Výuková aplikace demonstrující nejnovější postupy při tvorbě klient-server aplikací

Viktor Schwarz

Bakalářská práce
2017

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

akademický rok: 2016/2017

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Viktor Schwarz**
Osobní číslo:	**A14139**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Forma studia:	**prezenční**
Téma práce:	**Výuková aplikace demonstrující nejnovější doporučené postupy při tvorbě klient-server aplikací**
Téma anglicky:	**A Tuitorial Tool for Demonstrating the Latest Recommended Approaches to the Creation of Client-Server Applications**

Zásady pro vypracování:

1. **Popište technologie, doporučené postupy, návrhové vzory a techniky pro tvorbu klient-server aplikací.**
2. **Navrhněte vhodnou architekturu a strukturu příkladů pro výuku s využitím aplikační platformy .NET Core a pomocí technologie Xamarin.**
3. **Vytvořte příklady a vzorová řešení pro výuku moderních postupů tvorby aplikací.**
4. **Popište klíčové části navržených příkladů a vzorových řešení.**
5. **Demonstrujte výsledky.**

Rozsah bakalářské práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **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-mobileapps-with-xamarin-forms/.**
2. **FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003. Addison-Wesley signature series. ISBN 0-321-12742-0.**
3. **NAGEL, Christian. Professional C# 6 and .Net Core 1.0. Indianapolis, IN: Wrox, a Wiley brand, published by John Wiley & Sons, 2016. ISBN 111909660X.**
4. **J. PRICE, Mark. C# 6 and .NET Core 1.0: Modern Cross-Platform Development. Birmingham: Packt Publishing, 2016. ISBN 9781785285691.**
5. **SINGLETON, James. ASP.NET Core 1.0 High performance. Birmingham: Packt Publishing, 2016. ISBN 9781785881893.**
6. **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/.**
7. **.NET - Powerful Open Source Cross Platform Development [online]. 2017 [cit. 2017-01-22]. Dostupné z: https://www.microsoft.com/net**

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

Datum zadání bakalářské práce: **20. července 2017**

Termín odevzdání bakalářské práce: **29. srpna 2017**

Ve Zlíně dne 20. července 2017

L.S.

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

Jméno, příjmení: Viktor Schwarz

Název bakalářské/diplomové práce: Výuková aplikace demonstrující nejnovější postupy při tvorbě klient-server aplikací

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/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 diplomová/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 diplomovou/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 diplomové/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 diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/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 diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

.....
podpis diplomanta

ABSTRAKT

Cílem této bakalářské práce je vytvoření vzorových aplikací demonstrujících nejnovější doporučené postupy při tvorbě klient-server aplikací a multiplatformních mobilních aplikací. Výstupem práce je zadání příkladů a vzorových řešení pro výuku moderních postupů tvorby aplikací. V teoretické části budou popsány vybrané technologie, doporučené postupy, návrhové vzory a techniky pro tvorbu klient-server aplikací. Praktická část je potom zaměřena na návrh doporučených úkolů pro studenty a jejich následnou realizaci implementující doporučené postupy, které jsou popsány v teoretické části.

Klíčová slova: klient-server aplikace, .NET Core, Web API, Windows Forms, multiplatformní vývoj, Xamarin.Forms, architektonické vzory, dependency injection

ABSTRACT

The aim of this bachelor's thesis is to come up with sample applications showing modern best practices for client-server applications and multiplatform mobile applications. The product of this work is proposal of task assignments for teaching modern best practices of application development. Theoretical section will deal with describing technologies, recommended approach, architecture patterns and techniques for client-server application development. The practical section of this thesis is focused on designing recommended tasks for students and their realization implementing best practices described in theoretical part.

Keywords: client-server application, .NET Core Web API, application Windows Forms, console application, multiplatformní application Xamarin.Forms, architecture patterns, dependency injection

Tímto bych chtěl bych poděkovat panu Ing. et Ing. Eriku Králi, Ph.D. za odborné rady, konzultace a čas, který mi věnoval při vedení této práce.

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

OBSAH

ÚVOD

Ve světě programování je jedním z nejdůležitějších faktorů čitelnost kódu. Proto by měl být na tento fakt kladen velký důraz při výuce programování již od úplného začátku. Spousta začátečníků se velmi rychle naučí základy programování, jako je zápis výrazů, podmínek a cyklů, ale když se setkají s více rozvinutou aplikací, dodržující určitý návrhový vzor, mohou mít značné potíže se zorientovat. Existuje několik doporučených postupů, technologií a vzorů, které poskytují vhodný způsob k tvorbě čitelného a testovatelného kódu, kterými se tato práce bude zabývat.

Dalším tématem, na které se tato práce zaměří, bude návrh zadání studijních úkolů pro výuku tvorby klient-server aplikací za využití moderních postupů a doporučení. Studentům bude představen koncept webového aplikačního rozhraní (Web API) a od vývoje jednodušších konzolových aplikací v roli klienta budou úkoly postupně nabývat na složitosti a seznamovat studenty s vhodnějšími technologiemi a postupy. Po splnění navrženého kurzu budou studenti schopni naprogramovat různé typy klient-aplikací pro poskytnuté Web API za pomoci rozlišných frameworků a taktéž vyvinout vlastní webové aplikační rozhraní.

V praktické části budou pro demonstraci rovněž všechny navržené úkoly vyvinuty a popsány klíčové části jednotlivých programů. Web API, vystupující jako bankovní server, vyvinuté v této části práce, bude rovněž sloužit studentům v počátečních úkolech. Na závěr budou zhodnoceny jednotlivé způsoby vývoje, použité technologie a metody strukturování kódu aplikace.

I. TEORETICKÁ ČÁST

1 FRAMEWORK

Framework je platforma pro vývoj softwarových aplikací. Poskytuje určité základy, na kterých mohou softwaroví developeři stavět programy pro specifickou platformu. Framework může například obsahovat předdefinované třídy a funkce, které jsou využity pro zpracování vstupů, obsluhu hardware zařízení, nebo interakci se systémovým softwarem. Toto zjednodušuje vývojový proces, neboť programátor není nucen psát opakovaně základní prvky programu v každé nové aplikaci.

1.1 .NET Framework

.NET Framework je komponentou systémů a platformou společnosti Microsoft, určenou k vývoji aplikací. Skládá se ze dvou hlavních částí: Common Language Runtime (CLR) a knihoven .NET Frameworku [1]. CLR zodpovídá za poskytování základních služeb, jako jsou správa paměti, správa vláken, nebo bezpečnostních opatření. Pro správu paměti se využívá tzv. Garbage Collector. Díky tomuto nástroji se programátor nemusí zabývat přiřazováním a uvolňováním operační paměti, neboť se nepoužívané zdroje uvolní z paměti za pomoci sady komplexních algoritmů, obsažených v CLR. Druhou komponentou jsou knihovny .NET Framework Class Library, rozdělené do jmenných prostorů, nabízející programátorům funkcionalitu pro práci se soubory, přístup k datům, práci s uživatelským rozhraním a mnoho dalších. Tyto knihovny rovněž umožňují vývoj různých typů aplikací, například: konzolové aplikace, aplikace s grafickým uživatelským rozhraním (Windows Forms) nebo webové aplikace (ASP.NET). Pro .Net Framework byl v době jeho vzniku navržen zcela nový programovací jazyk C# [2]. V době, kdy byl tento framework vyvinut, sestával z okolo 3 000 tříd a CLR [2].

1.1.1 Windows Forms

Windows Forms je grafická knihovna, poskytující platformu pro vývoj desktopových aplikací. Tato technologie již u vývoje nových aplikací nebývá častou volbou, a spíše je nahrazována Windows Presentation Foundation (WPF), využívající technologie založené XAML [2].

Základním prvkem této technologie je Form (formulář, či okno). Ten slouží jako kontejner pro komponenty uživatelského prostředí, umožňující uživateli požadovanou interakci. Princip je založen na událostně řízeném modelu – tedy akce uživatele vyvolá událost a spustí se náležitý obslužný kód.

1.1.2 ADO.NET

Technologie ADO.NET, postavená na .NET Frameworku, poskytuje konzistentní přístup k datovým zdrojům (např.: Microsoft SQL Server). Umožňuje tak vývojáři připojení k určitému zdroji dat a odtud získat, zpracovat a aktualizovat obsažená data.

1.1.3 ASP.NET

Framework ASP.NET vznikl jako následník Active Server Pages (ASP) od Microsoftu. [9] Využívá se ke stavění webových aplikací, jako jsou dynamické webové stránky, webové aplikace, či webové služby. Jeho dalšími nástupci jsou ASP.NET MVC a nejmodernější ASP.NET MVC Core, implementující architektonický vzor MVC (viz kapitola 2). Kromě webových služeb umožňují také vývoj REST Web API. Tato programová rozhraní sestávají z jednoho nebo několika veřejně přístupných přístupových bodů (endpointů) k definovanému systému požadavků a odpovědí. Je dostupné na webu, nejčastěji pomocí http webserveru. [3]

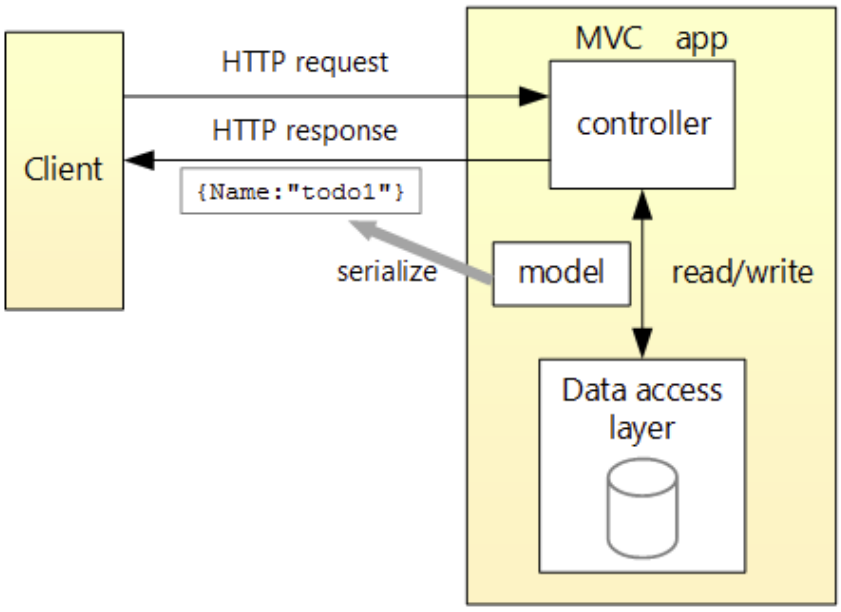
1.1.4 RESTful Web API

Koncept Representational State Transfer (REST) poprvé navrhnul a popsal v roce 2000 Roy Fielding [4] ve své dizertační práci a od této doby konzistentně roste počet dostupných web API. Jedná se o alternativní přístup ke strukturování operací poskytovaných webovou službou. Hlavní výhodou modelu REST je, že je založen na otevřených standardech a neomezuje nijak implementaci klientských aplikací [5]. Klientská aplikace tak může být realizována v libovolném programovacím jazyce a provedení za předpokladu, že zvládá generovat http požadavky a zpracovávat HTTP odpovědi. Webové služby založené na REST přijímají požadavky pomocí jednotné a předdefinované sady operací a umožňují tak přístup a manipulaci s textovou reprezentací svých webových zdrojů.

Navigační schéma reprezentuje objekty a služby systému (označovány jako zdroje). Mnoho systémů implementujících REST dovoluje klientské aplikaci manipulovat se zdroji za pomoci požadavku HTTP směrovaným na definovanou cestu (URI) a metody protokolu HTTP (nejčastější GET, POST, PUT, nebo DELETE) značící žádanou operaci s daným zdrojem [4]. Důležitým principem tohoto konceptu je bez stavový tvar požadavků. To znamená, že požadavky na službu mohou přicházet v libovolném pořadí a neměly by spolu nijak souviset. Jediné místo uchovávající informaci jsou samy zdroje a každý požadavek je atomickou operací. Proto by ani například ověřování uživatele nemělo být řešeno uchováváním informace

ve formě cookies, či záznamem v session. Tuto funkcionalitu je možné zajistit doplněním ověřovacích údajů ke každému požadavku, podle kterých REST API pozná, kdo požadavek odeslal.

Na následujícím obrázku je zobrazeno schéma znázorňující tento systém.



Obrázek 1 – Základní design web API [6]

Dalším ze základních rysů konceptu je povědomí o relacích mezi různými zdroji, ke kterým systém nabízí přístup. Tyto zdroje bývají typicky organizovány jako kolekce a jejich vzájemné relace. Jako příklad může být uveden bankovní systém obsahující informace o klientech a jejich bankovních účtech. Každý klient i účet systému by měl být jasně označen unikátním identifikačním klíčem. Je běžnou praxí, že přístupový bod ve tvaru */clients,* vrací seznam všech klientů pro HTTP požadavek GET. Pro získání textové reprezentace jednoho vybraného klienta by tento přístupový bod mohl být doplněn o unikátní identifikátor požadovaného subjektu. [4]

Webová služba implementující REST by také měla vracet informaci formou stavového kódu, umožňující developerům vyvíjejícím klientské aplikace identifikovat případné chyby v odesílaných požadavcích [4]. Ty nejpoužívanější jsou zobrazeny v tabulce č 1.

Tabulka 1 – Popis nejpoužívanějších stavových kódů

200	OK	požadavek proběhl v pořádku
201	Created	byl vytvořen nový obsah
400	Bad Request	požadavek na server je nějakým způsoben nečitelný
401	Unauthorized	klient není ověřen
403	Forbidden	klient nemá přístup k danému obsahu
404	Not Found	zdroj není nalezen

1.2 Xamarin

Platforma Xamarin, od stejnojmenné společnosti založené roku 2011, nabízí multiplatformní vývoj aplikací [7]. Je cílena především na vývoj aplikací, u kterých hraje hlavní roli logika v pozadí aplikace a není kladen velký důraz na jednotnost prezenční vrstvy u všech platforem.

1.2.1 Xamarin.Forms

V květnu roku 2014 představila společnost Xamarin novou technologii Xamarin.Forms, umožňující psát kód kompilovatelný pro iOS, Andriod a Windows zařízení. Xamarin.Forms podporuje pět různých aplikačních platforem [8]:

- iOS pro programy běžící na zařízeních iPhone, iPad a iPod Touch
- Android pro programy chytrých telefonů a tabletů Android
- Universal Windows Platform (UWP) pro aplikace systému Windows 10 a Windows 10 Mobile
- Windows Runtime API systému Windows 8.1
- Windows Runtime API systému Windows Phone 8.1

Obecně sestává aplikace Xamarin.Forms z několika (až pěti) oddělených projektů pro každou z těchto platforem a jednoho projektu obsahujícího sdílený kód.

Knihovny Xamarin.Forms.Platform obsahují třídy zvané rendery, které transformují objekty uživatelského rozhraní Xamarin.Forms na UI objekty ostatních platforem. Například objekt pro přepínání booleovské hodnoty je v Xamarin.Froms známý pod identifikátorem S*witch*,

který je pak individuálně vygenerován jako *UISwitch* pro iPhone, S*witch* pro Android a *ToggleSwitch* na Windows Phone. [8]

2 TECHNOLOGIE, DOPORUČENÉ POSTUPY, NÁVRHOVÉ VZORY A TECHNIKY TVORBY KLIENT-SERVER APLIKACÍ

2.1 Architektura

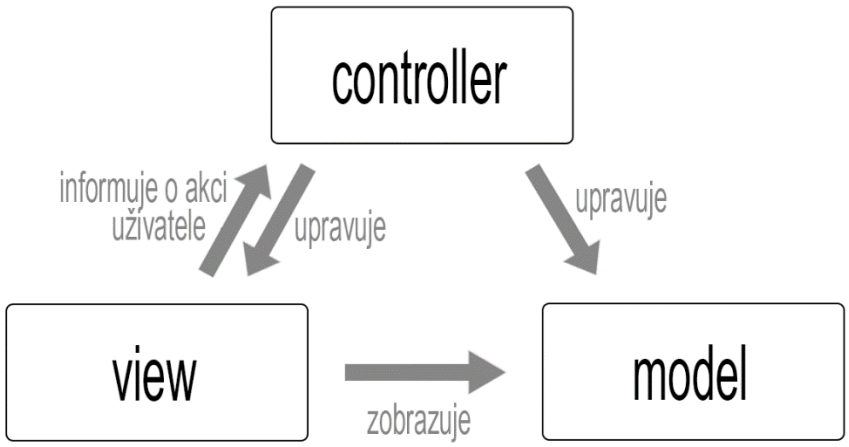
2.1.1 MVC

Architektonický vzor Model-View-Controller má počátky kolem roku 1970 [10] a od té doby hraje velmi vlivnou roli u několika frameworků a chápání návrhu uživatelského rozhraní. MVC rozděluje interakci uživatele s rozhraním do tří rozdílných rolí.

Model je objekt, reprezentující informaci o určité části systému. Nese pouze data a chování daného objektu bez jakékoliv informace o jejich zobrazení uživateli.

View reprezentuje zobrazení modelu v uživatelském rozhraní (UI). Pokud tedy modelem bude klient v bankovním systému, naším view pak může být například HTML stránka zobrazující informaci o tomto klientovi. View se stará pouze o zobrazovaní, jakékoliv změny těchto informací jsou obstarávány třetím prvkem této architektury – controllerem [12].

Controller zpracovává vstup uživatele, manipuluje s modelem a zajišťuje náležité aktualizování view.



Obrázek 2 – Schéma ilustrující architektonický vzor MVC

Tento vzor tak rozděluje prezentaci dat (view) od nositele dat (modelu), což je jedna ze základních podstat dobrého softwarového návrhu. Toto rozdělení je důležité z několika důvodů [11]:

- Uživatelské rozhraní a zpracování dat jsou dvě rozlišné záležitosti. Pokud se programátor zabývá vývojem UI, zajímají ho mechanismy a rozložení rozhraní s cílem vytvořit srozumitelné, uživatelsky přívětivé rozhraní. Při práci s modelem je potřeba přemýšlet nad obchodní logikou, nebo interakcí s databází. V obou případech se tedy vývojář zabývá naprosto rozlišnými vrstvami systému a s největší pravděpodobností bude využívat rozdílné knihovny a funkce.
- V závislosti na kontextu vyžadují uživatelé často zobrazení stejného základního modelu různými způsoby. Při oddělení těchto dvou vrstev pak není problém používat stejné informace modelu ve zcela odlišných rozhraních. Je možné tak využít stejný kód modelu například v konzolové aplikaci, ve webovém prohlížeči, či aplikačním rozhraní (API). Dokonce i v jedné webové aplikaci může více stránek různě zobrazovat stejný model.
- Nevizuální objekty jsou mnohem snadněji testovatelné. Oddělení prezentační části a modelu nám tedy ještě usnadní testování systémové logiky bez nutnosti manuální interakce s rozhraním, či psaní skriptovacích nástrojů pro testování pomocí grafického uživatelského rozhraní.

Klíčovým bodem tohoto rozdělení je směr závislosti: zobrazení závisí na modelu, ale model nezávisí na své grafické interpretaci. Programování modelu by tak mělo být možné zcela bez znalosti jeho prezentace. Z toho vyplývá, že změny rozhraní pak mohou být libovolné, bez nutnosti následných úprav modelu. [11]

Tento princip však nese i určitá úskalí. Více rozsáhlá klientská rozhraní, pracující s několika okny současně, přináší riziko výskytu více prezentací jednoho modelu současně. Pokud tak uživatel provede změny v jednom zobrazení modelu, ostatní se musí náležitě změnit také. Pro zajištění správné funkčnosti v této situaci, bez zavedení závislosti, bývá implementován vzor pozorovatele. Rozhraní pak pracuje jako pozorovatel změn modelu. Při jakékoliv změně vyšle model informaci o události a obnoví se zobrazená informace. [11]

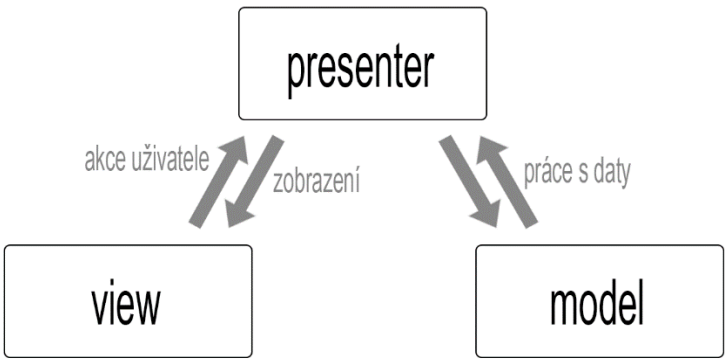
S tímto návrhovým vzorem se nejčastěji setkáme u vývoje webových aplikací.

2.1.2 MVP

Architektura Model-View-Presenter je podobná výše popisované v oblasti rozdělení na jednotlivé vrstvy. Třída view (může se jednat o aplikační okno) se stará o zobrazení a interakci uživatele. Události vyvolané touto interakcí posílá třídě presenter, která obsahuje logiku pro odpověď na danou událost, aktualizuje model (obchodní logika a data aplikace) a na základě obsažené logiky manipuluje se stavem view. Pro zjednodušení testování mívá presenter referenci na rozhraní view místo konkrétní implementace. Díky tomu je možné snadno nahradit skutečné view pomocí takzvané „mockovací“ implementace určené právě pro testování. [12]

Když je Model změněn, je potřeba obnovit View, aby zobrazilo provedené úpravy. Toho lze dosáhnout několika způsoby. Varianty MVP architektury Passive View a Supervising Controller umožňují různé přístupy implementace obnovení view [12].

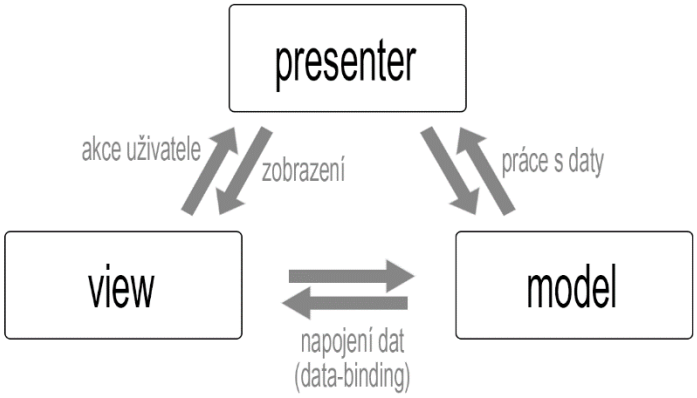
V případě Passive View (pasivní pohled) se stará presenter o aktualizaci view při změnách modelu. View nemá žádné povědomí o změnách modelu samotného, pouze informaci od presenteru. Jednoduché schéma této varianty je zobrazeno na obrázku 3.



Obrázek 3 – MVP varianta Passive View

U Supervising Controller (dohlížející kontroler) dochází k interakci view přímo s Modelem k dosažení jednoduchého napojení dat (data-binding), kterého je možné dosáhnout deklarací, bez nutnosti zásahu presenteru [12]. Presenter se stará o změny modelu a manipuluje se stavem view pouze v případě, kde je potřeba více komplexní UI logika, která se nedá vyřešit

deklarací (například dynamicky skrývat a zobrazovat ovládací prvky). Schéma varianty Supervising Controller je zobrazeno na obrázku 4.

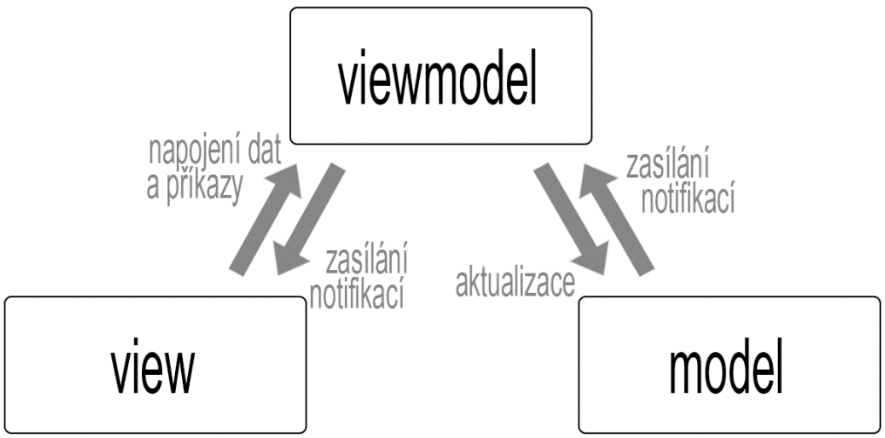


Obrázek 4 – MVP varianta Supervising Controller

S architekturou MVP se můžeme setkat například při vývoji Windows Forms aplikací.

2.1.3 MVVM

Návrhový vzor Model-View-ViewModel je možné využít u všech XAML platforem [13]. Stejně jako u dříve popsaných architektonických vzorů je jeho záměrem poskytnout čisté oddělení uživatelského rozhraní od logiky. Tři dílčí komponenty tohoto vzoru, model, view a viewmodel hrají různé, oddělené role. Toto schéma je vyobrazeno na následující ukázce.



Obrázek 5 – Schéma návrhového vzoru MVVM

ViewModel izoluje view od tříd modelu a umožňuje tak na sobě navzájem nezávislý vývoj obou složek.

View zodpovídá za strukturu, rozložení a vzhled rozhraní, poskytnutého uživateli. Ideálně je view definováno čistě pomocí XAML, s minimem kódu obsahujícího obchodní logiku (business logic) [13]. Typicky bývá view realizováno pomocí jedné stránky aplikace, nebo jako podřazená komponenta jiného view [14]. View získává data ze svého viewmodelu pomocí datového napojení (data-binding), nebo voláním metod viewmodelu.

Pro provedení data bindingu mezi prvkem uživatelského rozhraní a viewmodelem se využívá interface ICommand [14]. Díky tomu je možné vykonat kód viewmodelu na základě interakce uživatele, jako je například stisk tlačítka nebo výběr možnosti, bez nutnosti ošetření této události přímo ve view.

Model je v architektuře MVVM implementací objektu aplikace, obsahujícího data společně s obchodní a validační logikou. Příkladem takových objektů jsou obchodní modely, položky z databáze nebo objekty pro přenos dat (DTO).

Viewmodel pracuje jako prostředník mezi view a modelem a je zodpovědný za zajištění logiky view. Viewmodel získává data a volá metody modelu a následně je poskytuje v co nejjednodušší formě do view. Kromě toho také implementuje příkazy, které může uživatel svou interakcí vyvolat ve view. [13]

Viewmodel dále zodpovídá za změny stavu, ovlivňující určitý aspekt zobrazení ve view, jako je například indikace právě probíhající operace. Viewmodel implementuje interface INotifyPropertyChanged a vyvolá událost PropertyChanged v případě, že se změnila některá vlastnost [14].

Výhody architektonického vzoru MVVM jsou následující [13]:

- Umožňuje oddělení vrstvy starající se o vzhled a zobrazení uživatelského zobrazení uživateli od vrstvy starající se o logiku a komplexní operaci v pozadí. Díky tomu mohou na aplikaci pracovat současně designéři a developeři a soustředit se pouze na svou část úkolu.
- Implementace jednotkových testů viewmodelu je možná bez nutnosti použití view.
- Bezproblémové přepracování designu UI aplikace bez nutnosti úprav kódu. Nová verze view by měla pracovat s již hotovým viewmodelem.

2.2 Testování

Pro zaručení správné požadované funkčnosti programu je nezbytné testování vyvíjeného softwaru. Vzhledem k tomu, že některé moderní systémy mají tendenci růst do značných velikostí, je výhodné je členit na jednotlivé menší programy. V této situaci je testovatelnost takového softwaru klíčovou vlastností, neboť nově přidané funkce musí správně fungovat společně s již existujícím systémem a zároveň jej nesmí poškodit. Pokud není kód dostatečně členěn, může se stát, že testování velmi krátkého úseku programu vyžaduje zdlouhavý průběh celé aplikace. Proto je vhodné jednotlivé moduly co nejvíce izolovat. [15]

Inversion of Control (IoC) je technika měnící běh programu. To pak umožní využití komponenty programu bez nutnosti existence reference na jinou vyžadovanou komponentu. Jednou z implementací vzoru IoC je Dependency Injection (DI), česky vkládání závislostí. [15]

2.2.1 Dependency Injection

Hlavní myšlenkou DI je implementace IoC za pomocí vložení závislostí do objektů z externí komponenty místo jejich vytváření uvnitř objektu. Klasické vytváření je znázorněno na následující ukázce kódu, která zobrazuje popis třídy, starající se o manipulaci s daty klientů – *ClientRepository*. Tato třída vyžaduje pro svou funkci připojení do databáze a operace CRUD (Create, Read, Update, Delete) poskytnuté třídou *DbContext*, a tak si ve svém konstruktoru vytváří její instanci. [11]

```
public class ClientRepository
{
    private DbContext context;

    public ClientRepository()
    {
        this.context = new DbContext();
    }
```

Zde je zřejmé, že pro testování *ClientRepository* vznikne nová instance *DbContext*, aniž by bylo možné tuto závislost jakkoliv ovlivnit. V další ukázce je stejný případ rekonstruován za pomoci vložení závislosti.

```
public class ClientRepository
{
    private readonly IDbContext _context;

    public ClientRepository(IDbContext context)
    {
        _context = context;
    }
```

Zde je do konstruktoru jako parametr posílán interface třídy *DbContext*, a tak je její vznik naprosto oddělen od třídy *ClientRepository*.

Existují tři typy vkládání závislostí pro implementaci IoC pomocí Dependency Injection.

Contructor Injection

Vkládání závislostí pomocí konstruktoru je jedním z nejrozšířenějších způsobů implementace tohoto vzoru [16]. Tato metoda byla představena v ukázkovém kódu na předcházejícím vzorovém případu. Konstruktor přijme veškeré své závislosti jako argumenty a je tak zajištěn vznik objektu se všemi potřebnými závislostmi během samotného provádění konstruktoru.

Setter Injection

Vkládání závislosti pomocí setteru je také běžným způsobem a využije se v případech, kdy má objekt větší počet závislostí [16]. V takovém případě je lepší volbou nastavovat jednotlivé závislosti zvlášť pomocí setterů (metod určených pro nastavení dané vlastnosti), než aby se do konstruktoru současně posílalo mnoho argumentů. Ukázka tohoto typu DI je zobrazena v následující části kódu.

```
public class ClientRepository
{
    private IDbContext context;

    public void SetDbContext(IDbContext context)
    {
        this.context = context;
    }
```

Interface Injection

Posledním typem DI je vkládání pomocí implementace rozhraní. Tento typ se podobá Setter Injection, pouze metoda pro nastavení závislosti se získá implementací rozhraní. Tento typ patří mezi ty méně používané metody [16].

```
public class ClientRepository : InjectDbContext
{
   private IDbContext context;

   public void injectDbContext(IDbContext context)
   {
       this.context = context
   }
}

public interface InjectDbContext
{
    void injectDbContext(IDbContext context)
}
```

2.2.2 Jednotkové testy

Jednotkové testy jsou zaměřeny na testování jistých funkcí, či oblastí v kódu – jednotek. Poskytují tak vývojáři povědomí o správné funkčnosti dané části programu. Testy by měly zajistit, že aplikace vykoná správně danou funkci s libovolnými parametry a obstará všechny výjimky, které mohou nastat. [17]

Při vývoji větších projektů je vhodné současně s růstem aplikace implementovat nové jednotkové testy pro veškerou přibývající funkcionalitu. Díky tomu lze zajistit nejen správné chování nových funkcí, ale také zanechání originálních funkcí nedotčených. Během stavby aplikace se tak neustále kontroluje dosavadní kód a chyba, způsobená úpravami, je bezprostředně odhalena. [17]

Aby bylo možné používat jednotkové testy, je nutné tomu přizpůsobovat kód [17]. Je vhodné používat méně obsáhlé funkce, soustředěné na konkrétní operaci. Kromě zjednodušení testování se s rozčleněným kódem rovněž zvýší jeho čitelnost.

Zavedení testů samozřejmě nese nevýhodu ve vyšší časové náročnosti vývoje, ale s rostoucí velikostí projektu usnadňuje problém s poškozenou existující funkcionalitou při zavedení nové funkcionality.

2.2.3 Test-driven development

Jedním z možných přístupů vývoje je tzv. Test-driven development (programování řízené testy) [18]. Jedná se o techniku psaní kódu, kdy programátor nejprve navrhne a sestaví test samotné nové funkcionality a teprve poté připraví implementaci v kódu.

Prvním krokem je tedy příprava testu, který ověřuje, zda nově přidaný blok kódu pracuje tak, jak je zamýšleno. Tím je jasně stanoven přesný popis a chování funkce, která má být do systému implementována. Poté, co jsou všechny nové testy nachystány, musí být otestovány pro vracení chybného vyhodnocení. Tímto krokem se eliminují případné chyby v kódu testu samotného. [18]

Dalším krokem je napsání kódu vykonávajícího novou funkcionalitu tak, aby byly splněny veškeré testy na něj zaměřené. Zde často programátor volí nejsnadnější a nejrychlejší přístup, bez ohledu na efektivnost kódu [18]. Pokud je však kód správný a splňuje všechny podmínky stanovené testy, přichází na řadu refaktorizace kódu, kdy je snahou developera udělat kód co nejelegantnější a nejefektivnější. Během tohoto procesu je funkce průběžně kontrolována a testována, aby se při úpravách nezměnil výsledek jednotlivých testů.

2.2.4 Vzor AAA

Vzor Arrange, Act, Assert (příprava, vykonání, predikce) je běžným způsobem psaní jednotkových testů pro testovanou metodu [18]. Metoda testu je podle tohoto vzoru rozdělena na tři části:

- Sekce **Arrange** se stará o inicializaci objektů a nastavení hodnot dat, která jsou předávána testované metodě.
- Sekce **Act** se potom stará o samotné vykonání testované metody s připravenými daty.
- V poslední části **Assert** se ověřuje, zda výsledek testované metody funguje dle očekávání. Lze zde ověřit nejen správně vrácená data, ale i správné chování v případě výjimky.

Tento vzor je aplikován v následující ukázce kódu na jednoduché, demonstrační metodě pro sčítání.

```
[TestClass]
public class CalculatorTests
{
    [TestMethod]
    public void TestAddition()
    {
        // arrange
        var calculator = new Calculator();
        var expectedResult = 5;

        // act
        var result = calculator.Add(3, 2);

        // assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

2.3 Objektově relační mapování

V objektově orientovaném programování je vše ztvárněno pomocí objektů. V relačních databázích se naopak setkáváme s tabulkami a relacemi mezi nimi. Při programování nastává často problém, který vyžaduje propojení těchto dvou konceptů. Ke konverzi mezi těmito dvěma datovými reprezentacemi existuje technika Object-Relational Mapping (ORM). K realizaci této techniky existuje celá řada frameworků a balíčků, v některých případech může programátor volit možnost vlastní implementace ORM.

2.3.1 Entity Framework

Jedním z frameworků, poskytujících techniku ORM, je Entity Framework vydán jako součást ADO .NET v srpnu roku 2008 [19]. Entity Framework umožňuje dva hlavní přístupy vývoje datově orientovaných aplikací využívajících databázi. Jedná se o přístup Code First, zaměřený na popis modelu pomocí tříd C# dle stanovených konvencí v dokumentaci tohoto frameworku a za využití takzvaných data annotations, k přesnější definici jednotlivých vlastností. Jednou z důležitých součástí tohoto frameworku je *DbContext.* Tato třída slouží pro interakci s databází a udržuje informaci o datových objektech.

Při využívání code first vývoje se většinou nejprve tvoří konceptuální model v .NET Frameworku a stanoví se, které tyto třídy budou zahrnuty pomocí *DbContex*. K tomuto účelu vzniká vlastní implementace kontextu vycházejícího z *DbContext.* Kontext ob-

sahuje vlastnosti *DbSet* pro typy, které mají být součástí modelu. Entity Framework se následně postará o to, aby byly zahrnuty všechny tyto typy a jejich případné reference a vygeneruje náležitou databázi. [20]

Entity Framework označí vlastnost jako primární klíč tu, která je pojmenována „ID“, nebo stejně jako název třídy doplněný klíčovým slovem „ID“. Pokud je tento typ numerický, nebo typ Guid, potom je rovněž sloupec nakonfigurován jako identity column (sloupec unikátních hodnot) [21].

Druhou možností přístupu je Database First, která umožní generování .NET modelů z již existující databáze. S touto technikou se často setkáme při zavádění ORM do již existujících projektů.

II. PRAKTICKÁ ČÁST

3 NÁVRH

Tato část bakalářské práce se bude zabývat návrhem zadání úkolů pro výuku tvorby klient-server aplikací za využití moderních postupů a technologií. Zadané úkoly budou postupně nabývat na složitosti, od jednodušších aplikací postupně až ke komplexnějším, lépe strukturovaným řešením. Studenti tak uvidí v praxi porovnání vývoje nejprve jednoduchého projektu, jaký dokáže vytvořit nejspíše většina začátečníků, a projektu implementujícího osvědčené praktiky a doporučené postupy.

3.1 Popis systému

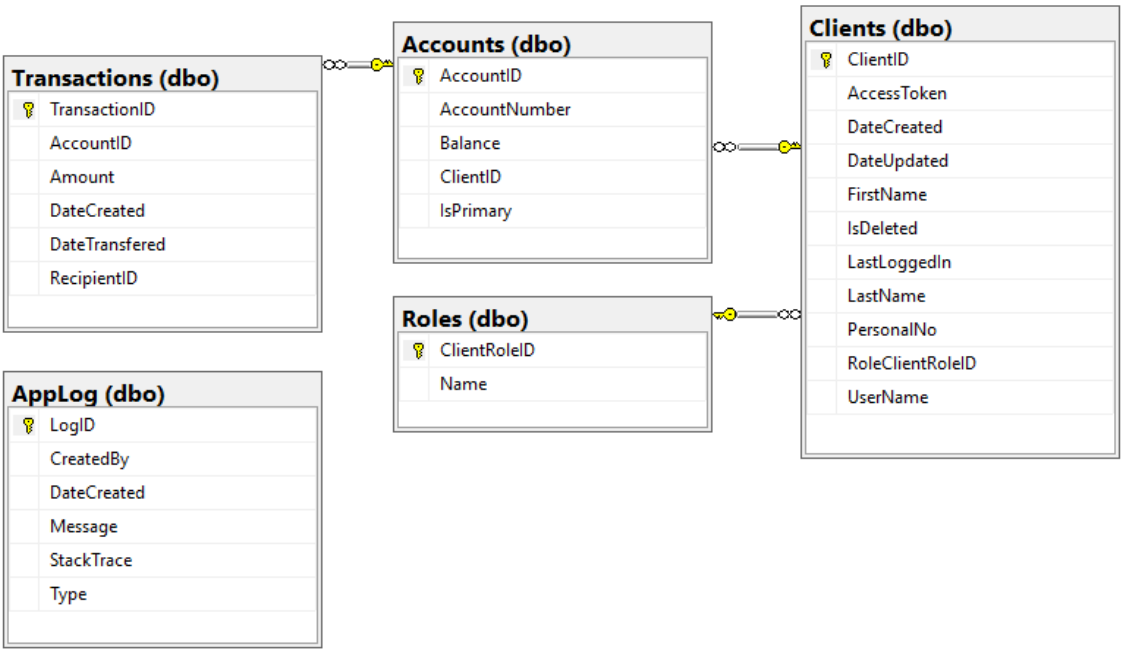
Cílem návrhu obsahu výuky je, aby studenti připravili systém sestávající z bankovní webové služby – serveru, a několika typů aplikací tento server využívajících – klientů. Bankovní server zpřístupňuje svým zákazníkům informace o jejich účtech a umožňuje jim na základě ověření totožnosti určitou funkcionalitu.

3.1.1 Bankovní server

Bankovní server je realizován pomocí REST web API frameworku .NET Core. Server bude poskytovat některé ze svých zdrojů veřejně a současně pro přístup k dalším zdrojům bude vyžadovat ověření (autentizaci) uživatele. Webová služba bude spojena s databází, ve které budou vygenerovány účty zapsaných studentů v daném předmětu. K získání veřejných informací o těchto studentech bude využita webová služba portálu IS/STAG Univerzity Tomáše Bati ve Zlíně.

Struktura

Databáze bankovního serveru bude sestávat z několika tabulek a vzájemných relací, které jsou zobrazeny na následující ukázce.



Obrázek 6 – Struktura databáze bankovního serveru

Tabulka klientů bude plněna automaticky na základě přihlášených studentů a ponese informaci o jednotlivých uživatelích systému. Každý klient v systému může mít libovolný počet bankovních účtů, jak je znázorněno relací 1:N na diagramu databáze. Dále pak zavedeme vazbu na tabulku rolí v systému. Poslední vázanou tabulkou systému jsou transakce na účtech. V návrhu databáze je znázorněna tabulka *AppLog* bez relace. Tato bude sloužit pouze k zaznamenávání chyb, varování, či informačních zpráv pro kontrolu a správu systému.

Zdroje

Veřejným zdrojem se rozumí kolekce bankovních účtů v systému, přístupná bez jakéhokoliv ověření. Tento zdroj bude poskytovat pouze veřejně dostupnou informaci a nebude odkrývat žádná citlivá data.

Pro přístup k soukromým zdrojům je vyžadována autentizace pomocí externího přihlášení přes portál IS/STAG. K tomuto účelu bude webová služba poskytovat přístupový bod, na který budou klientské aplikace přesměrovávat uživatele po přihlášení. Díky tomu webová služba získá přístupový token daného uživatele, pomocí kterého bude provádět autentizaci.

Server umožní přihlášenému uživateli získat detail svého uživatelského účtu a náležících bankovních účtů. Dále poskytne uživateli možnost vytváření nových a výpis všech transakcí na vybraném bankovním účtu.

Dokumentace

Poslední funkcionalitou zhotovovaného serveru bude veřejně přístupná dokumentace popisující možnosti a práce s touto službou.

3.1.2 Klient aplikace

Pro výše popsanou webovou službu vznikne několik typů klient aplikací, na kterých si studenti postupně osvojí vybrané technologie a doporučené postupy vývoje. Bude se jednat o postupný vývoj konzolové aplikace, nabývající na komplexnosti. Další variantou aplikace v roli klienta bude desktopová aplikace zhotovená pomocí technologie Windows Forms a na závěr multiplatformní aplikace pro mobilní telefony vyvinutá za použití frameworku Xamarin.Forms.

3.2 Návrh úkolů

3.2.1 Úkol 1 – Konzolová aplikace pro komunikaci s web API

Cíl: Seznámení se s třídou HttpWebRequest a základním principem komunikace s web API.

Úkol: Realizujte konzolovou aplikaci .Net Framework, která bude stahovat veřejné údaje o všech bankovních účtech z web API dostupné na adrese http://faibankapi.azurewebsites.net/. Získaná data vypište buď na konzoli, nebo zapisujte do souboru. Pro komunikaci s webovým aplikačním programovacím rozhraním je možné využít třídu HttpWebRequest, a požadavek odeslat na endpoint „/api/accounts“

Dokumentace web API je dostupná na adrese http://faibankapi.azurewebsites.net/swagger.

3.2.2 Úkol 2 – Konzolová aplikace umožňující serializaci a deserializaci

Cíl: Seznámení se se správcem NuGet balíčků, konkrétně pak s balíčkem Newtonsoft.Json, a jeho následným využitím pro serializaci a deserializaci dat.

Úkol: Vytvořte třídu (model) reprezentující bankovní účet, odpovídající strukturou data obdržená z Web API v prvním úkole. Rozšiřte stávající aplikaci o serializaci získaných dat bankovních účtů na připravené objekty a jejich vybrané vlastnosti vyobrazte na konzoli.

3.2.3 Úkol 3 – Konzolová aplikace využívající externí přihlášení přes portál IS/STAG UTB

Cíl: Pokročilá práce s třídou HttpWebRequest, používání Network Credentials.

Úkol: Pro používání většiny dotazů web API bankovního systému je za potřebí externího přihlášení přes IS/STAG a získání přístupového tokenu (access token). Portál IS/STAG UTB nabízí webovou službu, popsanou na adrese https://stag-ws.utb.cz/ws/web?pp_locale=cs&pp_reqType=render&pp_page=tech, kterou pro tuto problematiku využijete. Stávající aplikaci rozšiřte o metodu Login, implementující externí přihlášení do systému pomocí přihlašovacích údajů IS/STAG UTB.

Pro obdržení access tokenu je potřeba odeslat své přihlašovací údaje ve formě „NetworkCredential“ třídy HttpWebRequest na endpoint https://stag-ws.utb.cz/ws/login?originalURL=http%3A%2F%2Ffaibankapi.azurewebsites.net/api/security&basic=1.
Systém STAG poté přesměruje odpověď na endpoint web API, obsahující přihlašovací

jméno a access token. Tyto údaje jsou k dispozici v property „ResponseUri.Query“ třídy WebResponse. Pro realizaci následujících úkolů je nutné tuto informaci uchovat v programu.

3.2.4 Úkol 4 – Konzolová aplikace – klient bankovního serveru

Cíl: Tvorba plně funkční klientské aplikace využívající doposud získané znalosti. Zpracování odpovědí serveru, ošetření výjimek.

Úkol: Rozšiřte konzolovou aplikaci o implementaci funkčnosti umožňující následující scénář. Ošetřete všechny možné případy výjimek a informujte o případných chybách klienta formou zprávy na konzoli. Snažte se strukturovat kód tak, aby byl snadno čitelný.

Scénář: Uživatel se přihlásí do systému pomocí externí služby přes portál IS/STAG. V systému uvidí své osobní údaje, informace o svých bankovních účtech a historii všech transakcí. Transakce se dále dají filtrovat pouze na příchozí platby, nebo odchozí platby. Dále může uživatel zadat příkaz k úhradě (vytvoření nové transakce).

3.2.5 Úkol 5 – Konzolová aplikace .NET Core

Cíl: Student se naučí pracovat s vkládáním závislostí pomocí výchozí implementace Dependency Injection v NuGet balíčku *Microsoft.Extensions.DependencyInjection*. Seznámí se rovněž s návrhovým vzorem *Repository pattern*. Student se rovněž seznámí s asynchronními metodami frameworku .NET Core.

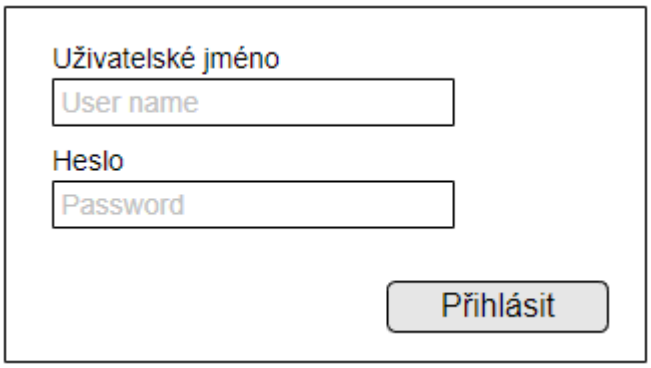
Úkol: Vytvořte konzolovou aplikaci .NET Core implementující stejnou funkcionalitu jako verze v úkolu 4. Aplikace však bude strukturována dle vzoru *Repository pattern* a bude implementovat vkládání závislostí (DI). K tomu lze využít vestavěný framework instalací NuGet balíčku *Microsoft.Extensions.DependencyInjection*.

3.2.6 Úkol 6 – Desktopová aplikace Windows Forms

Cíl: Seznámení se s technologií Windows Forms .NET Frameworku a návrhového vzoru MVP.

Úkol: Vytvořte desktopovou aplikaci za použití technologie Windows Forms poskytující stejnou funkcionalitu jako předchozí konzolové verze (viz scénář – úkol 4). Celý program implementujte pomocí vhodně navržené architektury MVP a strukturujte do menších modulů, starajících se o nezávislé části programu. Aplikaci můžete implementovat dle následující specifikace:

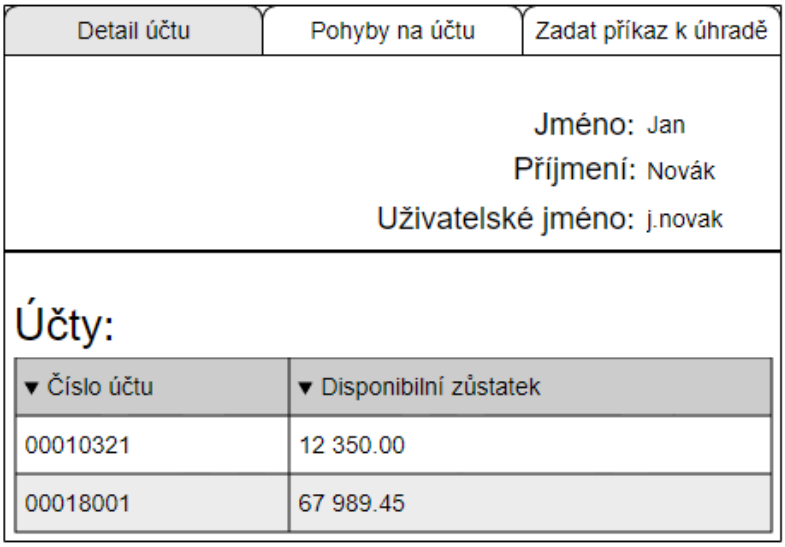
Po spuštění aplikace je uživatel vyzván k přihlášení do systému za pomoci externího přihlášení portálu IS/STAG UTB, viz obrázek 7.



Obrázek 7 – Formulář pro přihlášení do systému

Při vyplnění nevalidních dat je uživatel upozorněn na vyskytlou chybu. Po úspěšné autentizaci klienta se přihlašovací formulář zavře a otevře se hlavní formulář aplikace se třemi záložkami pro hlavní funkce programu.

První záložkou je obecné shrnutí informací o majiteli účtu a jeho seznamu jeho bankovních účtů.

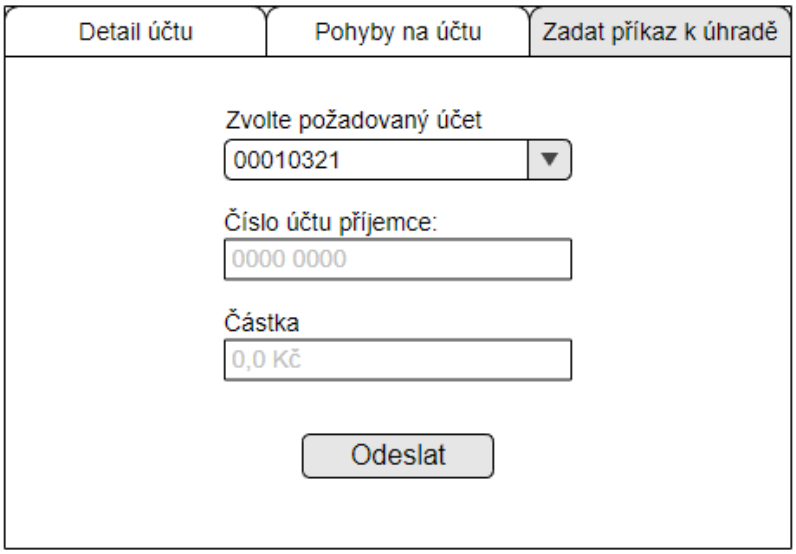


Obrázek 8 – Záložka obsahující detail účtu hlavního okna

Druhá záložka – Pohyby na účtu, poskytuje uživateli menu se seznamem všech přidružených bankovních účtů, a tabulku s přehledem všech transakcí právě zvoleného účtu. Příchozí a odchozí transakce je možno buď filtrovat, nebo budou odlišeny například barevně. Tato záložka je zobrazena na obrázku 9.

Obrázek 9 – Záložka transakcí na účtu

Poslední záložka ztělesňuje rozhraní pro zadávání plateb. Uživateli je opět nabídnut výběr ze všech jeho účtů, textové pole pro vyplnění cílového čísla účtu, textové pole pro zadání odesílané částky a tlačítko „odeslat“. V případě úspěšného, či chybného požadavku je uživatel informován zobrazenou hláškou.



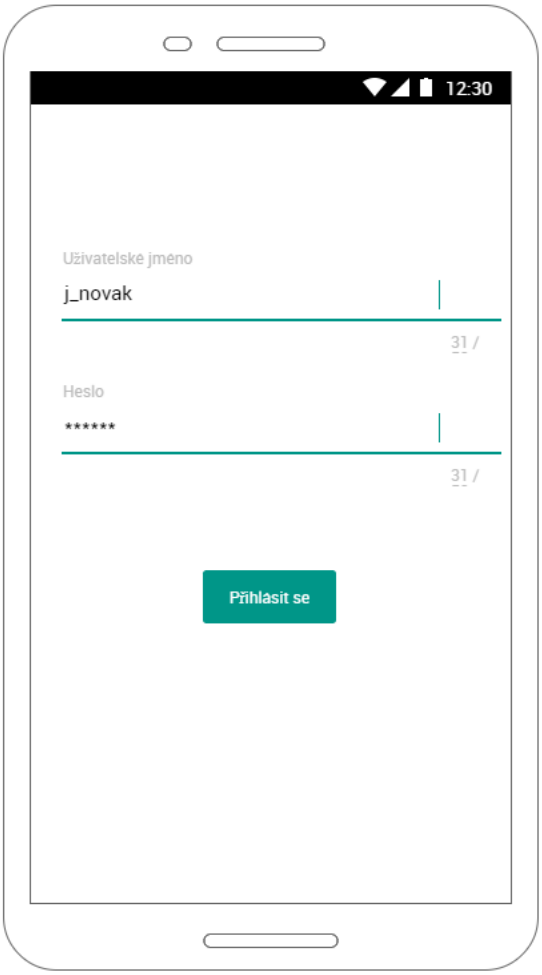
Obrázek 10 – Záložka pro zadání příkazu k úhradě

3.2.7 Úkol 7 – Multiplatformní aplikace Xamarin.Forms

Cíl: Seznámení studenta s multiplatformním vývojem pomocí frameworku Xamarin.Forms. Implementace architektonického vzoru MVVM. Student se naučí základy DataBinding a využití rozhraní *INotifyPropertyChanged* a *ICommand*.

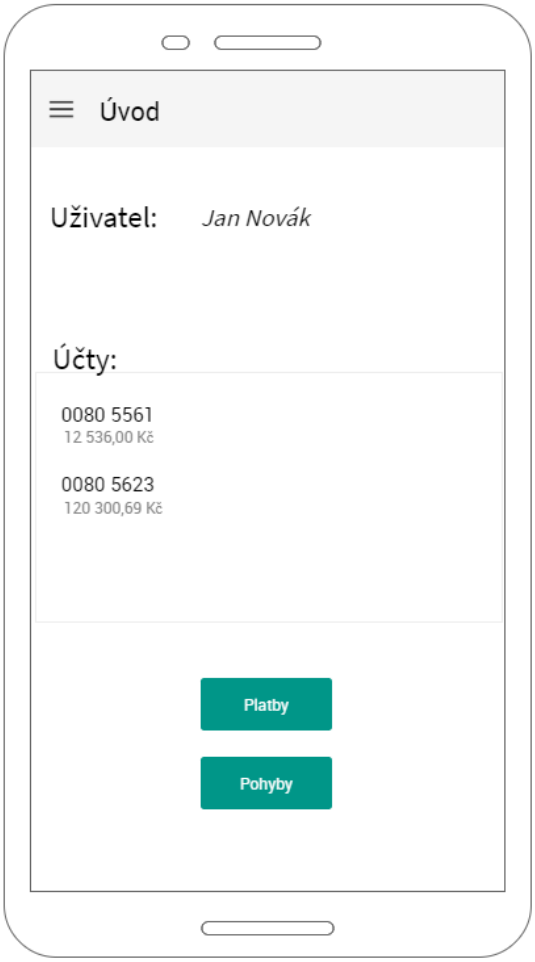
Úkol: Zhotovte multiplatformní aplikaci za pomoci frameworku Xamarin.Forms, umožňující komunikaci s web API http://faibankapi.azurewebsites.net/ tak, jako tomu bylo v předcházejících úkolech. Při vývoji aplikace dodržujte architektonický vzor MVVM a implementujte základní funkcionalitu klienta bankovního serveru dle následující specifikace.

Při spuštění aplikace je uživateli zobrazena stránka s přihlašovacím formulářem zajišťující externí přihlášení pomocí portálu IS/STAG UTB.



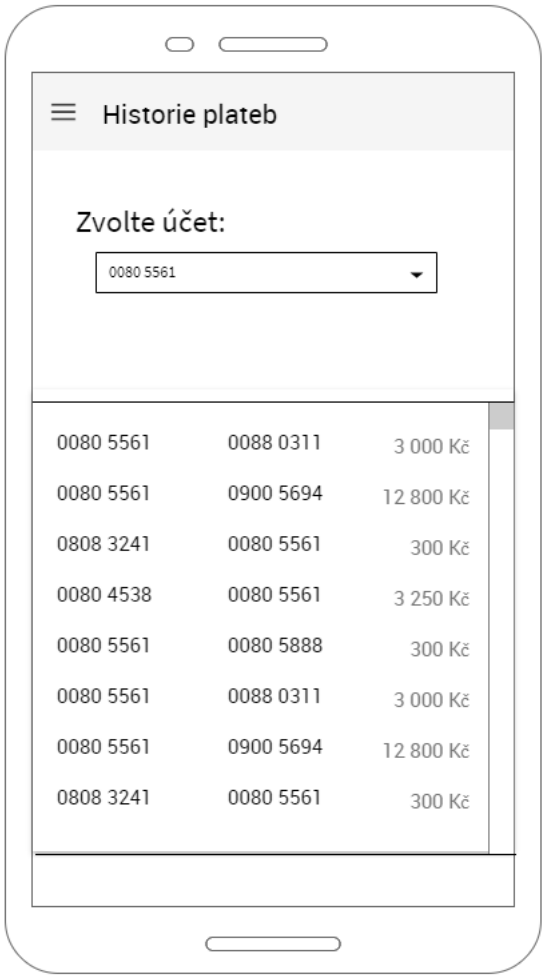
Obrázek 11 – Úvodní stránka s přihlašovacím formulářem (ukázka pro systém Android)

Po úspěšném přihlášení je uživatel přesměrován na obrazovku se základními informacemi o přihlášeném uživateli a jeho bankovních účtech v systému. Tato stránka také obsahuje tlačítko „Pohyby“ a „Platba“ odkazující na novou stránku. Mezi všemi těmito okny lze navigovat pomocí horního panelu. Tuto funkcionalitu lze dosáhnout pomocí implementace *NagivationPage*.



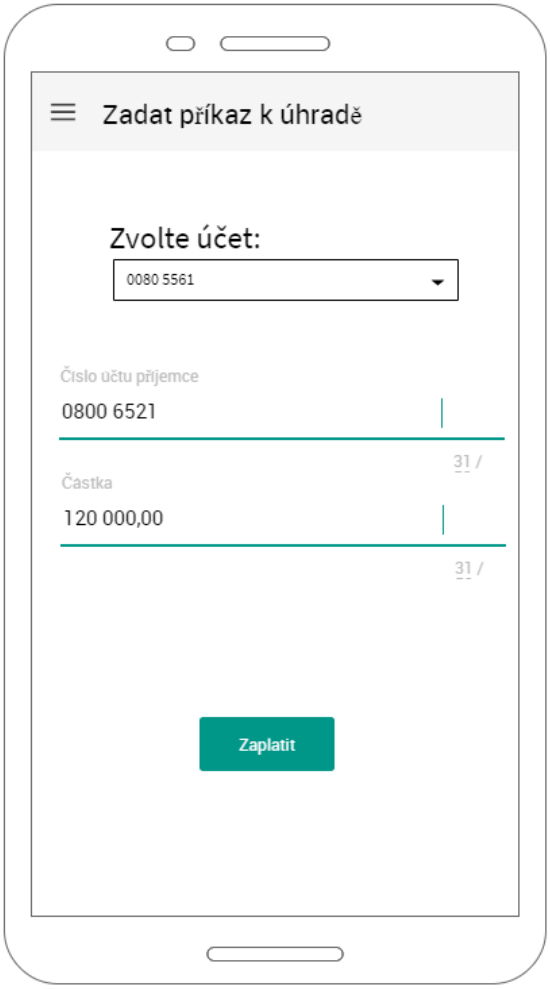
Obrázek 12 – Domovská stránka zobrazující základní informace o uživateli

Tlačítko „Platby“ přesměruje uživatele na stránku obsahující historii všech pohybů na účtu. K dispozici bude rozbalovací menu obsahující seznam všech bankovních účtů náležících danému uživateli. Po zvolení účtu se aktualizuje tabulka s historií transakcí. Pomocí horního menu lze navigovat zpět na úvodní stránku.



Obrázek 13 – Historie plateb na účtu

Posledním oknem je okno pro zadávání příkazu k úhradě, zobrazené po stisku tlačítka „Platba“ z úvodní obrazovky. Okno obsahuje rozbalovací menu poskytující seznam všech bankovních účtů náležících danému uživateli, textové pole pro zadávání čísla účtu, na který převádíme částku a textové pole pro zadávání převáděné částky. Tlačítko „Zaplatit“ volá patřičný dotaz na web API k vytvoření transakce.

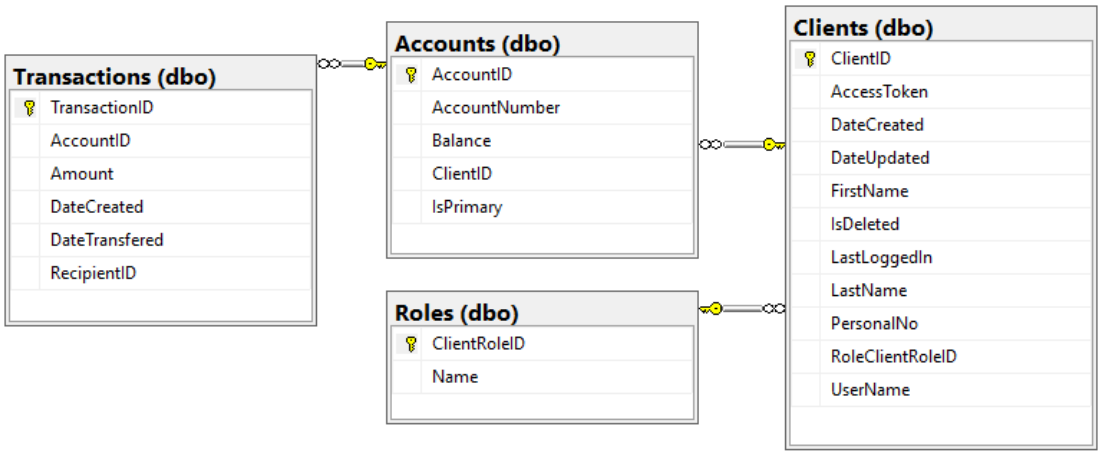


Obrázek 14 – Obrazovka pro tvorbu nové transakce

3.2.8 Úkol 8 – RESTful Web API s implementací ADO.NET

Cíl: Student se seznámí s tvorbou web API, prací s kontrolerem a třídou ActionResult. Cílem je připravit testovatelné API implementací vkládání závislostí. Základní práce s třídou *System.Data.SqlClient* umožňující komunikaci s databází.

Úkol: Vytvořte webovou službu navrženou podle vzoru REST s připojením do lokální SQL databáze. Tato databáze bude strukturována dle schématu na obrázku 15. Strukturu celé aplikace náležitě členěte pro snadnou čitelnost a pro testovatelnost implementujte Dependency Injection za pomoci vestavěného IoC kontejneru v metodě *ConfigureServices* třídy *Startup.cs.* Pro získání dat z databáze využívejte třídu *System.Data.SqlClient*.



Obrázek 15 – Struktura databáze

Systém je možné rozšířit o detailnější informace o uživateli implementací nové tabulky nesoucí tato data. To přidá možnost klientským aplikacím využít možnost doplnění vlastního profilu například o adresu a kontaktní informace a jejich následnou editaci.

3.2.9 Úkol 9 – RESTful Web API s využitím ORM Entity Framework

Cíl: Seznámení studentů s frameworkem objektově relačního mapování, konkrétně pak s přístupem *code-first.* Tvorba testovatelného softwaru. Práce s vkládáním závislostí.

Úkol: Vytvořte webovou službu stejně jako tomu bylo v úkole 8. Pro práci s daty využijte framework objektově relačního mapování – Entity Framework.

3.2.10 Úkol 10 – Implementace Swagger do Web API

Cíl: Obeznámení studenta se způsobem dokumentace web API, vlastní implementace frameworku Swagger k vytvoření specifikace web API z úkolu 9.

Úkol: Do projektu z úkolu 9 nainstalujte NuGet balíček *Swashbuckle.AspNetCore* a pomocí vhodných dokumentačních komentářů připravte specifikaci dříve zhotoveného bankovního serveru. Pro správné generování popisu služby pomocí framworku Swagger je nutné v nastavení sestavení projektu (Properties -> Build) povolit generování dokumentace v XML a výstupní cestu tohoto souboru poskytnout při deklaraci služby Swagger ve třídě *Startup.cs*.

3.2.11 Úkol 11 – Implementace jednotkových testů do Web API

Cíl: Tvorba jednotkových testů, promyšlení možných scénářů aplikace.

Úkol: Vytvořte projekt jednotkových testů *Unit Test Project (.NET Core)* a nastavte mu referenci na hotový projekt web API z úkolu 9. Sestavte jednotkové testy pro kontrolery tohoto bankovního serveru. Vytvořte tzv. mockovací objekty, které nahradí požadovanou závislost kontroleru zástupnými kopiemi, implementujícími chování potřebné pro test.

4 REALIZACE

V této části budou popsány klíčové části vzorových řešení jednotlivých úkolů. Pro testovací účely je ve vystavené verzi webové API vytvořen administrátorský účet s přihlašovacími údaji „admin“.

4.1 Úkol 1 – Konzolová aplikace pro komunikaci s web API

Klíčovou metodou při vypracovávání tohoto úkolu je metoda *RequestData*, využívající třídu *HttpWebRequest,* k odeslání dotazu na web api. Odpověď serveru je následně převedena na řetězec pomocí třídy *StreamReader* a vrácena. V programu pak s tímto řetězcem probíhá další manipulace v podobě výpisu na konzoli, či uložení do souboru.

```
private static string RequestData()
{
   string jsonResponse;
   HttpWebRequest request = (HttpWebRequest)HttpWebRequest
                                                .Create(API_ENDPOINT);
   request.Method = "GET";
   request.ContentType = "application/json; charset=utf-8";

   Console.WriteLine("Posilam pozadavek...");
   var response = request.GetResponse();

   using (var sr = new StreamReader(response.GetResponseStream()))
   {
       jsonResponse = sr.ReadToEnd();
   }

   return jsonResponse;
}
```

V programu používáme web API, která vznikne jako realizace úkolu 9, přesněji potom endpoint */api/Accounts*, vracející veřejně dostupné informace o všech účtech v systému.

4.2 Úkol 2 – Konzolová aplikace umožňující serializaci a deserializaci

Pro realizaci tohoto úkolu je možné rozšířit již zhotovený program z předcházejícího úkolu. Jednou z nejrozšířenějších metod serializace je využití NuGet balíčku *Newtonsoft.Json*, který bude pro tyto účely zaveden do projektu. Pro deserializaci obdrženého objektu ve formátu json je nutné připravit třídu odrážející strukturu obdrženého objektu, jak je znázorněno v následující ukázce.

Obdržená data ve formátu json

```
{
    "AccountID": 228,
    "AccountNo": "00000228",
    "Owner":
    {
      "FirstName": "Jan",
      "LastName": "Novák",
      "UserName": "j_novak ",
      "StagPersonalNumber": "A13993"
    }
}
```

Připravený model

```
public class AccountModel
{
    public int AccountID { get; set; }
    public string AccountNo { get; set; }
    public Owner Owner { get; set; }
}

public class Owner
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string UserName { get; set; }
    public string StagPersonalNumber { get; set; }
}
```

Samotná deserializace přijatých dat z web API je následně uskutečněna v hlavním programu pomocí metody *DeserializeObject* třídy *JsonConvert* znázorněné v následující ukázce kódu.

```
string jsonResponse = RequestData();

    var accounts = JsonConvert
                        .DeserializeObject<List<AccountModel>>(jsonResponse);
```

4.3 Úkol 3 – Konzolová aplikace využívající externí přihlášení přes portál IS/STAG UTB

Pro manipulaci s citlivými daty vyžaduje navržený server validaci pomocí externího přihlášení přes portál IS/STAG UTB. Postup pro využití této služby je popsán v dokumentaci portálu na https://stag-ws.utb.cz/ws/web?pp_locale=cs&pp_reqType=render&pp_page=tech. Autentizace klienta probíhá odesláním přihlašovacích údajů na server UTB společně s adresou pro přesměrování přístupového tokenu (access tokenu) na službu vyžadující externí přihlášení – v tomto případě web API simulující bankovní server.

V kódu je k této funkci využito třídy HttpWebRequest k odeslání dotazu a NetworkCredential jako bezpečný nosič přihlašovacích údajů klienta. Odpověď serveru obsahuje informaci o přesměrování na web API společně se třemi parametry, nesoucími informaci o uživateli. Jedná se o stagUserName, stagUserTicket a stagUserRole. Pro další komunikaci s web API je nutné uchovat v průběhu programu přístupový token z parametru stagUserTicket, v ukázce kódu uložen do proměnné „accessToken“.

```
HttpWebRequest request = (HttpWebRequest)HttpWebRequest.Create(AUTH_ENDPOINT);
request.Method = "GET";
request.UseDefaultCredentials = false;
request.Credentials = new NetworkCredential(UserName, pass);
var response = request.GetResponse();

var accessToken = response.ResponseUri.Query
                  .Replace("?stagUserTicket=", "")
                  .Substring(0, 64);
```

Pro zadávání hesla v konzolové aplikaci je implementována metoda, zaměňující znaky za hvězdičky, pro zvýšení bezpečnosti přihlašovacích údajů. Tato metoda je zobrazena v následující ukázce.

4.4 Úkol 4 – Konzolová aplikace – klient bankovního serveru

Hlavním cílem tohoto úkolu je zkombinovat prvky předcházejících aplikací a vytvořit funkční aplikaci pro komunikaci s web API, umožňující základní funkcionalitu tohoto typu aplikace. Metoda pro přihlášení je totožná s metodou popsanou v úkolu 3 a uživatelské jméno společně s access tokenem je uloženo pro následné využití v průběhu programu.

```
ConsoleKeyInfo key;
string pass = String.Empty;
do
{
	key = Console.ReadKey(true);
	if (key.Key != ConsoleKey.Backspace && key.Key != ConsoleKey.Enter)
	{
		pass += key.KeyChar;
		Console.Write("*");
	}
	else
	{
		if (key.Key == ConsoleKey.Backspace && pass.Length > 0)
		{
			pass = pass.Substring(0, (pass.Length - 1));
			Console.Write("\b \b");
		}
	}
} while (key.Key != ConsoleKey.Enter);
```

Po přihlášení se spouští hlavní smyčka programu, viz následující ukázka kódu. Zde je uživateli zobrazován výběr možností a dle jeho vstupu realizována zvolená funkcionalita.

```
while (!exit)
{
DisplayMenu();
switch (Console.ReadLine())
{
      case "1":
            Console.Clear();
            ShowBalance();
            Continue();
            break;
      case "2":
            Console.Clear();
            Deposit();
            Continue();
            break;
      case "3":
            Console.Clear();
            TransactionCreate();
            Continue();
            break;
      case "4":
            Console.Clear();
            Transactions();
            break;
      case "5":
            exit = true;
            break;
      }
}
```

Uživatel má k dispozici funkce jako Zobrazit zůstatek na účtu, Vložit finanční prostředky na účet, Zadat příkaz k úhradě a Zobrazení historie všech transakcí na účtu.
Vzhledem k tomu, že je systém navržen tak, aby každý uživatel mohl vlastnit více bankovních účtů, je nutno doplnit možnost výběru aktivního účtu u některých funkcí.

Metoda pro nastavení účtu je znázorněna v následující ukázce:

```
var requestParameters = string.Format(RequestStrings.ACCOUNT_BY_USERNAME,
UserName, AccessToken);
var jsonResponse = GetResponse(requestParameters, RequestMethods.GET);

if (jsonResponse!= null)
{
    var accounts = JsonConvert.DeserializeObject<List<Entities.Account>>(json-
Response);
    if (BankAccount == null)
    {
        if (accounts.Count > 1)
        {
            Console.WriteLine("Vyberte jeden z vasich uctu:\n");
            Console.WriteLine("#  Cislo uctu    | Zustatek");
            for (int i = 0; i < accounts.Count; i++)
            {
                  Console.WriteLine($"{i + 1}) {accounts[i].AccountNumber} |
                  {accounts[i].Balance}");
            }
        }

        int index = 0;
        while (index < 1 || index > accounts.Count)
        {
            Console.Write("Zadej index: ");
            int.TryParse(Console.ReadLine(), out index);
        }
        BankAccount = accounts[index - 1];
    } else
    {
        BankAccount = accounts
                    .Where(a => a.AccountNumber == BankAccount.AccountNumber)
                    .SingleOrDefault();
    }
    Console.Clear();
}
```

Na této ukázce je zobrazen celý průběh nastavení účtu od samotného dotazu na web API, přes deserializaci na model (objekt) aplikace, až po zobrazení a ošetření interakce uživatele při výběru ze seznamu dostupných účtů. Na této ukázce je zobrazena metoda, která se stará o několik na sobě nezávislých úkonů a pro lepší čitelnost by bylo vhodné kód rozčlenit na menší části. Takové řešení bude předvedeno v další verzi klient aplikace v následujícím úkolu.

4.5 Úkol 5 – Konzolová aplikace .NET Core

Realizací tohoto úkolu vzniká nová aplikace, implementující vestavěný IoC kontejner frameworku .NET Core. V hlavní metodě programu se vytváří třída DependencyConfiguration, která ve své metodě SetUp zajišťuje sestavení třídy ServiceProvider – kontejneru, který se stará o vkládání závislostí.

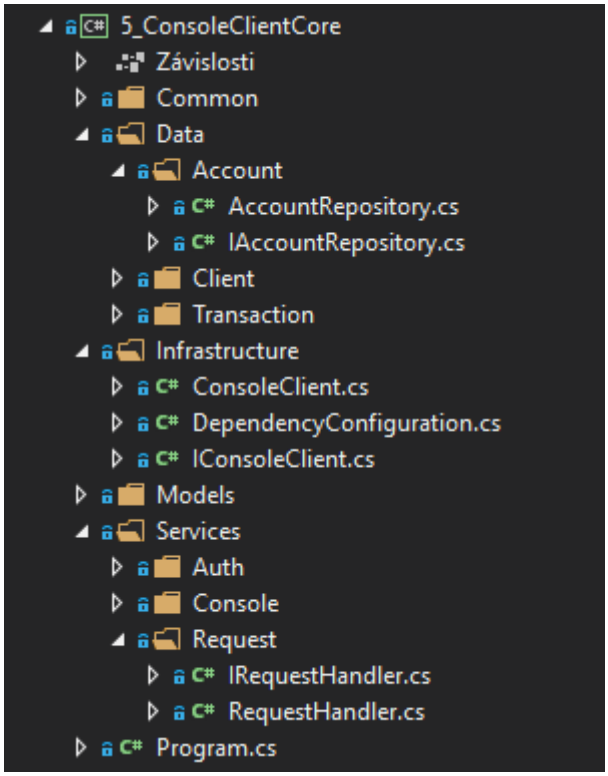
```
static void Main(string[] args)
{
	var configuration = DependencyConfiguration.SetUp();
	configuration.GetService<IConsoleClient>().Run().Wait();
}
```

Samotné nastavení závislostí pak vypadá následovně:

```
public static IServiceProvider SetUp()
{
	var serviceProvider = new ServiceCollection()
		.AddTransient<IConsoleClient, ConsoleClient>()
		.AddTransient<IAuthService, AuthService>()
		.AddTransient<IConsoleManager, ConsoleManager>()
		.AddTransient<ITransactionRepository, TransactionRepository>()
		.AddTransient<IAccountRepository, AccountRepository>()
		.AddTransient<IClientRepository, ClientRepository>()
		.AddTransient<IRequestHandler, RequestHandler>()
		.BuildServiceProvider();
	return serviceProvider;
}
```

Metoda *AddTransient* přidává do kolekce služeb nové služby definované pomocí jejich náležitých rozhraní. Dalšími možnostmi deklarování služeb do kolekce je *AddSingleton*, a *AddScoped*. Rozdíl těchto tří metod spočívá v životnosti vzniklých instancí. Pro verzi singleton, může vzniknout současně vždy pouze jediná instance objektu, verze scoped pak ohraničuje životnost pouze pro definovaný scope. Pro toto řešení výběr životnosti nehraje velkou roli, a tak je volená verze transient, umožňující vznik více instancí objektu, často volená pro nestavové služby.

Projekt je členěn do několika částí, viz obrázek 16. Služby, starající se o získávání dat, jsou umístěny ve složce **data**, kde jsou dále rozděleny dle typu objektů. Složka **infrastructure** obsahuje třídu určenou k sestavení IoC kontejneru a obslužný kód programu. Ve složce **models** jsou umístěny všechny modely využívané aplikací. Na závěr složka **services** obsahuje služby pro komunikaci se serverem a práci s konzolí.



Obrázek 16 – Struktura aplikace

Jak bylo ukázáno v první ukázce, v hlavní metodě programu main se sestaví služba ConsoleClient, jejíž konstruktor je na následující ukázce:

```
public ConsoleClient(IAuthService authService,
                     ITransactionRepository transactionRepository,
                     IAccountRepository accountRepository,
                     IClientRepository clientRepository,
                     IConsoleManager consoleManager)
{
    _auth = authService;
    _transactionRepository = transactionRepository;
    _accountRepository = accountRepository;
    _clientRepository = clientRepository;
    _consoleManager = consoleManager;
    exit = false;
    choice = 0;
}
```

Dochází zde pomocí constructor injection k inicializaci všech potřebných služeb a nastavení výchozích hodnot vlastností. První spouštěnou metodou je pak asynchronní metoda *Run*, zajišťující běh programu.

```
public async Task Run()
{
    await _auth.Login();

    await SetClientDetail();

    await MainProgram();
}
```

Vzhledem k označení metody *async* je možné využívat klíčové slovo *await*, zajišťující čekání programu na vykonání asynchronních metod. Tato metoda zajistí externí přihlášení uživatele do systému, uložení jeho údajů do programu k dalšímu užití a spuštění hlavního programu.

V metodě MainProgram poté probíhá pomocí připojených služeb obdobná funkcionalita, jako tomu bylo v úkolu 4, kód je však psán čitelněji a celý program je svou strukturou lépe testovatelný.

```
private async Task MainProgram()
{
    _consoleManager.Continue();
    while (!exit)
    {
        await SetClientDetail();
        _consoleManager.DisplayMainMenu();
        do choice = _consoleManager.GetInt(); while (choice == null);

        switch (choice)
        {
            case 1:
                _consoleManager.DisplayAccounts();
                _consoleManager.Continue();
                break;
            case 2:
                var account = _consoleManager.ReadAccount();
                await _accountRepository.AccountUpdate(account);
                _consoleManager.Continue(true);
                break;
```

```
            case 3:
                var transaction = _consoleManager.ReadTransaction();
                _transactionRepository.TransactionCreate(transaction);
                _consoleManager.Continue(true);
                break;
            case 4:
                await TransactionMenu();
                break;
            case 5:
                var accounts = await _accountRepository.GetPublicAccounts();
                _consoleManager.DisplayAccounts(accounts);
                _consoleManager.Continue();
                break;
            case 6:
                await _accountRepository.AccountCreate();
                _consoleManager.Continue(true);
                break;
            case 7:
                exit = true;
                break;
        }
    }
}
```

Příkladem volané metody služby TransactionRepository je vytváření nové transakce, která je na následující ukázce kódu.

```
public void TransactionCreate(Models.Transaction transaction)
{
    var request = string.Format(Common.RequestEndpoints.TRANSACTIONS,
                    Auth.CurrentUser.UserName, Auth.CurrentUser.AccessToken);
    var json = JsonConvert.SerializeObject(transaction, Formatting.Indented);

    _requestHandler.GetResponse(request, Common.RequestMethods.POST, json);
}
```

Model transakce je předán jako parametr metody a serializován do formátu json k přenosu. K endpointu serveru, využívanému k vytváření transakcí, je připojeno uživatelské jméno a access token k autorizaci, a je odeslán POST požadavek na server. Obdobně fungují ostatní funkce této klient aplikace.

4.6 Úkol 6 – Desktopová aplikace Windows Forms

Tato aplikace částečně zrcadlí svou strukturou předcházející konzolovou aplikaci. Dochází zde však k zásadnímu rozdílu, a tím je implementace architektury MVP (viz kapitola 2.1). Na nadcházející ukázce si můžeme prohlédnout kód spuštěný po startu programu. Nejprve

je inicializováno view přihlašovací obrazovky a autentizační služba. Jejich reference jsou dále předány jako parametry do konstruktoru presenteru, obsluhujícího přihlašovací formulář.

```
static void Main()
{
	Application.EnableVisualStyles();
	Application.SetCompatibleTextRenderingDefault(false);

	var view = new LoginForm();
	var authService = new AuthService();
	new LoginPresenter(view, authService);
	view.Show();

	Application.Run();
}
```

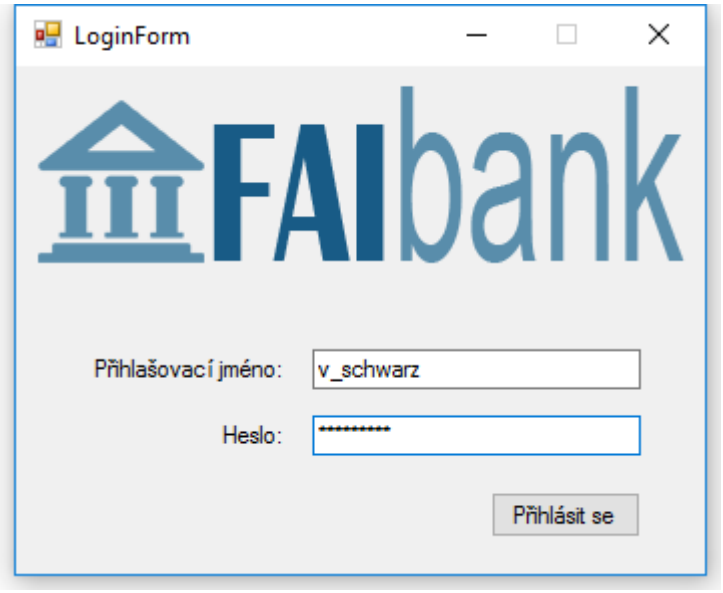
Třída LoginPresenter je zobrazena v následující ukázce.

```
public class LoginPresenter
{
	private readonly ILoginForm _view;
	private readonly IAuthService _authService;
	private User _user;

	public LoginPresenter(ILoginForm view, IAuthService service)
	{
		_authService = service;
		_view = view;
		_view.Presenter = this;
	}

	public async Task<bool> Login()
	{
		_user = await _authService.Login(_view.Credentials);
		if (_user == null)
		{
			_view.ErrorLoggingIn();
			return false;
		}
		else
		{
			var mainView = new MainForm();
			var mainPresenter = new MainPresenter(mainView, _user);

			mainView.Show();
			this._view.Close();
			return true;
		}
	}
};
```

V konstruktoru je presenteru poskytnuto rozhraní view a autentizační služby, a view je rovněž předána reference samotného presenteru. Tento presenter obsahuje jedinou metodu, a tou je asynchronní metoda Login, zajišťující přihlášení uživatele. Z view jsou získány přihlašovací údaje a odeslány na server, zajišťující externí přihlášení do systému. Tato služba zůstává stejná, jako v předchozích řešeních. Při nevalidním pokusu o přihlášení je uživatel upozorněn hláškou, v opačném případě je vytvořena instance nového formuláře – MainForm, a jeho presenteru. Metoda obsluhující stisk tlačítko pro přihlášení ve view (viz následující ukázka) poté zajišťuje správné zobrazení rozhraní uživateli.

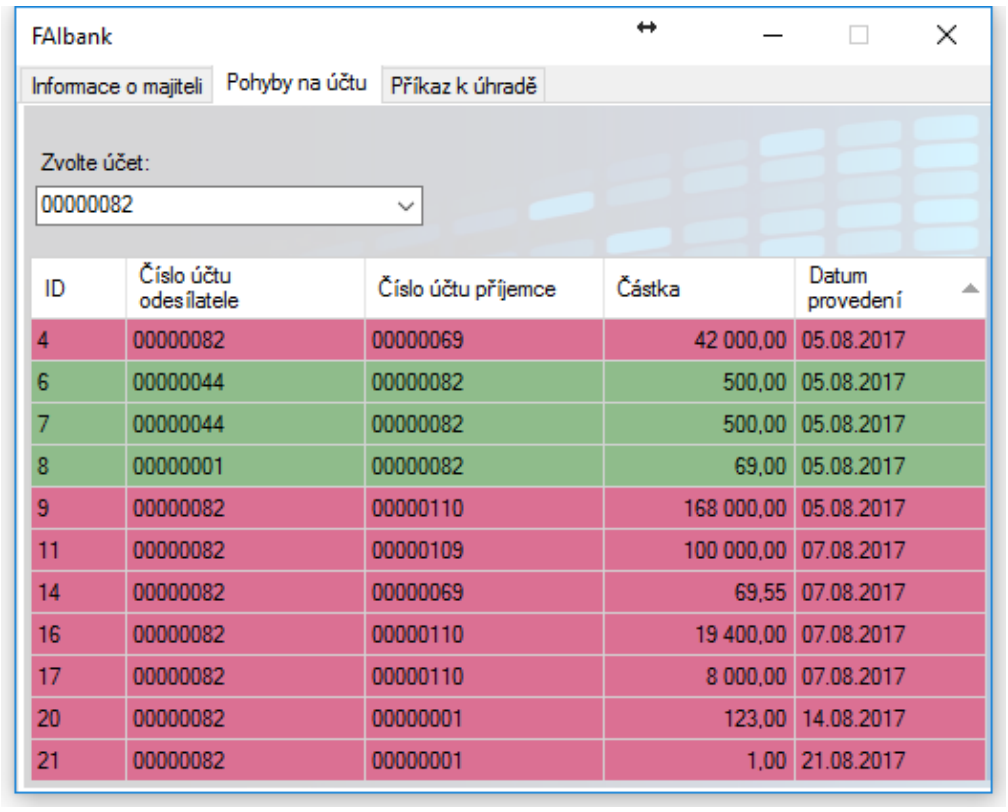
```
private async void btn_login_Click(object sender, EventArgs e)
{
       this.Enabled = false;
       if (!await Presenter.Login()) this.Enabled = true;
}

public void ErrorLoggingIn()
{
       MessageBox.Show("Zkontrolujte jméno a heslo.",
                       "Chyba přihlašování",
                       MessageBoxButtons.OK);
}
```

Celý proces přihlašování tedy začíná po stisku tlačítka *Přihlásit* uživatelem. Nejprve je deaktivováno tlačítko *Přihlásit*, pro ošetření duplikovaného volání funkce a následně očekávána metoda presenteru, při jejímž negativním výsledku je tlačítko opět obnoveno a presenterem je spuštěna metoda zobrazující chybovou hlášku *ErrorLoggingIn.* V případě úspěšného přihlášení směřuje presenter do nového view a současné je zavřeno. Ukázka formuláře pro přihlášení je zobrazena na obrázku 17. Okno obsahuje komponenty TextBox pro zadání přihlašovacích údajů, PictureBox obsahující logo a Button – tlačítko pro spuštění procesu přihlašování.



Obrázek 17 – Přihlašovací formulář aplikace zhotovené pomocí technologie Windows Forms

Hlavní okno aplikace obsahuje záložky pro jednotlivé funkce, realizované pomocí komponenty TabControl. Dále je využita komponenta DataGridView pro zobrazení tabulky a ComboBox pro přepínání mezi dostupnými bankovními účty. Realizace tohoto view je ke zhlédnutí na obrázku 18.



ID	Číslo účtu odesílatele	Číslo účtu příjemce	Částka	Datum provedení
4	00000082	00000069	42 000,00	05.08.2017
6	00000044	00000082	500,00	05.08.2017
7	00000044	00000082	500,00	05.08.2017
8	00000001	00000082	69,00	05.08.2017
9	00000082	00000110	168 000,00	05.08.2017
11	00000082	00000109	100 000,00	07.08.2017
14	00000082	00000069	69,55	07.08.2017
16	00000082	00000110	19 400,00	07.08.2017
17	00000082	00000110	8 000,00	07.08.2017
20	00000082	00000001	123,00	14.08.2017
21	00000082	00000001	1,00	21.08.2017

Obrázek 18 – Záložka pohyby na účtu

Plnění tabulky nesoucí záznamy o transakcích na vybraném účtu zajišťuje metoda view volaná z presenteru, viz následující ukázka. Celá tabulka je po naplnění procházena foreach cyklem a řádkům jsou nastaveny barvy, odlišující příchozí a odchozí platby.

```
public void SetTransactionHistory(List<Transaction> transactions)
{
      foreach (var item in transactions)
      {
            dataGridView_Transactions.Rows.Add(item.TransactionID,
                                               item.AccountNo,
                                               item.RecipientNo,
                                               item.Amount,
                                               item.DateCreated);
      }

      foreach (DataGridViewRow row in dataGridView_Transactions.Rows)
            if (cbHistoryAccountNumber.SelectedItem.ToString() == row.Cells[2]
                                                                  .Value.ToString())
                  row.DefaultCellStyle.BackColor = Color.DarkSeaGreen;
            else
                  row.DefaultCellStyle.BackColor = Color.PaleVioletRed;
}
```

4.7 Úkol 7 – Multiplatformní aplikace Xamarin.Forms

Další verzí klienta bankovního serveru je multiplatformní aplikace zhotovená za pomoci frameworku Xamarin.Forms. Tato varianta implementuje architekturu MVVM, využívanou u aplikací založených na jazyku XAML.

Aplikace opět využívá služby komunikace se serverem z předchozích řešení a poskytuje i stejnou funkcionalitu. Po startu aplikace tedy poskytuje uživateli možnost přihlášení do systému přes portál IS/STAG. Stránka s přihlašovacím formulářem, zobrazeným na skutečném android zařízení, je na obrázku 19.

Obrázek 19 – přihlašovací formulář, zobrazený na zařízení Android

V konstruktoru tohoto view (stránky) se nastaví data binding.

```
public LoginView()
{
	InitializeComponent();
	BindingContext = new LoginViewModel();
}
```

Pomocí této deklarace je možné v XAML souboru přihlašovací stránky zobrazovat data viewmodelu. Následuje ukázka, demonstrující data binding pro tlačítko přihlášení a indikátor načítání.

```
<Button
      Command="{Binding SubmitCommand}"
      Text="Přihlásit se"
      TextColor="White"
      FontAttributes="Bold"
      FontSize="20"
      HorizontalOptions="FillAndExpand"
      BackgroundColor="#2674a5" />

<ActivityIndicator IsRunning="{Binding IsBusy}" Color="DodgerBlue" />
```

Ve viewmodelu je pro navázaný Command vytvořená metoda, obsluhující proces přihlášení a hodnota uložená v proměnné *IsBusy* obstarává zobrazení indikátoru činnosti.

```
public async void OnSubmit()
{
      IsBusy = true;
      var loginInfo = new LoginModel() { UserName = userName,
                                         Password = password };
      _user = await _authService.Login(loginInfo);
      IsBusy = false;
      if (_user != null)
      {
            NavigationMessage.PutData("user", _user);

            App.Current.MainPage = new NavigationPage(new Views.MainView());
      }
}
```

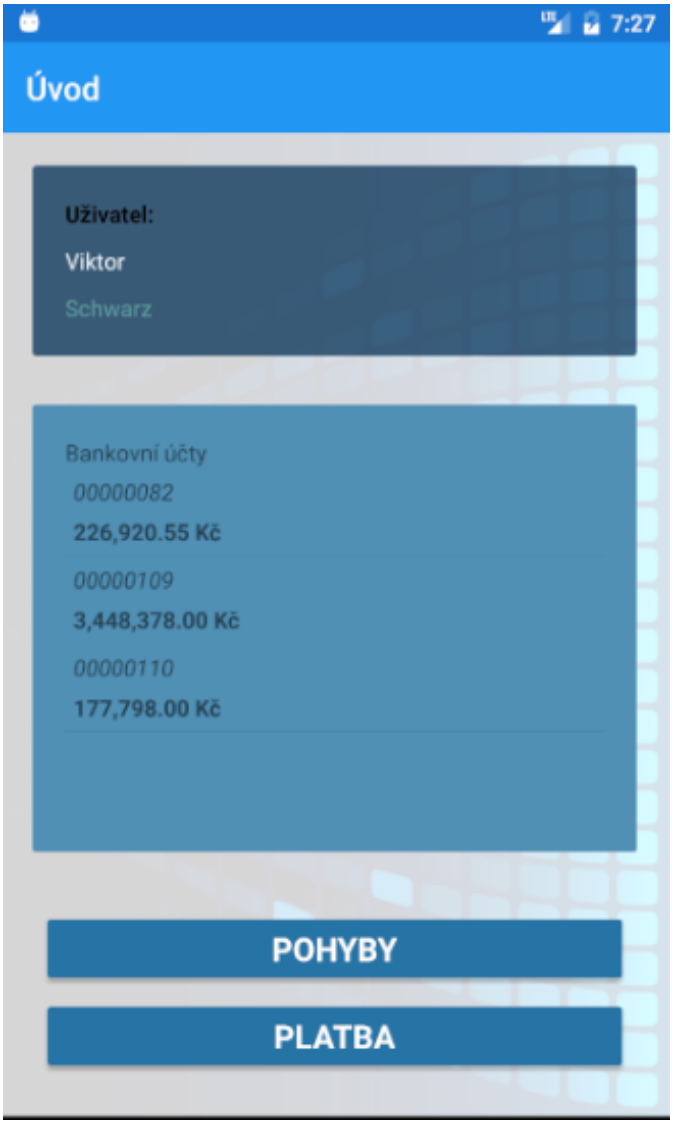
Zajištění správného zobrazení dat aktualizovaného viewmodelu je dosaženo implementací rozhraní *INotifyPropertyChanged.* Každá změna vlastnosti modelu pak vyvolá notifikaci, oznamující příslušnou událost a view se náležitě aktualizuje. V následující ukázce kódu je příklad deklarace proměnné, vyvolávající notifikaci o změně a volaná metoda.

```
private bool isBusy;
public bool IsBusy
{
	get
	{
		return isBusy;
	}
	set
	{
		isBusy = value;
		OnPropertyChanged();
	}
}

protected virtual void OnPropertyChanged([CallerMemberName]
                                         string propertyName = null)
{
	PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(
                                                    propertyName));
}
```

Při úspěšném přihlášení jsou informace o přihlášeném uživateli uloženy do statické paměti programu a je zobrazena nová stránka jako *NavigationPage*. Pomocí této třídy je možné realizovat navigaci mezi nově přidanými stránkami.

Úvodní stránka obsahuje informace o přihlášeném uživateli a jeho bankovních účtech. Její zobrazení na zařízení Android je k vidění na obrázku 20. Seznam účtů je realizován pomocí komponenty *ListView*, viz následující ukázka.

```
<ListView x:Name="listAccounts" ItemsSource="{Binding ClientAccounts}"
                                        HasUnevenRows="True">
  <ListView.ItemTemplate>
    <DataTemplate>
      <ViewCell>
        <StackLayout Padding="5" Orientation="Vertical">
          <Label Text="{Binding AccountNumber}" />
          <Label Text="{Binding Balance, StringFormat='{0:N} Kč'}" />
        </StackLayout>
      </ViewCell>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```



Obrázek 20 – Úvodní stránka aplikace zobrazená na zařízení Android

Tlačítka pro přepínání oken jednotlivých funkcionalit aplikace přidají do fronty navigace náležité view.

```
private void btn_pay_Clicked(object sender, EventArgs e)
{
      Navigation.PushAsync(new PaymentView());
}

private void btn_history_Clicked(object sender, EventArgs e)
{
      Navigation.PushAsync(new TransactionsView());
}
```

4.8 Úkol 8 – RESTful Web API s implementací ADO.NET

První verzí web API v roli bankovního serveru je ručně vytvořena databáze, jejíž struktura je na obrázku 6. Projekt je založen jako Web API – Webová aplikace ASP.NET Core. Třída StartUp.cs, zajišťující spuštění aplikace, byla doplněna o konfiguraci služeb, které je následně možné získávat pomocí constructor inejction.

```
public void ConfigureServices(IServiceCollection services)
{
      services.AddTransient<IAccountService, AccountService>();
      services.AddTransient<IAccountRepository, AccountRepository>();

      services.AddTransient<ITransactionService, TransactionService>();
      services.AddTransient<ITransactionRepository, TransactionRepository>();

      services.AddTransient<IClientService, ClientService>();
      services.AddTransient<IClientRepository, ClientRepository>();

      services.AddTransient<ISignInMannager, SignInMannager>();

      services.AddTransient<IDbConnectionProvider, DbConnectionProvider>();

      services.AddMvc();
}
```

Aplikace je strukturována tak, že kontroler přijme požadavek a zavolá metodu náležité služby (service). Tato služba využívá jinou třídu (repository), která se postará o získání požadovaných dat z databáze a vrátí je zpět službě. Zde dále probíhá obchodní logika nad získanými daty a výsledek je vrácen kontroleru. Odtud je informace dále zpracována a předána klientovi formou odpovědi. V následujících ukázkách bude předveden průběh požadavku na transakce na účtu.

```
public TransactionsController(ITransactionService transactionService,
                              ISignInMannager signInMannager)
{
      _transactionService = transactionService;
      _signInMannager = signInMannager;
}
```

Kontroleru transakcí je v jeho konstruktoru poskytnuta služba *TransactionService* a správce přihlášení *SignInMannager*.

```
[HttpGet]
[Route("transactionsByAccountID/{accountId}")]
public IActionResult GetAccountsTransactions(int accountId,
                                             [FromQuery()] string userName,
                                             [FromQuery()] string accessToken)
{
  if (_signInMannager.Authorize(userName, accessToken))
  {
    return new ObjectResult(_transactionService
                                .GetTransactionsOnAccount(accountId));
  }
  return Unauthorized();
}
```

Pro získání transakcí na účtu je vytvořen endpoint */transactionsByAccountID/{accountId}*, pomocí atributu Route v úvodu metody. Atribut *HttpGet* určuje metodu požadavku, která tuto funkci při přístupu na tento endpoint vyvolá. Z parametrů předaných v URL jsou získány údaje uživatele a zkontrolovány s databází. Při korektním vstupu je dále volána metoda pro vyhledání transakcí na účtu služby *TransactionService.* V opačném případě je klientovi odeslána odpověď v podobně stavového kódu „neověřeno“.

```
public string GetTransactionsOnAccount(int accountID)
{
  var result = _transactionRepository.GetTransactionsOnAccount(accountID);
  return JsonConvert.SerializeObject(result, Formatting.Indented);
}
```

Tato služba volá metodu datové vrstvy *TransactionRepository*, vracející seznam transakcí. Obdržený list objektů je následně serializován do formátu json a vrácen do kontroleru.

```
public List<DTOs.TransactionDTO> GetTransactionsOnAccount(int accountID)
{
 var queryCondition = String.Format(
                  " WHERE t.[AccountID] = {0} OR t.[RecipientID] = {1}",
                  accountID,
                  accountID);
 return Transactions(queryCondition);
}
```

Výše zobrazená metoda vytvoří podmínku v SQL jazyce a předá ji obecné metodě pro hledání transakcí, viz následující ukázka kódu.

```
private List<DTOs.TransactionDTO> Transactions(string query)
{
      List<DTOs.TransactionDTO> transactions =
                                          new List<DTOs.TransactionDTO>();

      using (var con = _dbConectionProvider.GetConnection())
      {
            SqlCommand cmd = new SqlCommand()
            {
                  CommandText = string.Concat("SELECT t.[TransactionID],
                  t.[AccountID],
                  c.[FirstName] + c.[LastName] as 'Recipient',
                  t.[DateCreated],t.[Amount],
                  a.AccountNumber AS 'RecipientNo',
                  t.[RecipientID],
                  aa.[AccountNumber] AS 'AccountNo'
                  FROM [Transactions] t
                  JOIN [Accounts] a ON a.[AccountID] = t.[RecipientID]
                  JOIN [Accounts] aa ON aa.[AccountID] = t.[AccountID]
                  JOIN [Clients] c ON c.[ClientID] = a.[ClientID]",
                  query),
                  CommandType = CommandType.Text,
                  Connection = con
            };
            con.Open();
            using (SqlDataReader reader = cmd.ExecuteReader())
            {
                  while (reader.Read())
                  {
                        transactions.Add(new DTOs.TransactionDTO()
                        {
                              TransactionID = reader.GetInt32(0),
                              AccountID = reader.GetInt32(1),
                              Recipient = reader.GetString(2),
                              DateCreated = reader.GetDateTime(3),
                              Amount = reader.GetDecimal(4),
                              RecipientNo = reader.GetString(5),
                              RecipientID = reader.GetInt32(6),
                              AccountNo = reader.GetString(7)
                        });
                  }
            }
      }

      return transactions;
}
```

V této metodě je tvořen klasický SQL dotaz, doplněn o podmínku výběru, poskytnutou jako parametr. Odpověď SQL serveru je čtena pomocí třídy SqlDataReader a ručně pak mapována na objekty.

4.9 Úkol 9 – RESTful Web API s využitím ORM Entity Framework

Třída StartUp.cs se u této varianty trochu liší oproti předchozímu řešení. Služby, starající se o obchodní logiku, se nachází i v tomto řešení, ale datová vrstva, dříve realizována pomocí tříd „repository“, je nyní nahrazena jednou třídou – *ApplicationDbContext*. Na následující ukázce je zobrazena definice služeb v IoC kontejneru.

```
services.AddTransient<IAccountService, AccountService>();
services.AddTransient<ITransactionService, TransactionService>();

services.AddTransient<IClientService, ClientService>();
services.AddTransient<IUserManager, UserManager>();
services.AddTransient<ISeedingService, SeedingService>();
services.AddMvc();
services.AddDbContext<ApplicationDbContext>(options
                                =>options.UseSqlServer(connectionString));
```

ApplicationDbContext vychází z třídy DbContext Entity Frameworku a obsahuje vlastnosti DbSet pro všechny typy, pro které bude generována tabulka v databázi.

Jednotlivé modely jsou dále označeny patřičnými atributy, pro zajištění vygenerování požadovaných relací a vlastností tabulek.

```
public class ApplicationDbContext : DbContext
{
      public ApplicationDbContext(DbContextOptions options) : base (options){}

      public DbSet<Account> Accounts { get; set; }
      public DbSet<Transaction> Transactions { get; set; }
      public DbSet<Client> Clients { get; set; }
      public DbSet<ClientRole> Roles { get; set; }
      public DbSet<AppLogEntry> AppLog { get; set; }
}
```

```
public class Account
{
	[Key]
	[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
	public int AccountID { get; set; }

	public Client Client { get; set; }
	public bool IsPrimary { get; set; }

	[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
	[Required]
	public string AccountNumber { get; set; }
	[Required]
	public decimal Balance { get; set; }

	[InverseProperty("Recipient")]
	[JsonIgnore]
	public ICollection<Transaction> IncomingTransactions { get; set; }

	[InverseProperty("Account")]
	[JsonIgnore]
	public ICollection<Transaction> OutgoingTransactions { get; set; }
}
```

V uvedeném příkladu modelu bankovního účtu je využito atributů označujících vlastnost *AccountID* jako primární klíč a také identity sloupec. Vlastnost *AccountNumber* je označena jako vypočítaná hodnota při vkládání do databáze (tvoří se pomocí ID záznamu) a také je označena atributem *Required* pro validaci modelu. Kolekce příchozích a odchozích transakcí zde slouží k realizaci M:N relace, a k tomu sloužící atribut *InverseProperty*. Kolekce jsou dále doplněny o atribut JsonIgnore, neboť je nežádoucí, aby byly serializovány.

Ve chvíli, kdy je v kódu vše připraveno, dochází ke generování databáze pomocí příkazů konzole správce balíčků. Nejprve příkaz `add-migration [název]` připraví skript pro generování, popřípadě úpravu databáze, a následně `update-database` pro spuštění připraveného skriptu nad definovanou databází.

Práce s kontextem je zobrazena na následující ukázce metody, vracející bankovní účty podle uživatelského jména klienta.

```
public string GetAccountsByUserName(string userName)
{
	try
	{
		return JsonConvert.SerializeObject(_db.Accounts
					.Include(c => c.Client)
					.Where(c => c.Client.UserName == userName)
					.ToList(), Formatting.Indented);
	}
	catch (Exception e)
	{
		_db.Add(new AppLogEntry
		{
			CreatedBy = userName,
			Type = Common.AppLogMessageTypes.ERROR,
			Message = e.Message,
			StackTrace = e.StackTrace
		});
		_db.SaveChanges();
		return null;
	}
}
```

V této metodě jsou z tabulky účtů vybrány ty záznamy, které na základě propojení s tabulkou klientů patří zadanému uživateli. K propojení tabulek na základě cizího klíče slouží metoda *Include* u kolekce kontextu a pomocí LINQ výrazů lze pak nalezené prvky dále filtrovat.

Při jakékoliv chybě je rovněž v programu připraven zápis do logu v databázi a provedené změny uloženy pomocí metody *SaveChanges*.

4.10 Úkol 10 – Implementace Swagger do Web API

Nedílnou součástí tvorby web API je poskytování kvalitní dokumentace popisující nabízené funkce a zacházení s nimi. Jednoduchou možností dosažení tohoto cíle je implementace frameworku Swagger instalací NuGet balíčku *Swashbuckle.AspNetCore*.

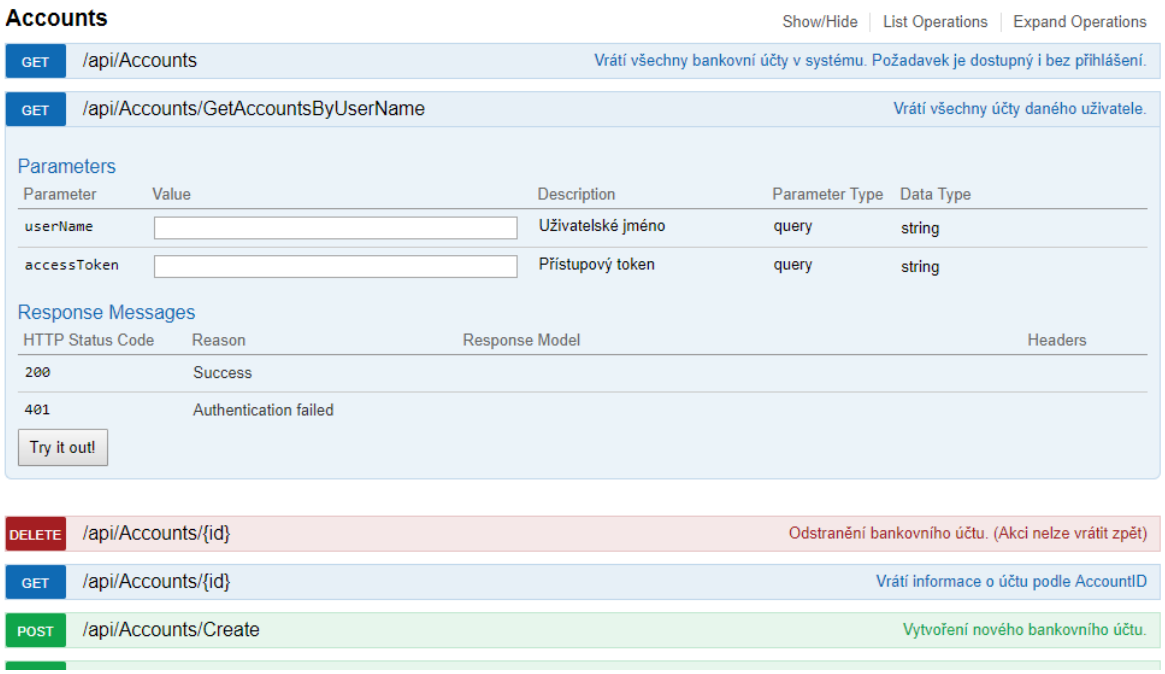
Tento framework využívá dokumentačních komentářů (summary), vygenerovaných v XML souboru. Toto generování je nejprve potřeba nastavit ve vlastnostech projektu. Přesněji pak v sekci „sestavení“, položka „soubor dokumentace XML“. Cestu k tomuto souboru je nutné doplnit v konfiguraci služeb a v konfiguraci aplikace dle následující ukázky:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSwaggerGen(c =>
  {
      c.SwaggerDoc("v1", new Info { Title = Common.SwaggerInfo.TITLE,
                                    Version = "v1",
                                    Description = Common.SwaggerInfo.DESCRIPTION,
      });
      c.IncludeXmlComments(string.Format(@"{0}\\BankAPI_ORM.xml",
                  PlatformServices.Default.Application.ApplicationBasePath));
  });

...



public void Configure(IApplicationBuilder app,
                      IHostingEnvironment env,
                      ILoggerFactory loggerFactory)
{
 app.UseSwagger();
 app.UseSwaggerUI(c =>
      {
            c.SwaggerEndpoint("/swagger/v1/swagger.json", "FaiBank V1");
      });

...
```

Popis pomocí dokumentačního komentáře pak může vypadat následovně:

```
/// <summary>
/// Odstranění bankovního účtu. (Akci nelze vrátit zpět)
/// </summary>
/// <param name="id">ID odebíraného účtu</param>
/// <param name="userName">Přihlašovací jméno</param>
/// <param name="accessToken">Přístupový token</param>
/// <response code="200">OK</response>
/// <response code="401">Authentication failed</response>
/// <response code="404">Not found</response>
/// <returns></returns>
```

Ukázka výsledné vygenerované dokumentace je na obrázku 21.



Obrázek 21 – Vygenerovaná dokumentace pomocí frameworku Swagger

4.11 Úkol 11 – Implementace jednotkových testů do Web API

V rámci tohoto úkolu byl vytvořen nový projekt testu jednotek s připojeným odkazem závislosti na web API z úkolu 10. Pro jednotlivé kontrolery byly sepsány jednotkové testy, zajišťující jejich správnou funkčnost.

```
[TestMethod]
public void GetAllAccounts()
{
    // arrange
    SetUp();
    var controller = new AccountsController(_userManager, _accountService);
    var mockService = _accountService as MockServices.AccountService;
    var count = mockService.Accounts.Count;

    // act
    var result = controller.Get();
    var okResult = result as OkObjectResult;
    var accounts = JsonConvert.DeserializeObject<List<Account>>(okResult
                                                        .Value.ToString());

    // assert
    Assert.IsNotNull(okResult);
    Assert.AreEqual(count, accounts.Count);
    Assert.AreEqual(200, okResult.StatusCode);
}



private void SetUp()
{
    _userManager = new MockServices.UserManager();
    _accountService = new MockServices.AccountService();
}
```

Na ukázce je test metody *GetAllAccounts*, která vrací informaci o všech účtech v databázi. Metoda testu je rozdělena do tří částí, viz sekce 2.2.4. Nejprve je volána metoda *SetUp*, inicializující závislé služby pomocí zástupných implementací, určených jen pro testování. Následně je vytvořena instance testovaného kontroleru a připraven počet účtů, které by služba měla vrátit.

Dalším krokem je provedení testované metody – dotaz na vrácení všech účtů. Výsledná odpověď je deserializována a připravena ke kontrole výsledků testu v poslední fázi.

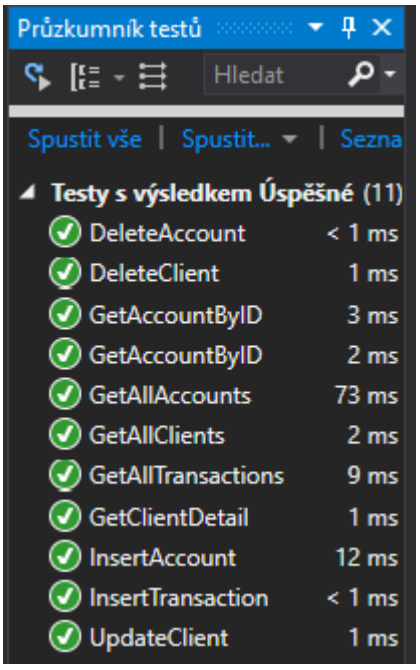
Nejdříve se testuje odpověď pro nenulovou hodnotu, neboť i při chybném požadavku musí web API zaslat odpověď informující o vyskytlé chybě. Dále se porovnává počet deserializovaných objektů s počtem stanovených v první fázi, a nakonec se kontroluje stavový kód odpovědi, který musí souhlasit s výsledkem požadavku.

Podobně budou vypadat testy pro ostatní metody i kontrolery.

Závislosti testovaných jednotek, které nejsou právě terčem testu, jsou nahrazeny zástupnými třídami, jako je tomu na následující ukázce. Jedná se o implementaci rozhraní, jejíž logika není v dané chvíli relevantní a potřebujeme pouze obstarat závislý běh programu.

```
public class UserManager : IUserManager
{
	public bool Authorize(string userName, string accessToken,
					string restrictedRole = null)
	{
		return true;
	}

	public void Login(User user)
	{
		return;
	}
}
```

Po sestavení testů lze v okně *Průzkumník testů* vývojového prostředí Visual Studio všechny testy spouštět a pravidelně tak kontrolovat stávající funkcionalitu aplikace i po dalších úpravách.



Obrázek 22 – Průzkumník testů

ZÁVĚR

V této práci byly navrženy úkoly pro výuku tvorby klient-server aplikací za využití moderních postupů a osvědčených technik. Předpokládají se základní vstupní znalosti programování, přesněji pak základy jazyka C# a objektově orientovaného přístupu k programování. Aplikace v navržených zadáních postupně nabývají na komplexnosti. Studentům je nejprve představeno jednodušší řešení, následně jsou doplněny vhodnější postupy a praktiky.

Pro tvorbu klienta v podobě konzolové aplikace vznikly 3 úkoly, které nejprve studenta seznámí s technologií využitou k realizaci veškeré zadané funkcionality a následně je student vyzván k vývoji dalších dvou verzí plně funkční konzolové aplikace. První verzí je běžná aplikace .NET Frameworku, splňující všechny zadané požadavky, avšak není kladen velký důraz na testovatelnost a elegantnost kódu. V druhé verzi vzniká konzolová aplikace .NET Core, implementující vkládání závislostí (Dependency Injection) a strukturování kódu do menších, specializovaných modulů pro zvýšení čitelnosti a testovatelnosti softwaru. Toto řešení přináší několik výhod za cenu vyšší náročnosti při programování. Jednou z výhod je snadnější implementace nové funkcionality, snadnější čitelnost kódu a možnost méně pracného zavedení jednotkových testů do projektu.

Další aplikací na seznamu navržených úkolů je aplikace, zhotovená za pomoci technologie Windows Forms. Ač je v dnešní době tato technologie méně používaná a setkáváme se s ní spíše u již hotových projektů než u nově vznikajících, jedná se o dobrý způsob seznámení studenta s architekturou MVP.

Poslední variantou aplikace, vystupující jako klient bankovního serveru, je multiplatformní aplikace zhotovená za použití frameworku Xamarin.Forms. Zde se studenti obeznámí s tvorbou aplikace založenou na jazyku XAML. U desktopové aplikace tento jazyk využívá například Windows Presentation Foundation. Pro tyto aplikace se nabízí vývoj dodržující pravidla architektury MVVM, jehož aplikování do projektu je součást zadání daného úkolu. Stavění aplikací v tomto frameworku má výhodu v úspoře času, neboť je současně vyvíjena verze pro více platforem současně, avšak nese i nevýhody v podobě méně snadné práce s vizualizací uživatelského prostředí. Pokud však není kladen příliš velký důraz na přesně zadaný vzhled UI a větší prioritu má samotná logika v pozadí, potom je tento framework vhodnou volbou.

V prvních sedmi navržených úkolech studenti pracují s web API, simulujícím bankovní server, který je současně výsledkem vzorového řešení, vzniklým jako součást této bakalářské

práce. V další části navržených úkolů budou studenti sami realizovat stejnou službu. Jediným rozdílem bude, že svou aplikaci nebudou vystavovat volně k dispozici na internet a budou využívat pouze lokální databázi.

Pro webové rozhraní API vznikly dva úkoly. V první variantě jsou studenti seznámeni s konceptem samotné tvorby web API, se systémem routování a způsobem práce kontrolerů. Vzniká lokální databáze uchovávající informace o klientech, účtech a transakcích. K získání těchto dat je pak využito dotazování v podobě jazyka SQL za pomoci třídy SqlCommand. Jedná se o velmi jednoduché, základní řešení, s kterým se můžeme setkat i v praxi u méně komplexních projektů. Jedná se však o nevhodné řešení, které není v souladu se současnými doporučenými postupy. Proto je následujícím úkolem implementace ORM frameworku, zajišťujícího objektově-relační mapování. Tento přístup je vhodný především u vzniku nových projektů, které nevyžadují příliš mnoho logiky umístěné v procedurách databáze. Pro potřeby stanoveného zadání se jedná o ideální řešení.

Při tvorbě aplikace, využívající API, může docházet k potížím z důvodu nedostatečně popsaných metod dané služby. Poskytnutí kvalitní a informativní dokumentace je tak důležitou součástí vývoje web API. Z tohoto důvodu je dalším navrženým úkolem zhotovení podrobné dokumentace vzniklé služby za pomoci frameworku Swagger. Tento nástroj umožní připravit nejen přehledný popis jednotlivých funkcí systému, ale rovněž rozhraní pro testování jednotlivých dotazů.

Posledním úkolem je nasazení jednotkových testů webového rozhraní. V tomto úkolu vynikne výhoda využití vkládání závislostí, neboť je možné opravdu testovat nejmenší jednotky systému bez nutnosti deklarace a ošetření závislých metod, či služeb.

V rámci této bakalářské práce byly navrženy úkoly, seznamující studenty s několika, v praxi běžně využívanými, frameworky a technologiemi. K vypracování těchto zadání byla rovněž navržena vhodná architektura a struktura řešení a následně bylo řešení všech úkolů demonstrováno na vzorových řešeních. Doporučené postupy, návrhové vzory a techniky využívané při vypracování byly popsány v teoretické části. Na závěr byly popsány klíčové části programů společně s ukázkou vzniklých aplikací.

SEZNAM POUŽITÉ LITERATURY

[1] BĚHÁLEK, Marek. .NET Framework [online]. Katedra informatiky, VŠB-TU Ostrava, 2007 [cit. 2017-08-20]. Dostupné z: http://www.cs.vsb.cz/behalek/vyuka/pcsharp/text/ch01.html

[2] NAGEL, Christian. Professional C# 6 and .Net Core 1.0. Indianapolis, IN: Wrox, a Wiley brand, published by John Wiley & Sons, 2016. ISBN 111909660X.

[3] CHOWDHURI, Shahed. ASP.NET Core Essentials. 35 Livery Street, Birmingham: UK: Packt Publishing, 2016. ISBN 978-1-78588-915-8.

[4] API design. Microsoft Azure [online]. 2017 [cit. 2017-08-13]. Dostupné z: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

[5] J. PRICE, Mark. C# 6 and .NET Core 1.0: Modern Cross-Platform Development. Birmingham: Packt Publishing, 2016. ISBN 9781785285691.

[6] Create a web API with ASP.NET Core MVC and Visual Studio for Windows. Microsoft [online]. © Microsoft: ANDERSON Rick, Mike WASON, 2017 [cit. 2017-08-11]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?tabs=aspnet1x

[7] Xamarin.Forms: Cross-Platform User Interfaces with Xamarin.Forms [online]. Xamarin, 2017 [cit. 2017-08-09]. Dostupné z: https://developer.xamarin.com/guides/xamarin-forms/

[8] PETZOLD, Charles. Creating Mobile Apps with Xamarin.Forms: Cross-platform C# programming for iOS, Android, and Windows. One Microsoft Way Redmond, Washington 98052-6399: Microsoft Press - A Division of Microsoft Corporation, 2016. ISBN 978-1-5093-0297-0.

[9] Microsoft journey from Classic ASP to ASP.Net Core. Cross Platform Support. LinkedIn [online]. Khuram Shehzad: LinkedIn Corporati-on© 2017, 2016 [cit. 2017-08-04]. Dostupné z: https://www.linkedin.com/pulse/microsoft-journey-from-classic-asp-aspnet-core-10-rc-2-khuram-shehzad

[10] FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003. Addison-Wesley signature series. ISBN 0-321-12742-0.

[11] MARTIN, Robert C. Agile software development: principles, patterns, and practices. Upper Saddle River, N.J.: Prentice Hall, c2003. ISBN 978-0135974445.

[12] Presentation Patterns : MVC, MVP, PM, MVVM. Manoj Jaggavarapu [online]. 2012 [cit. 2017-08-20]. Dostupné z: https://manojjaggavarapu.wordpress.com/2012/05/02/presentation-patterns-mvc-mvp-pm-mvvm/

[13] GAROFALO, Raffaele. Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern. [Online-Ausg.]. Sebastopol, Calif: O'Reilly Media, 2011. ISBN 9780735650923.

[14] Implementing the Model-View-ViewModel Pattern. Microsoft: Developer Network [online]. © 2017 Microsoft [cit. 2017-08-20]. Dostupné z: Dostupné z: https://msdn.microsoft.com/en-us/library/ff798384.aspx

[15] SEEMANN, Mark. Dependency injection in .NET. Shelter Island, NY: Manning publications, 2011. ISBN 978-193-5182-504.

[16] Introduction to Dependency Injection in ASP.NET Core. Microsoft [online]. © 2017 Microsoft [cit. 2017-08-20]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

[17] The Beginner's Guide to Unit Testing: What Is Unit Testing? MCFARLIN Tom. Envato Tuts+ [online]. © 2017 Envato, 2012 [cit. 2017-08-14]. Dostupné z: https://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728

[18] Unit Test Basics. Microsoft Developer Network [online]. Microsoft, ©2017 [cit. 2017-08-14]. Dostupné z: https://msdn.microsoft.com/en-us/library/hh694602.aspx?f=255&MSPPError=-2147217396

[19] Entity Framework Version History. Microsoft: Developer Network [online]. Microsoft, ©2017 [cit. 2017-08-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/jj574253(v=vs.113).aspx

[20] ADO.NET Data Platform. Microsoft: Developer Network [online]. © 2017 Microsoft [cit. 2017-08-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/ex6y04yf(v=vs.90).aspx

[21] Entity Framework Code First Conventions. Microsoft: Developer Network [online]. © 2017 Microsoft [cit. 2017-08-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/jj679962(v=vs.113).aspx

[22] .NET - Powerful Open Source Cross Platform Development [online]. 2017 [cit. 2017-08-09]. Dostupné z: https://www.microsoft.com/net

[23] SINGLETON, James. ASP.NET Core 1.0 High performance. Birmingham: Packt Publishing, 2016. ISBN 9781785881893.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface – rozhraní pro programování aplikací
URI	Uniform Resource Identifier – řetězec znaků pro identifikaci zdroje v síti
URL	Uniform Resource Locator –adresa zdroje (příklad URI)
HTML	Hyper-Text Mark-up Language
json	JavaScript Object Notation – formát zápisu dat
ORM	Object-relational Mapper
LINQ	Language Integrated Query –jazyk .NET frameworku, umožňující manipulaci s daty (třídění kolekcí, vyhledávání v nich, propojování dat)
DI	Dependency Injection
IIS	Internet Information Services
IoC	Inversion of Control
REST	Representational State Transfer – v souvislosti s API
XML	eXtensive Markup Language
CRUD	Operace Create, Read, Update a Delete
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
XAML	Extensible Application Markup Language
DTO	Data Transfer Object
SOAP	Simple Object Access Protocol
ASP	Active Server Pages

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: SEZNAM PŘÍLOH NA CD

- BankAPI – adresář obsahující řešení s projekty