModel(IServiceDataProvider provider)
    {
        _provider = provider;
    }

    public void GetDetailData(int id)
    {
        Item = _provider.GetContractDetail(id);
    }
}
```



Obrázek 22 Detail smlouvy Android a iOS

Po získání dat z webové služby se nadřízenému zobrazí následující stránka, v níž má stručný přehled nad danou smlouvou.

ZÁVĚR

Návrhové principy, návrhové vzory, softwarová architektura a testování, jsou jedny ze stěžejních částí kvalitního softwaru. V této práci byl za využití zmíněných technik vytvořen ukázkový systém s architekturou klient-server. Tento systém obsahuje dvě klientské strany, které pro svůj chod potřebují server, jež je pro ně zprostředkovatelem dat.

Hlavní klientskou stranou je web, který je realizován technologií .Net Core 2.0 s architekturou MVC. Uživatel se zde může registrovat a zapisovat či mazat odpracované hodiny na úkolu. V případě nadřízeného kontrolovat stavy projektů, nákladovosti a spravovat přístup zaměstnanců do aplikace. Případně zaměstnance editovat, či smazat z databáze.

Ve vytvořené mobilní aplikaci, realizované pomocí technologie Xamarin.Forms a architektury MVVM, která je určena pro nadřízeného, může uživatel kontrolovat stavy projektů. Například v době, kdy nemá pohodlný přístup k webovému prohlížeči. Vzhledem k multiplatformnímu řešení, není nadřízený nucen přistupovat k aplikaci pouze z jedné platformy. Například pokud je nadřízených více a každý vlastní jinou mobilní platformu.

Vytvořený server je realizován pomocí frameworku .Net Core 2.0. Zprostředkovává jak přístup k datům, tak autentifikaci uživatele, jež se snaží vstoupit do aplikace. Autentifikace je vyřešena pomocí přístupových tokenů. Toto API disponuje dokumentací, která je vytvořena pomocí frameworku Swagger a dokumentuje jednotlivé koncové body.

Cílem praktické části bylo, po provedené analýze a návrhu implementace, vytvořit ukázkový systém pro malou stavební firmu, s využitím vhodných technologií. Na tomto systému poté demonstrovat doporučené postupy a klíčové prvky řešení. Tento systém byl úspěšně vytvořen a disponuje logikou, umožňující základní funkcionality. Díky doporučeným postupům jsou jednotlivé komponenty tohoto systému od sebe izolovány a můžou být kdykoliv, v případě potřeby, jednoduše rozšířeny o další funkcionality. Kdykoliv lze například v mobilní aplikaci přidat část pro zaměstnance a implementovat pro ně funkcionalitu, která jim umožní zapisování času z mobilního telefonu. Jelikož se jedná o architekturu klient-server, stačí uživatelem zapsaná data pouze odeslat na koncový bod API, který se postará o vložení dat do databáze.

Shrnout doporučené postupy, návrhové vzory a popsat vybrané nejnovější technologie bylo cílem teoretické části. Tato část byla úspěšně vytvořena a slouží, mimo jiné, jako úvod do problematiky k lepšímu pochopení praktické části.

V rámci této bakalářské práce vzniklo shrnutí doporučených postupů při vývoji softwaru a popis moderních technologií pro tvorbu klient-server aplikací. Následně použitím těchto technologií a postupů, vznikl ukázkový systém pro malou stavební firmu, jehož klíčové prvky jsou demonstrovány v praktické části této práce.

SEZNAM POUŽITÉ LITERATURY

1. msatechnosoft. *TECH BLOG.* [Online] 3. Duben 2017. [Citace: 20. Březen 2018.] https://msatechnosoft.in/blog/tech-blogs/types-of-client-server-architecture/.

2. SINGLETON, James. *ASP.NET Core 1.0 High performance.* Birmingham : Packt Publishing, 2016. 9781785881893.

3. NAGEL, Christian. *Professional C# 6 and .Net Core 1.0.* Indianapolis : John Wiley, 2016. 9781119096603.

4. .NET Documentation. *Microsoft Docs.* [Online] Microsoft, 3. Březen 2017. [Citace: 20. Březen 2018.] https://docs.microsoft.com/cs-cz/dotnet/framework/get-started/net-core-and-open-source.

5. API Design guidance. *Microsoft Docs.* [Online] Microsfot, 12. 1 2018. [Citace: 23. Březen 2018.] https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design.

6. HTTP/1.1 Status Code Definitions. *W3.* [Online] World Wide Web Consortium. [Citace: 25. Březen 2018.] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

7. Entity Framework Overview. *Microsoft Docs.* [Online] Microsoft, 30. Březen 2017. [Citace: 25. Březen 2018.] https://docs.microsoft.com/en-US/dotnet/framework/data/adonet/ef/overview.

8. MARTIN, Robert C. Design Principles and Design Patterns. [Online] 2000. [Citace: 28. Březen 2018.] https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

9. Lieberherr, Karl. Law of Demeter: Principle of Least Knowledge. *College of Computer and Information Science.* [Online] Northeastern University. [Citace: 28. Březen 2018.] http://www.ccs.neu.edu/home/lieber/LoD.html.

10. Don't Repeat Yourself. *Programmer 97-things.* [Online] O'Reilly commons. [Citace: 2. Duben 2018.] http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Repeat_Yourself.

11. Developer Network. *ASP.NET MVC Overview.* [Online] Microsoft. [Citace: 15. Duben 2018.] https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx.

12. Fowler, Martin. Supervising Controller. *Martin Fowler.* [Online] 2006. Červen 2006. [Citace: 25. Duben 2018.] https://martinfowler.com/eaaDev/SupervisingPresenter.html.

13. —. Passive View. *Martin Fowler.* [Online] 18. Červenec 2006. [Citace: 2018. Duben 28.] https://martinfowler.com/eaaDev/PassiveScreen.html.

14. The MVVM Pattern. *Developer Network.* [Online] Microsoft. [Citace: 28. Duben 2018.] https://msdn.microsoft.com/en-us/library/hh848246.aspx.

15. McFarlin, Tom. The Beginner's Guide to Unit Testing: What Is Unit Testing? *EnvatoTuts+.* [Online] 19. Červen 2012. [Citace: 29. Duben 2018.] https://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728.

16. Unit Test Basics. *Developer Network.* [Online] Microsoft. [Citace: 29. Duben 2018.] https://msdn.microsoft.com/en-us/library/hh694602.aspx?f=255&MSPPError=-2147217396.

17. Hlava, Tomáš. Integrační testování. *Testování Softwaru.* [Online] 21. Srpen 2011. [Citace: 30. Duben 2018.] http://testovanisoftwaru.cz/tag/integration-testing/.

18. Palermo, Jeffrey. Guidelines for Test-Driven Development. *Developer Network.* [Online] Microsoft, Březen 2006. [Citace: 29. Duben 2018.] https://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx.

19. The Repository Pattern. *Developer Network.* [Online] Microsoft. [Citace: 30. Duben 2018.] https://msdn.microsoft.com/en-us/library/ff649690.aspx.

20. Purdy, Doug. Exploring the Factory Design Pattern. *Developer Network.* [Online] Microsoft, Únor 2002. [Citace: 30. Duben 2018.] https://msdn.microsoft.com/en-us/library/ee817667.aspx.

21. Dependency injection in ASP.NET Core. *Developer Network.* [Online] Microsoft, 14. Říjen 2016. [Citace: 30. Duben 2018.] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.0.

22. Dependency Injection. *Developer Network.* [Online] Microsoft. [Citace: 30. Duben 2018.] https://msdn.microsoft.com/en-us/library/dn178469(v=pandp.30).aspx.

23. Xamarin.Forms. *Microsoft Docs.* [Online] Microsoft. [Citace: 2. Květen 2018.] https://developer.xamarin.com/guides/xamarin-forms/.

24. OpenID Connect FAQ and Q&As. *OpenID.* [Online] OpenID. [Citace: 15. Květen 2018.] http://openid.net/connect/faq/.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API Programmable Application Interface – Programovatelné aplikační rozhraní

CRUD Create, Read, Update, Delete operace

DI Dependency Injection – vkládání závislostí

IoC Inversion of Control

OAuth Protokol zajišťující bezpečnou autorizaci či autentizaci

REST Representational State Transfer

SOAP Simple Object Access Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

XAML Extensible Application Markup Language – značkovací jazyk určený k popisu UI

XML eXtensible Markup Language – obecný značkovací jazyk

SEZNAM OBRÁZKŮ

Obrázek 1 Princip architektury Klient-Server [1] 10
Obrázek 2 Schéma znázorňující architekturu MVC 17
Obrázek 3 Schéma návrhového vzoru MVP 18
Obrázek 4 Schéma návrhového vzoru MVP Passive View 18
Obrázek 5 Schéma návrhového vzoru MVVM 19
Obrázek 6 Vzor Repository [19] 21
Obrázek 7 Vzor Factory [20] 21
Obrázek 8 Náhled na editaci zaměstnance 28
Obrázek 9 Databázový diagram 31
Obrázek 10 Autorizační proces 32
Obrázek 11 Ukázka dokumentace koncových bodů pomocí Swaggeru 37
Obrázek 12 Ukázka dokumentace konkrétního přístupového bodu 37
Obrázek 13 Náhled detailu smlouvy 45
Obrázek 14 Náhled na přidání nového úkolu pod smlouvou 46
Obrázek 15 Náhled na domovskou stránku Admina 47
Obrázek 16 Náhled na domovskou stránku zaměstnance 47
Obrázek 17 Náhled na detailu úkolu 48
Obrázek 18 Náhled registračního formuláře na webu 49
Obrázek 19 Přihlašovací stránka Android a iOS 52
Obrázek 20 Chybová hláška Android 53
Obrázek 21 Přehled smluv Android a iOS 55
Obrázek 22 Detail smlouvy Android a iOS 58

SEZNAM PŘÍLOH

P I SEZNAM PŘÍLOH NA CD

PŘÍLOHA P I: SEZNAM PŘÍLOH NA CD

- SRC – adresář obsahující zdrojové kódy ukázkové aplikace