Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvant. metod

Vývoj mobilní aplikace pro schvalování dokumentů v prostředí Xamarin

Bakalářská práce

Autor: Martin Hůlek

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové březen 2018

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 1.4.2018 ……

Martin Hůlek

Poděkování:

Tímto bych rád poděkoval vedoucímu této bakalářské práce, doc. Ing. Filipu Malému, Ph.D., za odborné vedení, připomínky, praktické rady a celkově za veškerý čas, který mi věnoval. Děkuji také svým přátelům rodině a zaměstnavateli za podporu během studia a přípravy práce.

ANOTACE

Bakalářská práce je zaměřena na vývoj mobilní aplikace pro schvalování dokumentů pomocí nástroje Xamarin. Popisuje základní pojmy ve vztahu k webovým službám a architekturu REST, její základní rysy, výhody a nevýhody. Představuje analýzu a návrh nových informačních systémů a s tím spojený vývoj výsledných aplikací.

Klíčová slova:

Xamarin, Android, .NET, API, REST, webová služba, web service, representational state transfer, SOAP, JSON, nativní aplikace, schvalování dokumentů

ANNOTATION

Title: Development of mobile application for approval of documents in the Xamarin

Bachelor's thesis is focused on development of the mobile application for the approval of documents by Xamarin tool. The thesis describes basic notions in the relationship with web services and REST architecture – basic features, advantages and disadvantages. Representing the analysis and new information systems proposal, the thesis shows development of the resulting applications.

Key words:

Xamarin, Android, .NET, API, REST, web service, representational state transfer, SOAP, JSON, native application, document approval

Obsah

1. Úvod

Každá organizace s větším počtem zaměstnanců jistě vlastní několik informačních systémů, ve kterých její zaměstnanci pracují. Může se jednat např. o systémy pro správu dovolených, cestovních příkazů, požadavků, dále nákup nových položek, plán dovolených, zápis výkazů aj. Obecně můžeme pojmenovat tyto záznamy jako „dokument“. Zaměstnanec pak může schvalovat různé fáze životního cyklu dokumentu, který se může měnit podle jeho typu. Cestovní příkazy a dovolené většinou může schválit vedoucí zaměstnanec, zatímco plán dovolených může mít složitější životní cyklus, kde bude vyžadováno schválení vedoucím a následně zaměstnancem z marketingu. Při nákupu nové věci je naopak nutné mít schválení z účtárny. Takovýchto dokumentů může být za den vygenerováno velké množství a až na výjimky je žádoucí, aby byly schváleny v co nejkratší době. Může však nastat situace, kdy schvalovatel nemůže být přístupný na služebním počítači např. z důvodu služební cesty. V tu chvíli není od věci využít zařízení, které většinou nosíme vždy u sebe – mobilní telefon.

Můžeme tedy uvažovat o využití mobilního telefonu ke schvalování dokumentů a vytvořit tak mobilní aplikaci, ze které je možné schvalovat různé typy dokumentů bez nutnosti nosit s sebou velký a neskladný notebook. K tomuto účelu bude nutné vytvořit mobilní aplikaci a webové API, které bude komunikovat s touto aplikací a sdružovat interní informační systémy do jednoho místa.

Cílem této bakalářské práce je vytvořit mobilní aplikaci pro operační systém Android pomocí nástroje Xamarin a webové API založené na frameworku .NET s podporou webových služeb na architektuře REST. V první části bakalářské práce je popsán nástroj Xamarin pro tvorbu mobilních/wear aplikací a jeho hlavní výhoda sdílení kódu. Následně si detailněji popíšeme architekturu webových služeb nazývanou REST architektura – princip, na kterém je založena, její hlavní rysy a v neposlední řadě i výhody a nevýhody.

V další části bakalářské práce si představíme systém Android a nativní aplikace. Popíšeme si nejdůležitější pojmy při vývoji nativních aplikací, životní cyklus aplikace a její stavy. Následně přejdeme z teoretické části do praktické, kde se v první části se dozvíme, jak bychom měli postupovat při návrhu informačního systému. Bude vysvětleno detailnější

zadání, ze kterého bude provedena následná analýza. Na základě této analýzy v poslední části bude vysvětlen vlastní vývoj obou aplikací.

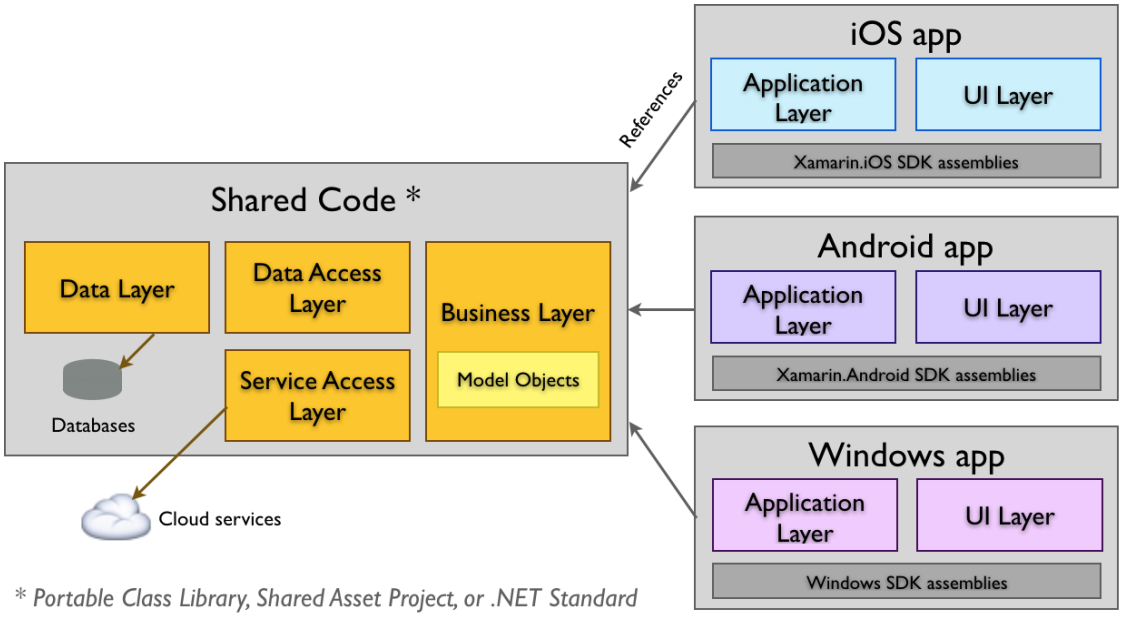
2. Xamarin a webové služby

Xamarin je nástroj pro tvorbu mobilních/wear aplikací, který umožňuje vyvíjet aplikace přes tři mobilní platformy iOS, Android a Windows Mobile pomocí jednoho jazyka C#. Jeho největší výhodou, oproti vývoji v nativních jazycích, je sdílení společného kódu mezi platformami a následné zrychlení vývoje výsledné aplikace pro všechny platformy. Zároveň zůstává zachována 100% dostupnost nativního API daného operačního systému. Více informací o této problematice lze nalézt na [10].

Základ Xamarinu je postaven na projektu zvaném *Mono*, jehož zakladatelem je Miguel de Icaza [19]. Původně byl spravován společností Novell, následně působil jako samostatný subjekt a 24. února 2016 byl Xamarin koupen společností Microsoft[1] [19]. Produkt Mono je založený na ECMA standardech pro C# a Common Language Runtime, který umožňuje psaní multiplatformních aplikací [19]. Název Xamarin vznikl ze názvu opice Tamarin a výměny počátečního písmene T za X [43].

Největší výhodou Xamarinu je sdílení společného kódu (knihovny) mezi platformami. Do tohoto sdíleného kódu (anglicky Shared Code) se definují základní datové modely tříd, business vrstva, servisní vrstva nebo komunikace s webovými službami [10]. Z této knihovny se pak vystaví API, které využívají jednotlivé platformy. Konkrétní projekty jednotlivých platforem pak obsahují pouze aplikační vrstvu a definici uživatelského rozhraní.

[1] Microsoft – americká akciová nadnárodní společnost zabývající se vývojem, výrobou, licencováním a podporou produktů spjaté s výpočetní technikou



Obrázek 1 - Diagram znázorňující použití sdílené knihovny

Zdroj: BURNS: *Přehled sdílení kódů* (2017) dostupné z: *https://docs.microsoft.com/cs-cz/xamarin/cross-platform/app-fundamentals/code-sharing-images/netstandard.png*

Další výhoda, spojená s použitím této metody vývoje, spočívá v ušetřeném času při vývoji a v lepší údržbě zdrojového kódu. Pokud během existence aplikace je potřebné aplikaci rozšiřovat nebo upravovat, provádí se tyto úpravy pouze jednou a na jednom místě, tím výsledný projekt zůstává konzistentní. Nemalou výhodou je větší čistota návrhu a prakticky oddělená business logika s prezentační vrstvou aplikace.

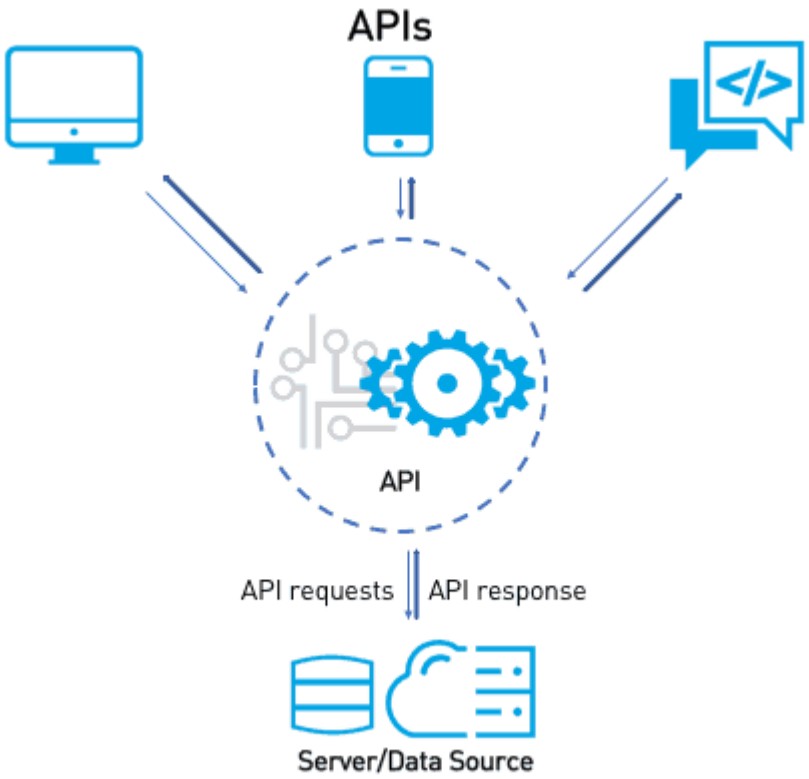
Existují tři metody pro sdílení kódu mezi aplikacemi a platformami [2]:

- **Sdílení projektů** – nejjednodušší způsob, používá se #if direktiva pro výběr správných souborů pro kompilaci
- **Knihovna přenosných tříd** – soběstačná knihovna, která pro komunikaci s platformami využívá rozhraní (interface)
- **Standardní knihovna .NET** – funguje obdobně jako knihovna přenosných tříd, ale je mnohem jednodušší a lépe komunikuje s frameworkem[2] .NET.

[2] Framework – je softwarová struktura sloužící jako podpora při vývoji

2.1. Webové služby: Architektura REST

Webová služba je softwarový systém, který umožňuje komunikaci dvou a více strojů, připojených na síť (internet, intranet apod.) [42]. Data jsou posílána ve strojově zpracovatelném stavu a vždy jsou popsána přiloženou dokumentací.



Obrázek 2 - Ukázka komunikace API s klienty

Zdroj: VERMA: *APIs versus web services* (2018), dostupné z *https://blogs.mulesoft.com/dev/api-dev/apis-versus-web-services/*

Existuje několik architektur, které implementují komunikaci mezi dvěma stroji. Mezi nejznámější patří protokol SOAP[3]. Ten interpretuje data pomocí přiloženého schématu, který se nazývá WSDL[4]. Další velmi novou a známou je a architektura REST

[3] SOAP – Zkratka pro Simple Object Access Protocol, jedná se o protokol pro výměnu zpráv v síti pomocí XML souborů
[4] WSDL – Zkratka pro Web Services Description Language, jedná se o jazyk popisující funkce, které poskytuje webová služba

(Representational State Transfer). V následujících podkapitolách si vysvětlíme základní pojmy této architektury, její principy a úzký vztah k protokolu HTTP.

2.1.1. Historie

Historie architektury REST je spojena s dizertační prací Roye T. Fieldinga [9], který se proslavil jako spoluzakladatel projektu Apache HTTP Server nebo jako jeden z hlavních autorů webového protokolu HTTP [28].

V roce 2000 měli Roy Fielding a jeho kolegové jeden cíl, a to vytvořit standard webových služeb tak, aby každý server mohl komunikovat s jakýmkoliv serverem kdekoliv po světě. Fielding měl tou dobou připomínky od více jak 500 vývojářů, ze kterých byl navržen model zásad, vlastností a omezení, které se nazývají REST [29].

2.1.2. Orientace na zdroje

Základním kamenem pro REST architekturu je orientace na zdroj (Resource), zkráceně ROA (Resource-oriented Architecture) [39]. Zdrojem může být prakticky cokoliv. Může se jednat například o objekt z databáze, soubor, webovou stránku či konkrétní matematický výpočet. Takovýto zdroj ale musí splňovat jedno pravidlo, a to, že každý zdroj se musí označovat jedinečnou URL (Uniform Resource Locator) ve formátu:

Protokol://server:port/cesta?parametry

URL je soubor znaků, který slouží pro jednoznačnou identifikaci zdroje na internetu.

Např.:
https://www.antstudio.cz/slovnik/co-je-url.htm
https://calendar.google.com/calendar/
https://www.google.cz/search?q=hledaný+text

2.1.3. Metody pro přístup

REST implementuje 4 základní metody pro práci se zdrojem, které jsou označovány pod pojmem CRUD [40]:

- Čtení (Read nebo Retrieve)
- Vytvoření (Create)
- Editace (Update)
- Mazání (Delete)

Tyto metody jsou implementovány pomocí metod HTTP protokolu. Nejčastěji se jedná o metody GET, POST, PUT a DELETE [20].

Čtení (Read nebo Retrieve)

Jedná se o základní metodu pro získání zdroje. Každý zdroj na architektuře REST má vlastní identifikátor (URL). Abychom mohli k tomuto zdroji přistupovat, musí být metoda pro čtení implementována příslušnou metodou HTTP protokolu, v tomto případě pomocí GET metody. Více informací o této problematice lze nalézt na [20].

Vytvoření (Create)

Tato metoda slouží pro vytváření nových zdrojů, které ještě nemají svůj jednoznačný identifikátor URL (zdroje ještě neexistují). Z tohoto důvodu se v tomto případě používá předem domluvená společná URL, která je označována pojmem „endpoint“. K implementaci této metody slouží HTTP metoda POST, známá (minimálně) z HTML formulářů. Více informací o této problematice lze nalézt na [20].

Editace (Update)

Tato metoda je podobná operaci vytvoření (Create), s tím rozdílem, že již voláme URL konkrétního zdroje, který chceme změnit a v těle požadavku předáváme nová data. Na rozdíl od operace vytvoření, u operace editace již známe URL adresu zdroje.

Pro použití této metody je využita HTTP metoda PUT, v praxi ale bývá problém jí vyvolat, protože ne každý nástroj tuto metodu podporuje. Z tohoto důvodu RESTová API využívají standardní metodu POST. Více informací o této problematice lze nalézt na [20].

Mazání (Delete)

Tato metoda je obdobná, jako operace Čtení (Read nebo Retrieve). Měla by se zasílat HTTP metoda zvaná DELETE, ale stejně jako u Editace ji ne každý nástroj podporuje a velmi často se místo ní používá GET.

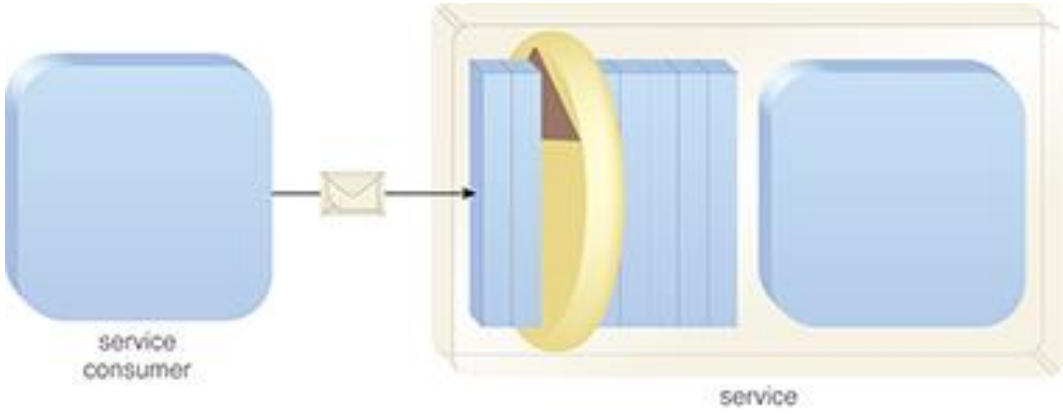
2.1.4. Hlavní rysy

V následující kapitole jsou uvedeny hlavní rysy architektury webových služeb. Webové služby, které splňují následující rysy označujeme pojmem RESTful (česky RESTové) [28].

Komunikační protokol

Architektura REST využívá komunikační architektury označované „Klient-Server“ (Obrázek 3) [28]. Tato architektura podporuje nezávislý vývoj aplikační logiky na straně klienta i serveru a rozšiřuje použitelnost napříč různými platformami jako jsou např. ASP.NET, Java, PHP aj.

Nejdůležitější však je, aby server nabízel jednu nebo více funkcí a naslouchal požadavkům, které přijdou od klienta. Server pak může na takový požadavek odpovědět příslušným chybovým kódem a požadavek odmítne, nebo provede požadovanou funkci a výsledná data vrátí klientovi.



Obrázek 3 - Ukázka klient - server komunikace

Zdroj: BALASUBRAMANIAN, CARLYLE, ERL and PAUTASSO: *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutins with REST.* ISBN: 0137012519. Dostupné z: http://whatisrest.com/rest_constraints/client_server

Bezstavovost

Jedním z nejdůležitějších rysů RESTové služby je bezstavovost. Každý klientský požadavek by měl být samostatný a nezávislý. Tyto požadavky by tedy měli obsahovat všechny potřebné informace, které slouží například pro správnou identifikaci uživatele nebo zdroje. To znamená, že identifikace uživatele by neměla záviset na Session nebo Cookies. Více informací o této problematice lze nalézt na [28].

Souvislost

Souvislost je pojmenování principu, který je známý pod akronymem HATEOS (Hypermedia as the Engine of Application State). Tento princip říká, že další stavy aplikace je možné získat pomocí hyperlinků z aktuálního stavu aplikace. Reálným příkladem je web. Po přechodu na libovolnou doménu, nám server většinou poskytne HTML stránku s různými hypertextovými odkazy, díky kterým se můžeme „proklikat“ na jiné stránky, formuláře, videa atd. Více informací o této problematice lze nalézt na [28].

Code-On-Demand

Code-On-Demand (zkráceně COD) v překladu znamená kód-na-vyžádání [1]. Je to jediné volitelné omezení, které může/nemusí RESTová služba implementovat. Toto omezení je primárně určeno pro rozšíření funkcionality klienta, aby server aktualizoval kód nezávisle na logice klienta. Server poskytuje klientovi možnost stáhnout si různé javascripty, java applety nebo flashové aplikace, které se na klientovi sami spustí. Více informací o této problematice lze nalézt na [9] a [28].

Cache

Podpora pro vyrovnávací paměť (cache) je dalším z omezení RESTových služeb. Ukládání do vyrovnávací paměti nikdy nevytváří stejnou odpověď dvakrát. Nejlepším způsobem, jak zvýšit efektivitu výsledného API, je použití cache brány nebo reverzní proxy, která požadavek na API nejprve zpracuje a v případě existence zdroje ve vyrovnávací paměti odesílá ihned odpověď klientovi. Touto podporou získáme několik výhod pro API:

- **Odolnost vůči krátkodobému výpadku** – systém může poskytovat data i v případě výpadku API
- **Snížení zatížení serverů** – server nemusí každý dotaz zpracovávat pokaždé
- **Rychlost komunikace** – načtení dat z vyrovnávací paměti je rychlejší, než sestavovat data pro každý dotaz zvlášť
- **Více klientů** – pomocí vyrovnávací paměti, můžeme zvýšit počet klientů, kteří mají stejné požadavky na server

Vyrovnávací paměť má i svoji nevýhodu. Tou je nekonzistence dat. Časově dlouhé limity na udržení dat ve vyrovnávací paměti mají za následek chybná, mnohdy i zastaralá data. Více informací o této problematice lze nalézt na [28] a [36].

2.1.5. Výhody RESTových služeb

Na základě informací uvedených v předchozích kapitolách, lze shrnout hlavní výhody použití architektury REST v praxi takto:

- **Jednoduchý vývoj klientů** – Jelikož RESTové služby jsou založeny na HTTP protokolu, vývoj klientů služby je velice jednoduchý. Většina platforem v současné době nabízí celou řadu knihoven pro podporu prací s protokolem HTTP.
- **Separace mezi klientem a serverem** – Architektura REST odděluje uživatelské rozhraní od serveru a úložiště dat. Zlepšuje přenositelnost rozhraní na různé typy platforem a umožňuje, aby různé komponenty vývoje byly vyvíjeny nezávisle.
- **Nezávislost na platformě** – RESTové API je vždy nezávislé na typu platformy, pouze se přizpůsobuje typu syntaxe. Jedinou podmínkou je, že odpovědi na žádosti by měly probíhat v jazycích, které slouží pro výměnu informací. Zejména se jedná o formát XML nebo JSON[5].
- **Libovolný formát zpráv** – Architektura REST nijak neomezuje v používání různých typů formátů. Nejčastějším formátem odpovědi z RESTového API jsou však formáty XML a JSON. Můžeme zde ale použít i formát HTML, vCard[6], iCalendar[7] nebo různé typy formátů pro audio, video, obrázky atd.
- **Jednoduchost** – Požadavky a odpovědi jsou velice jednoduché a většinou i lidsky čitelné.

Více informací o této problematice lze nalézt na [28].

[5] JSON – JavaScript Object Notation – nezávislý datový formát na počítačové platformě.
[6] Specifikace vCard (RFC 6350) dostupné z: http://tools.ietf.org/html/rfc6350
[7] Specifikace iCalendar (RFC 5545) dostupné z: http://tools.ietf.org/html/rfc5545

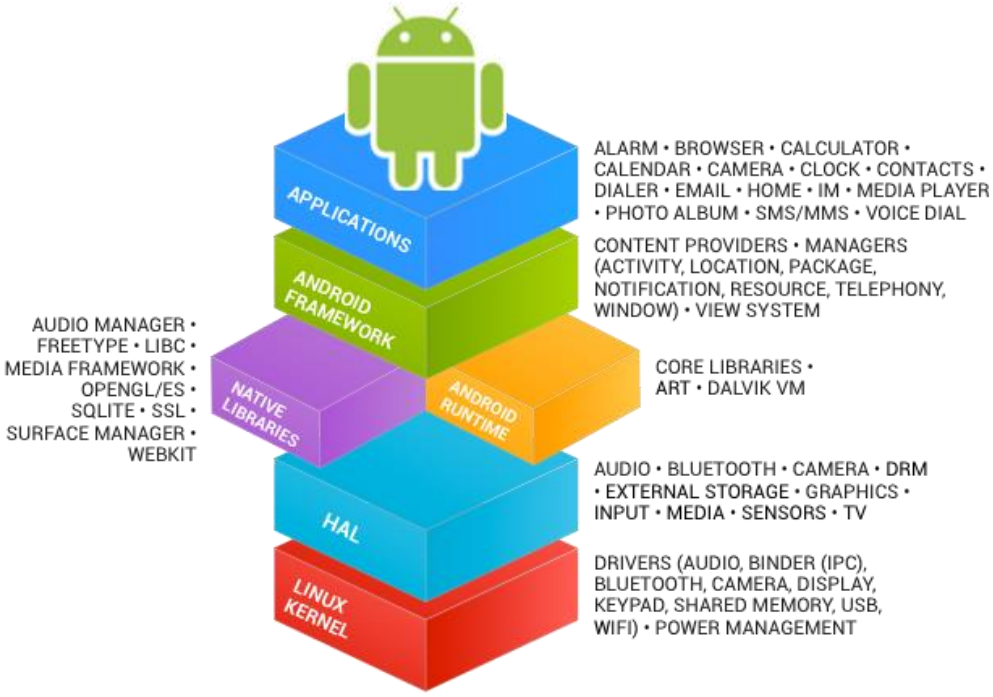
3. Systém Android a nativní aplikace

V této kapitole si představíme, co je to systém Android a nativní aplikace tvořené pro tento systém. Popíšeme si základní pojmy pro vývoj nativní aplikace, které budeme potřebovat v následujících kapitolách.

Operační systém Android je oblíbený operační systém v současné době vyvíjený společností Google[8], primárně určený pro dotyková zařízení, jako jsou např.: chytré telefony, tablety, chytré hodinky. V současné době se ale tento systém dostává i do autorádií a jiných dotykových zařízení. Jeho hlavní úlohou je předávání informací mezi uživatelem a zařízením (hardwarem) např.: Když si budeme chtít pustit hudbu do reproduktoru, systém Android poskytuje tlačítko, které když aktivujeme stiskem, předá zařízení informace, které vykonají spuštění muziky do reproduktoru.

Android je projekt s otevřeným zdrojovým kódem[9] (anglicky open source project), který je nazýván Android Open Source Projekt (zkráceně AOSP). Společnost Google využívá tento projekt jako základ pro vývoj vlastní verze operačního systému, který následně využívají výrobci hardwaru jako jsou např.: HTC nebo Samsung. Tento operační systém je založen na jádře Linux (anglicky Linux kernel). Jedná se o strukturu založenou na operačním systému GNU/Linux, které spadá mezi unixové operační systémy. Výhodou těchto systémů je jeho přenositelnost, multitasking[10] a vysoká bezpečnost. Více informací o této problematice lze nalézt na [15].

[8] Google, Inc. – je americká společnost sídlící v Silicon Valley v Kalifornii, známá především díky skvělému vyhledávači
[9] Otevřený zdrojový kód – jedná se o zdrojový kód, který může každý číst, editovat a vylepšovat
[10] Multitasking – schopnost provádět několik procesů současně



Obrázek 4 - Pohled na architekturu systému Android

Zdroj: ANDROID: *The Android Source Code*, Dostupné z *https://source.android.com/images/android_framework_details.png*

3.1. Nativní aplikace a základní pojmy při vývoji

Nativní aplikace je označení pro aplikaci, která byla vyvinuta pro konkrétní platformu a nelze jí přenést do jiné. Opakem pro nativní aplikace může být webová aplikace, která je platformě nezávislá a lze jí převážně spustit na jakémkoliv zařízení přes webový prohlížeč. Výhodou nativních aplikací je možnost komunikace s hardwarem zařízení, jako jsou fotoaparát, GPS, úložištěm a jiné. Aplikace dále mohou fungovat bez připojení k internetu a mohou využívat vlastnosti operačního systému, proto jsou většinou rychlejší a poskytují větší flexibilitu práce. Naopak nevýhodou takovýchto aplikací je, že pro její spuštění je nutná nová instalace a nelze jí přenést na jiné zařízení a vývoj takovéto aplikace zabere podstatně více času, jelikož je platformě závislá. Tím pádem pro podporu různých typů operačních systémů (v dnešní době mezi nejznámější mobilní operační systémy patří OS Android, iOS od společnosti Apple nebo Windows Phone od společnosti Microsoft) je nutné vytvořit vlastní aplikaci. Více informací o této problematice lze nalézt na [30].

3.1.1. Kompatibilita nativních aplikací

Před samotným vývojem a psaním aplikačního kódu, je nutné si vyspecifikovat, na jakém Android API budeme vyvíjet. Toto nastavení určuje kompatibilitu aplikace s více verzemi operačního sytému Android a určuje, jaké třídy a knihovny můžeme využít při tvorbě aplikace. Použitím vyšší verze API získáváme větší možnosti při vývoji, ale tím pádem ztrácíme podporu pro nižší verze operačních systémů. Proto, si musíme uvědomit, na jakou verzi API budeme cílit a podle ní navrhovat aplikaci. Android vystavuje tři nastavení rozhraní [23]:

- Cílová architektura
- Minimální verze Androidu
- Cílová verze Androidu

Cílová architektura (v nastavení označována jako „compileSdkVersion“) určuje verzi API, která bude sloužit pro vývoj aplikace a pří výsledné kompilaci aplikace. Určuje, jaké knihovny budou dostupné.

Minimální verze Androidu (v nastavení označována jako „minSdkVersion“) určuje nejstarší verzi operačního systému Android (nejnižší verzi API), na který bude možné aplikaci nainstalovat a spustit. Ve výchozím nastavení, by minimální verze měla odpovídat nastavení cílové architektury. V případě, že minimální verze bude nastavena na nižší verzi, než je cílová architektura je nutné zajistit přímo v kódu kontroly verzí a zajistit volání pouze podporovaných funkcí daného API.

```
if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.Lollipop)
{
    builder.SetCategory(Notification.CategoryEmail);
}
```

Kód 1 - Ukázka kontroly verze pomocí kódu

Zdroj: MCLEMORE: *Principy úrovní rozhraní API systému Android* (2018). Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/android-api-levels?tabs=vswin

Cílová verze Androidu (v nastavení označována jako „targetSdkVersion“) je určení, na které verzi operačního systému očekáváme spouštění aplikace. Toto nastavení určuje kompatibilitu verze při instalaci.

Označení	Verze	Datum vydání	API level
Oreo	8.1.0	2017/12	API level 27
Oreo	8.0.0	2017/08	API level 26
Nougat	7.1	2016/12	API level 25
Nougat	7.0	2016/08	API level 24
Marshmallow	6.0	2015/08	API level 23
Lollipop	5.1	2015/03	API level 22
Lollipop	5.0	2014/11	API level 21
KitKat Watch	4.4W	2014/06	API level 20
KitKat	4.4 - 4.4.4	2013/10	API level 19
Jelly Bean	4.3.x	2013/07	API level 18
Jelly Bean	4.2.x	2012/11	API level 17
Jelly Bean	4.1.x	2012/06	API level 16
Ice Cream Sandwich	4.0.3 - 4.0.4	2011/12	API level 15, NDK 8
Ice Cream Sandwich	4.0.1 - 4.0.2	2011/10	API level 14, NDK 7
Honeycomb	3.2.x	2011/06	API level 13
Honeycomb	3.1	2011/05	API level 12, NDK 6
Honeycomb	3.0	2011/02	API level 11
Gingerbread	2.3.3 - 2.3.7	2011/02	API level 10
Gingerbread	2.3 - 2.3.2	2010/11	API level 9, NDK 5
Froyo	2.2.x	2010/06	API level 8, NDK 4
Eclair	2.1	2010/01	API level 7, NDK 3
Eclair	2.0.1	2009/12	API level 6
Eclair	2.0	2009/11	API level 5
Donut	1.6	2009/10	API level 4, NDK 2
Cupcake	1.5	2009/05	API level 3, NDK 1
Base	1.1	2009/02	API level 2
Base	1.0	2008/10	API level 1

Tabulka 1 - Seznam vydaných verzí API

Zdroj: MCLEMORE*: Principy úrovní rozhraní API systému Android* (2018)

3.1.2. Android zdroje

Aplikace není pouze zdrojový kód. Součástí aplikaci je často mnoho dalších souborů, které tvoří aplikaci např.: obrázky, video, písma nebo zvukové soubory. Obecně se tyto

soubory nazývají zdroje (anglicky resources) a jsou kompilované spolu se zdrojovým kódem a zabalené do výsledného balíčku APK[11], připravené k instalaci do zařízení.

Obrázek 5 - Diagram postupu vytváření APK balíčku

Zdroj: MCLEMORE: *Android prostředky* (2018) dostupné z *https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/resources-in-android/images/packaging-diagram.png*

Existují dva způsoby přístupu k těmto zdrojům, přímý přístup z kódu aplikace, nebo deklarativně pomocí syntaxe XML. Tyto deklarativní zdroje se nazývají *výchozí zdroje* a používají se v případě, že není nalezen *alternativní zdroj*. Tyto alternativní zdroje rozšiřují výchozí zdroje a není potřeba zásah do kódu aplikace. Operační systém Android automaticky vybírá správný alternativní nebo výchozí zdroj podle schody v názvu zdroje. Alternativní zdroje jsou zadány krátkým řetězcem (kvalifikátorem) na konec objektu adresáře označující např. kvalitu obrazovky, národní prostředí uživatele, otočení obrazovky o 90°, noční režim aj. Pro změnu zdroje pro německé národní prostředí můžeme použít kvalifikátor „-de“ a výsledný adresář bude „Resources/drawable-de“. Tyto zdroje se dají využít k lokalizaci aplikace nebo změnu konfigurace pro konkrétní zařízení. V aplikaci můžeme mít tyto výchozí typy zdrojů [22]:

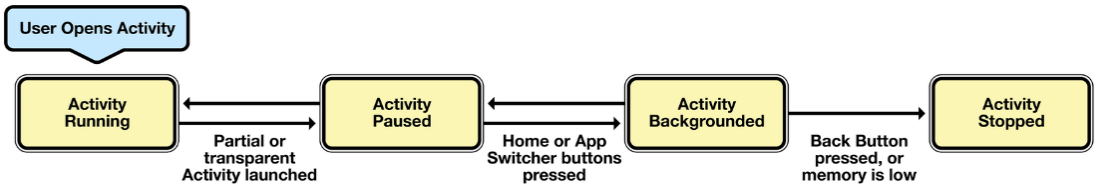
- **Animator** – Soubory XML, které popisují vlastnosti animací.
- **Anim** – Doplňující soubory XML k animacím.
- **Color** – Soubory XML, které obsahují seznamy barev.
- **Drawable** – Jedná se o obecný typ pro obrázky v různých formátech. Můžou existovat jako rastrové obrázky ve formátech GIF, PNG nebo JPEG, ale také jako XML soubor.

[11] APK – zkratka pro Android Package Kit. Jedná se o formát souboru balíčku používaný v operačním systému Android pro instalaci a distribuci mobilních aplikací

- **Layout** – Soubory XML popisující rozvržení uživatelského rozhraní.
- **Menu** – Soubory XML popisující nabídky aplikace.
- **Raw** – Libovolné soubory, které jsou uloženy v binární podobě.
- **Values** – Soubory XML, které obsahují jednoduché textové hodnoty klíč–hodnota.
- **XML** – Soubory XML, které určují konfiguraci aplikace.

3.1.3. Aktivita

Aktivita je základní stavební blok pro Android aplikaci [24]. Bez této části by nešla aplikace spustit. Aktivita je jakýsi controller, který je používán v MVC webových aplikací, pro zajištění komunikace uživatele s nižšími vrstvy aplikace. Aktivita může existovat v různých stavech. Operační systém hlídá stavy aktivit a identifikuje nepoužívané aktivity, které ukončuje, aby uvolnil prostředky pro právě spuštěné aktivity. Na následujícím diagramu jsou znázorněny všechny stavy aktivity během její existence, kde jsou znázorněny čtyři stavy aktivity.



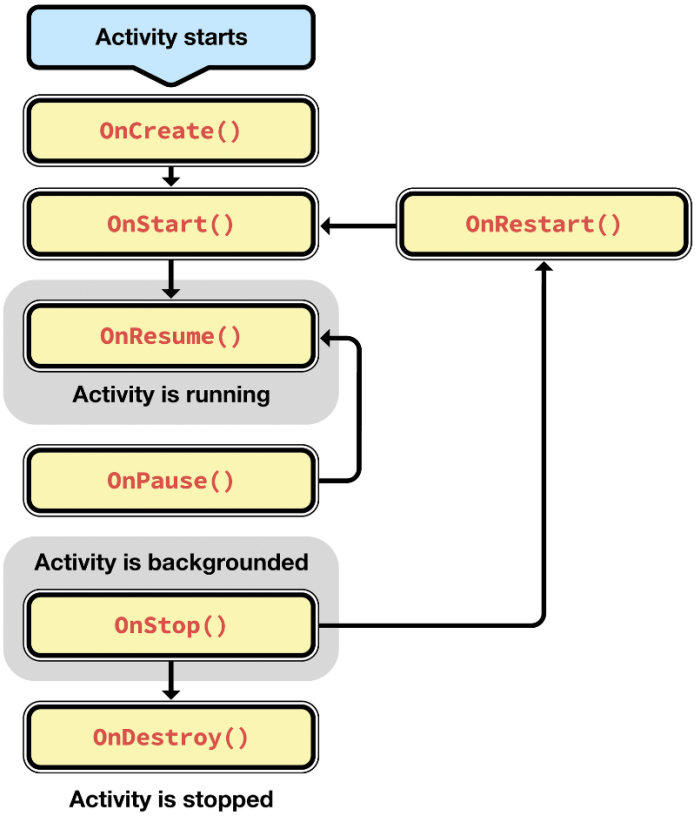
Obrázek 6 - Digram stavů aktivity

Zdroj: MCLEMORE: *Životní cyklus aktivity* (2018) dostupné z *https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/activity-lifecycle/images/image1.png#lightbox*

- **Běžící aktivita** – Jedná se o aktivitu, která je právě spuštěna na popředí a má nejvyšší prioritu.
- **Pozastavená aktivita** – Do tohoto stavu přejde aktivita z důvodů částečným překrytím jiné aktivity, nebo po přechodu zařízení do režimu spánku. Pozastavené aktivity jsou stále aktivní.
- **Aktivita na pozadí** – Takováto aktivita byla překryta zcela jinou aktivitou (spuštěna jiná aplikace). Má nejnižší prioritu.

- **Zastavena** – Jedná se o ukončenou aktivitu. Systém android takovouto aktivitu odebírá z paměti a uvolňuje po ní místo.

Při změně stavu aktivity je volána patřičná metoda odpovídající životnímu cyklu aktivity. V těchto metodách je nutné provádět patřičné operace (uložení stavů, dokončení operací atd.) při přechodech z aktivity do aktivity. Na následujícím diagramu je znázorněn životní cyklus aktivity.



Obrázek 7 - Diagram životního cyklu aktivity

Zdroj: MCLEMORE: *Životní cyklus aktivity* (2018) dostupné z *https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/activity-lifecycle/images/image2.png#lightbox*

4. Analýza a návrh řešení

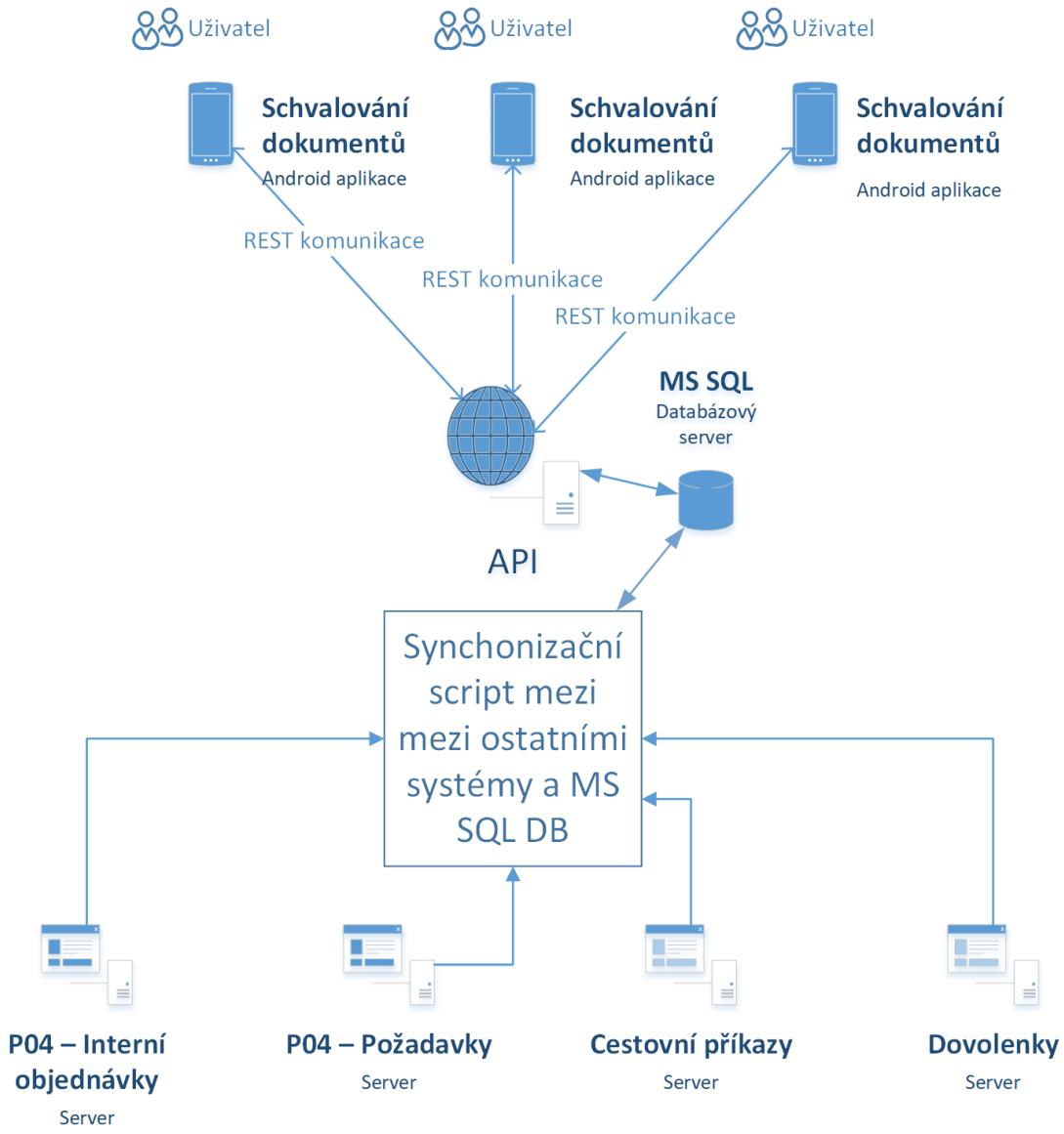
Aplikace *Schvalování dokumentů* má být vyvinuta jako prostředník mezi uživatelem a několika různými systémy pomocí vývojového prostředí Xamarin jako nativní aplikace pro mobilní zařízení s operačním systémem Android. Z této aplikace by se schvalovaly dokumenty, aniž by uživatel musel být přítomen na služebním PC. Aplikace by komunikovala s webovou RESTovou aplikací (dále jen API), která by distribuovala data do Android aplikace a sdružovala ostatní systémy do jednoho systému. Distribuce dat by byla měla být implementována pomocí REST architektury.

Aplikace by sdružovala tyto ostatní systémy:

- Systém **Dovolenky** – Správa žádostí o dovolenou
- Systém **Cestovní příkazy** – Správa žádostí o cestovní příkaz
- Systém **P04 – Požadavky** – Správa požadavků o nákup nových věcí
- Systém **P04 – Interní objednávky** – Správa interních firemních objednávek

Tyto systémy jsou interními aplikacemi dané společnosti.

Pro ukládání záznamů z různých systému by byl použit databázový server MS-SQL do kterého by synchronizační script synchronizoval data.



Obrázek 8 - Pohled na schéma komunikace a rozložení serverů v síti

Zdroj: vlastní zpracování

Synchronizační mechanizmus mezi API a ostatními systémy nejsou součástí této bakalářské práce.

4.1. Požadavky na aplikaci

V této kapitole si popíšeme funkční a non-funkční požadavky na funkčnost aplikace. Analýza požadavků, je velmi důležitá k úspěšnému dokončení projektu vývoje. Požadavky musejí být proveditelné, měřené, testovatelné a musí být vztahovány k tématu aplikace.

4.1.1. Funkční požadavky

Funkční požadavky určují, co má budoucí systém nabízet uživatelům za služby a jeho chování. Pro znázornění funkčních požadavků využijeme Use Case Diagram (česky diagram případu užiti). Tento diagram zobrazuje chování systému z pohledu uživatele a popisuje co má systém umět. Proto by tento diagram měl být vytvořen jako první při návrhu informačního systému.

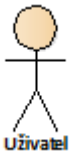
Use Case Diagram se skládá z use-case (případu užití), aktérů a vztahů mezi nimi.

Use-case (případ užití) je sada několika akcí, které vedou k úspěšnému dosažení určitého cíle. Definuje jednu funkcionalitu, kterou by měl navrhovaný systém umět a kým je spouštěn. Tento samotný use-case můžeme rozšířit o několik doplňujících atributů, pro zpřesnění funkcionality daného cíle. Jedná se o krátký popis, podmínky pro spuštění, podmínky pro dokončení a nejvíce důležitý scénář. Scénář obsahuje interakci mezi uživatelem a systémem. Zapisujeme jej jako posloupnost kroků (bodů). Více informací o této problematice lze nalézt na [4].

Obrázek 9 - Znázornění use-case v diagramu

Zdroj: vlastní zpracování

Aktér je role v systému, která komunikuje pomocí **vztahů** s případy užití. Role může být uživatel, externí systém nebo také např. čas.

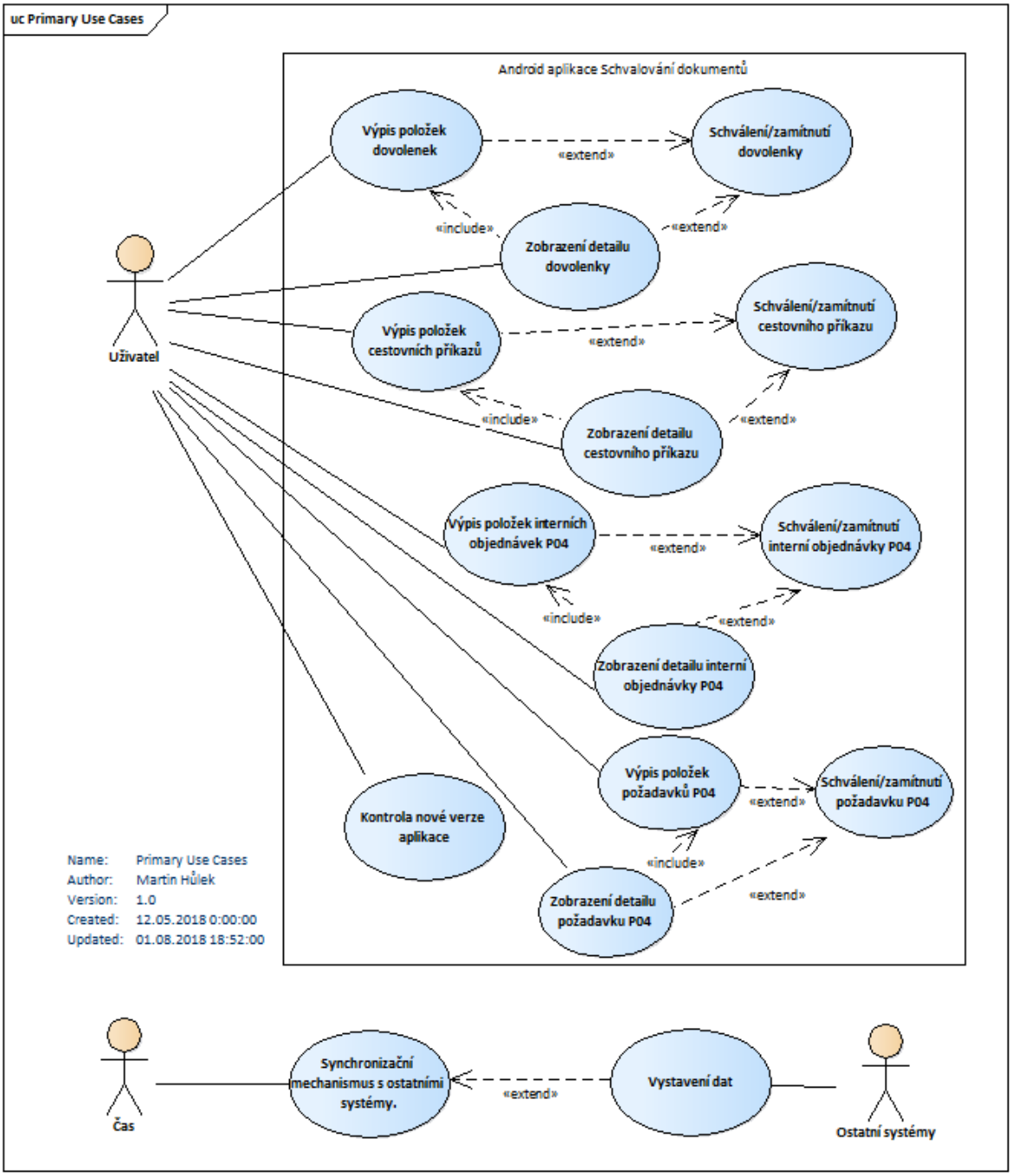


Obrázek 10 - Znázornění aktéra v diagramu

Zdroj: vlastní zpracování

Na základě výše uvedených pojmů můžeme vytvořit výsledný digram k navrhovanému systému.

Na diagramu níže je znázorněn Use Case Diagram navrhovaného systému z pohledu uživatele. Jsou zde použiti tři aktéři: Uživatel, Čas a Ostatní systémy. *Uživatel* zde představuje zástupce zaměstnance, který bude pracovat s mobilní aplikací pro schvalování dokumentů. Jedná se o hlavního aktéra, který má možnost např.: zobrazit výpis položek, zobrazit detail položky a schválit/zamítnout položku. *Čas* zde představuje plánovač úloh, který bude dle specifikace synchronizovat změněná data z/do ostatních systémů. Poslední aktér *Ostatní systémy*, zde představuje množinu vystavených služeb od různých systémů, které vystavují konkrétní data, která se mají schvalovat. V našem případě se jedná o systémy: *Dovolenky*, *Cestovní příkazy*, *Požadavky P04* a *Interní objednávky P04*.



Obrázek 11 - Use Case Diagram navrhované aplikace

Zdroj: vlastní zpracování

4.1.2. Non-funkční požadavky

Non-funkční požadavky jsou omezující podmínky uvalené na výslednou android aplikaci a webové API. Specifikují, jakým způsobem bude systém implementován (např. použité technologie, požadavky na kvalitu aj.). Non-funkční požadavky budou vypsány formou seznamu.

Číslo požadavku	Aplikace	Popis požadavku
1.	Android	Minimální verze operačního systému: Android 4.3
2.	Android	Cílová verze operačního systému: Android 5.1
3.		Komunikace přes https protokol
4.	API	Architektura REST a použití JSON formátu
5.		Rychlá odezva aplikace
6.	API	Framework 4.5 a více

Tabulka 2 - Non-funkční požadavky na aplikace

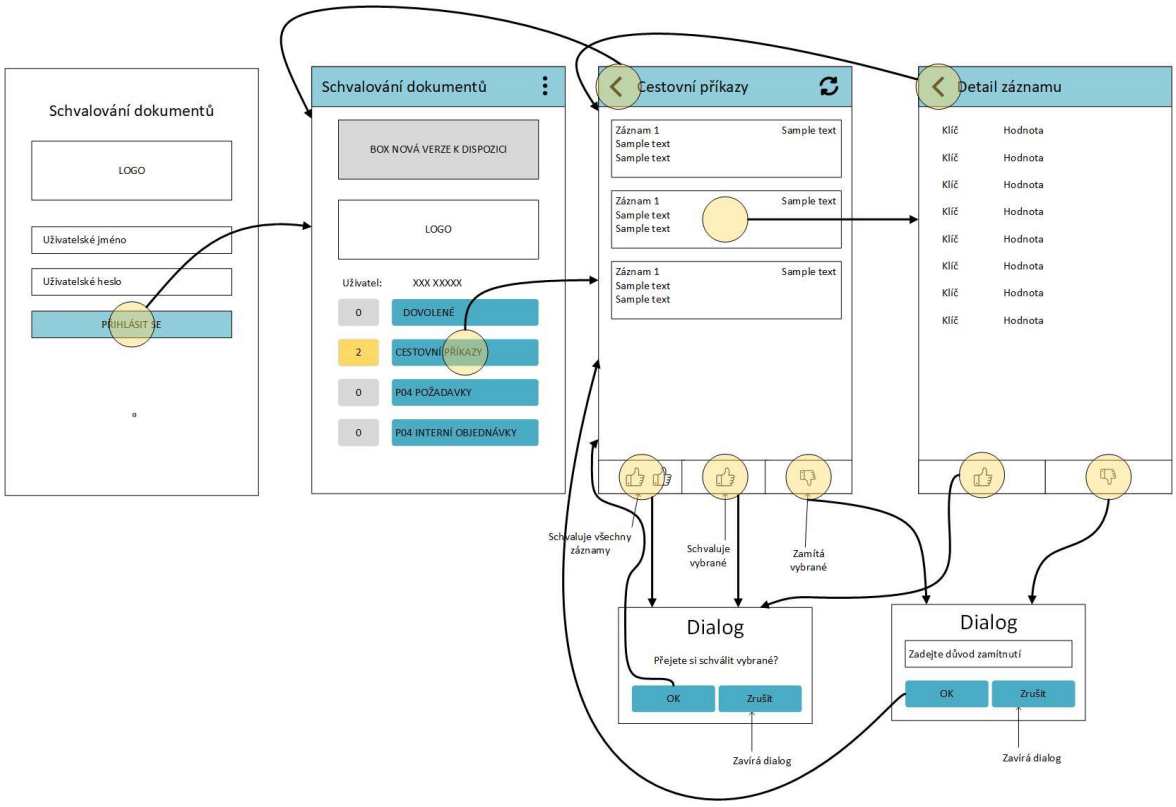
Zdroj: vlastní zpracování

4.2. Uživatelské rozhraní – Drátový model

Uživatelské rozhraní (anglicky User interface, dále jen UI) je nástroj, skrze který komunikuje aplikace (systém) s uživatelem. Je to první věc, kterou uživatel po spuštění aplikace uvidí a se kterou bude neustále přicházet do styku. Proto je potřeba dbát na to, aby UI bylo správně a nejlépe navrhnuto. Správné rozložení prvků na obrazovce nám poskytne snadné a intuitivní ovládání, zvýší efektivnost práce s aplikací a minimalizuje chyby způsobené špatným ovládáním. Návrh uživatelského rozhraní je tedy velice důležitá věc, a proto na základě Use-case požadavků můžeme nejprve vytvořit prototyp UI (drátový model). Pomocí tohoto prototypu navrhneme chování aplikace, vytvoříme všechny obrazovky aplikace a nastíníme průchod aplikací. Na základě tohoto modelu pak v budoucnu budeme vytvářet samotný vzhled aplikace, a tím se vyhneme vytváření a testování UI při vývoji aplikace.

V následujícím diagramu je znázorněn drátový model mobilní aplikace s možným uživatelským průchodem a popisem jednotlivých funkcí. Průchod je znázorněn pro jeden konkrétní modul „Cestovní příkazy“. Pro ostatní moduly se předpokládá použití stejného chování. První obrazovka znázorňuje úvodní přihlašovací formulář do aplikace, kde uživatel

zapisuje přihlašovací jméno a heslo. Druhá obrazovka znázorňuje domovskou stránku s funkcemi, logem a informačním boxem o nové verzi aplikace. Z této domovské stránky se uživatel dostává na seznam záznamů, kde může hromadně nebo jednotlivě schvalovat či zamítat záznamy. Kliknutím na jednotlivý záznam se uživatel přesune do detailu záznamu, kde uživatel může schválit nebo zamítnou vybraný záznam. Schvalování záznamů je podmíněno potvrzujícím dialogem. Zamítnutí záznamu je podmíněno zapsáním důvodu v dialogu a potvrzením tohoto dialogu.



Obrázek 12 - Drátový model s uživatelským průchodem a popisem funkcí

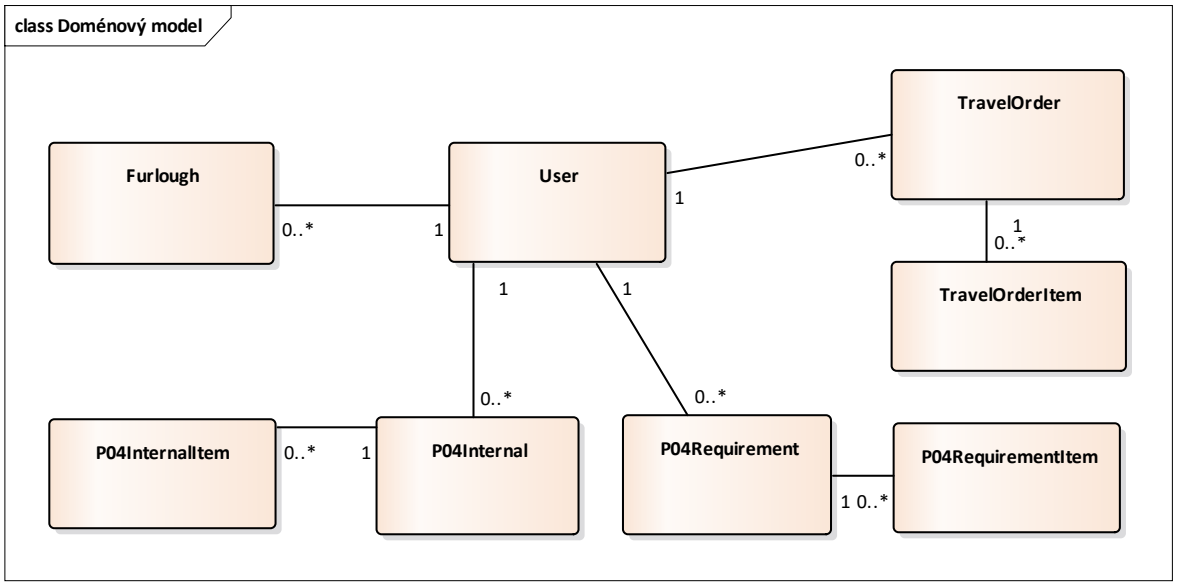
Zdroj: vlastní zpracování

4.3. Doménový model

Doménový model se vytváří v první fázi vývoje informačního systému spolu s Use Case Diagramem. Jeho úkolem je znázornit typy objektů a jejich vztahy, které se v systému budou objevovat. Základní entitou je v tomto diagramu třída. Třídy v doménovém modelu jsou značně zjednodušeny, neobsahují metody a mohou mít pouze důležité atributy bez

uvedení typů. Tento model je nezávislý na platformě (nespecifikuje konkrétní programovací jazyk) a představuje jakýsi statický pohled na modelovaný systém. Více informací o této problematice lze nalézt na [6].

V následujícím diagramu jsou znázorněny entity výsledného informačního systému. Základní entitou je zde *User*. Jedná se o třídu, která specifikuje uživatele (zaměstnance), který bude moci schvalovat příslušné dokumenty. Dalšími entitami jsou třídy určující daný typ dokumentu. O dokument cestovních příkazů se stará entita pod názvem *TravelOrder*. Tato třída má vztah se třídou *TravelOrderItem*, která reprezentuje jeden záznam cestovního příkazu. Takovýchto záznamů může mít třída *TravelOrder* N. Dokument žádost o dovolenou (zkráceně dovolenka) je reprezentována třídou *Furlough*. Dokument ze systému P04 Interní objednávky je znázorněn třídou pod názvem *P04Internal*. Tato třída obsahuje informace o objednávce (kdo jí zadal, popis, datum atd.) a kolekci tříd *P04InternalItem*, které obsahují informace o položkách objednávky. Poslední entitou v následujícím diagramu je třída *P04Requirement*. Tato třída reprezentuje dokument ze systému P04 Požadavky. Obsahuje informaci o novém požadavku zaměstnance a kolekci tříd *P04RequirementItem*, které reprezentují položku v požadavku.



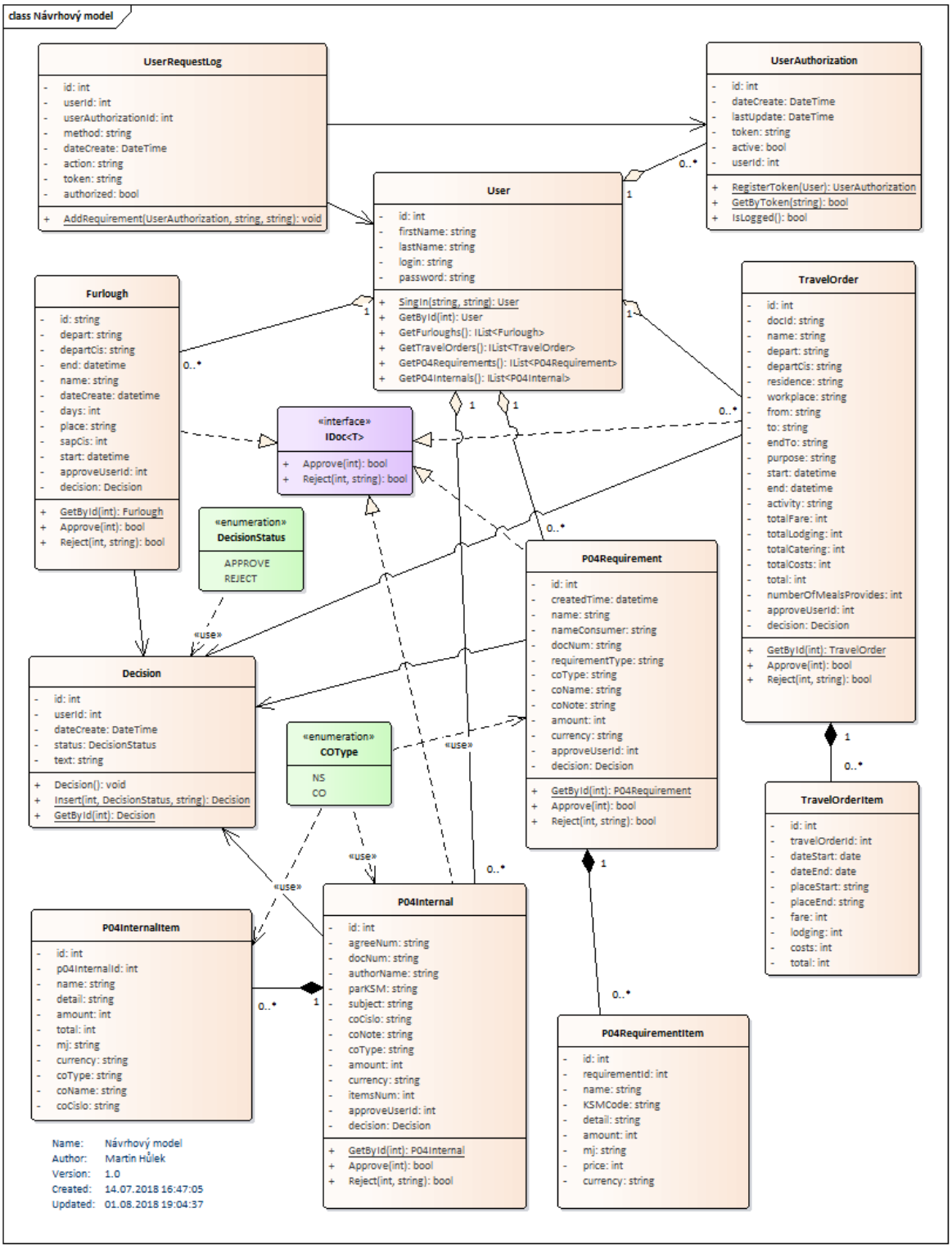
Obrázek 13 - Doménový model

Zdroj: vlastní zpracování

4.4. Návrhový model tříd

Návrhový model tříd (class diagram) vychází z doménového modelu a rozšiřuje jej. Oproti doménovému modelu, který je spíše analytickým pohledem na informační systém, class diagram musí být úplný. Obsahuje všechny třídy, které systém bude obsahovat se všemi atributy a metodami. Tento diagram oproti doménovému modelu je platformě závislý a je nutné jej modelovat pro konkrétní programovací jazyk. V tomto případě se bude jednat o programovací jazyk C#, který vyvinula společnost Microsoft. Více informací o této problematice lze nalézt na [7].

Diagram byl rozšířen o několik pomocných tříd a interface, které jsou potřeba pro správnou funkčnost systému. Interface s názvem *IDoc* slouží k určení definic, které třídy pro schvalované dokumenty musí implementovat. Jedná se o metody *GetById*, *Approve* a *Reject*. Třídy pro správu dokumentů: *Furlough*, *TravelOrder*, *P04Internal* a *P04Requirement* implementují tento interface. Dále byly doplněny všechny potřebné atributy a metody. Třída *User* byla rozšířena též o atributy a metody. Jedna z těchto metod je pojmenována jako *Login*. Tato metoda ověřuje identitu uživatele pomocí přihlašovacího jména a hesla. Po úspěšné autentifikaci se vytváří třída *UserAuthorization*, která registruje token pro následnou komunikaci. Obsahuje též metodu pro následné ověření uživatele pomocí tokenu nazvanou *GetByToken*. Další pomocnou třídou je *UserRequestLog*. Tato třída slouží pro logování volaných služeb. Obsahuje jednu metodu s názvem *AddRequirement*, která sbírá data a ukládá jej do databáze. Posledními pomocnými třídami jsou *ApproveStatus* a *COType* typu enum. Tyto třídy slouží pouze pro výčet hodnot.



Obrázek 14 - Návrhový model

Zdroj: vlastní zpracování

5. Vývoj vlastního řešení

Hlavní úlohou aplikace je schvalovat různé typy dokumentů přes mobilní aplikaci navrženou pro systém Android. Uživatel bude mít možnost vypsat neschválené a nezamítnuté záznamy, schválit nebo zamítnout záznam a zobrazit si detail záznamu. Aplikace je implementována na základě analýzy dle definovaných uživatelských požadavků v Kapitole 4. Výslednými aplikacemi jsou mobilní aplikace pro systém Android (dále jen Klient), která obstarává komunikaci s uživatelem, a webová API aplikace (dále jen API), která komunikuje s aplikací Klient a poskytuje data. Architekturou komunikace mezi Klientem a API je zvolena REST architektura s použitím JSON formátu.

V této kapitole si rozebereme implementaci obou systémů zvlášť. K implementaci obou systémů využijeme profesionální vývojářský nástroj od společnosti Microsoft s názvem Visual Studio. Tento nástroj v sobě obsahuje editor kódů podporující IntelliSense[12] pro vývoj mobilních aplikací na platformě Xamarin i vývoj webových aplikací.

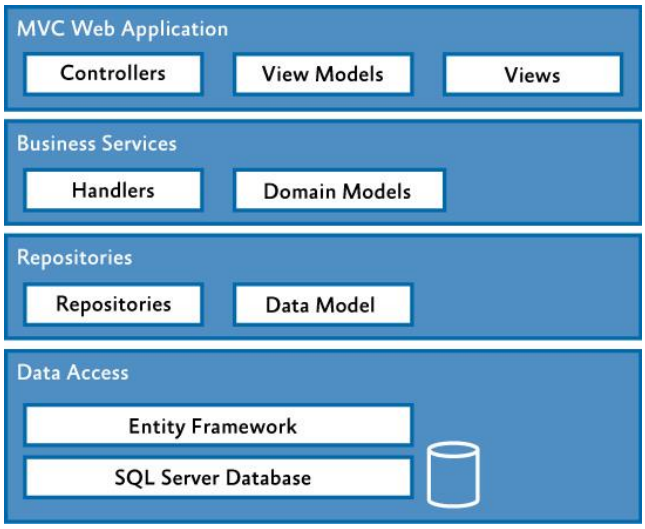
Na začátku vytvoříme nové řešení, které pojmenujeme *MobileApp*. Do tohoto řešení přidáme čtyři projekty. První projekt bude reprezentovat aplikaci API a pojmenujeme ho *WebApi*. Druhý projekt bude reprezentovat mobilní aplikaci pod označením *Mobile*. Abychom oddělili aplikace od sdíleného kódu (třídy pro práci s databází, modelové třídy, interface atd.) vytvoříme ještě dva projekty typu Knihovna tříd (anglicky Class library), první z nich ponese jméno *WebApiDataAccess* a druhý pak *MobileDataAccess*. Tyto knihovny tříd následně nareferencujeme pomocí odkazu na jiný projekt v řešení.

5.1. Vývoj API webové aplikace

Nyní si popíšeme vlastní vývoj API aplikace. Jejím hlavním úkolem je komunikace s Klientem a databází, kde jsou uložena data. Tato aplikace je implementována pomocí .NET

[12] IntelliSense – nástroj pro dokončování psaného kódu nebo zobrazení informací o parametrech (dokumentace)

frameworku[13] ve verzi 4.5.2 a vyvinuta na architektuře MVC[14]. O zprostředkování komunikace s databází se stará databázový framework nHibernate[15]. Na začátku vývoje je dobré si navrhnout správný postup vývoje informačního systému. Webové aplikace nejsou zodpovědné pouze za vracení obsahu ve formátu HTML, ale zahrnují také datové modely, přístupy k datům a ukládání dat, komunikaci s ostatními systémy, zabezpečení a i internacionalizaci. Proto je dobré oddělit vrstvy architektury nejenom v samotné aplikaci, ale i v samotné práci s daty. Na následujícím obrázku je znázorněno schéma architektury, která odděluje data na vysoké úrovni [26]. Máme zde k dispozici čtyři vrstvy architektury. Spodní vrstva s názvem **Data Access** nám udává, kde budou data uložena. V našem případě se jedná o databázový server MS-SQL. Druhá vrstva s názvem **Repozitories** nám obstarává komunikaci mezi databází a Business vrstvou. V našem případě se jedná o tzv. Dao třídy o kterých jsou popsány v kapitole č. 5.1.2. Další vrstvou je již zmíněná **Business vrstva**. Tato vrstva obsahuje servisní třídy, které již komunikují s webovou aplikací a s Dao třídami. V této vrstvě je implementována i aplikační logika. Poslední vrstva s názvem **MVC Web Application** je poslední vrstvou a jedná se o výslednou aplikaci. Ta je navíc ještě rozšířena o architekturu MVC.



Obrázek 15 - Schéma architektury

Zdroj: MICROSOFT: Chapter 11: Server-Side Implementation (2012), dostupné z https://docs.microsoft.com/en-us/previous-versions/msp-n-p/images/hh404093.f50cd137-c414-4c13-84cd-ff1dcabed82e%28en-us%2cpandp.10%29.png

[13] Fremework – softwarová struktura, která slouží jako podpora při vývoji a organizaci jiných softwarových produktů
[14] MVC – návrhový vzor Model-View-Controller, slouží k oddělení aplikační logiky a uživatelského rozhraní
[15] nHibernate – framework založený na objektově-relačního mapování (ORM) pro platformu .NET

5.1.1. Připojení k databázi

Jak bylo zmíněno výše, pro komunikaci s databází se stará framework nHibernate. Nejjednodušší způsob instalace podpory tohoto frameworku je přes balíčkovací nástroj NuGet. Po úspěšné instalaci je nutné udělat pár nezbytných kroků.

Prvním krokem je definovat konfiguraci a důležité parametry pro připojení k databázovému serveru. Do konfiguračního souboru web.config aplikace *WebApi* vložíme konfiguraci pro nHibernate. V sekci *configSection* vytvoříme element *section*, který inicializuje blok kódu *hibernate-configuration* pro konfiguraci nHibernate. V tomto bloku kódu nastavíme použitý ovladač pro komunikaci, název připojovacího řetězce (anglicky connection string) kde jsou uvedeny potřebné informace pro připojení k databázovému serveru a v neposlední řadě, název jmenného prostoru, kde jsou uloženy mapovací soubory k tabulkám a příslušené Dao[16] třídy. Více informací o této problematice lze nalézt na [34].

```
  <configSections>
    <section name="hibernate-configuration"
type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
  </configSections>
  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory>
      <property
name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
      <property
name="connection.connection_string_name">WebApiConnectionString</property>
      <property name="dialect">NHibernate.Dialect.MsSql2008Dialect</property>
      <mapping assembly="WebApiDataAccess" />
    </session-factory>
  </hibernate-configuration>
  <connectionStrings>
    <add name="WebApiConnectionString" providerName="System.Data.SqlClient"
connectionString="Data Source=MARTIN-I7\SQLEXPRESS;Initial
Catalog=mobileapp;Persist Security Info=True;User
ID=mobileapp;Password=mobileapp;" />
  </connectionStrings>
```

Kód 2 - Konfigurace pro nHibernate

Zdroj: vlastní zpracování

Po nakonfigurování nHibernatu je zapotřebí vytvořit třídu, která bude mít na starost komunikaci. Do projektu *WebApiDataAccess* vytvoříme třídu, kterou pojmenujeme *NHibernateHelper.cs*. Tato třída nám vygeneruje na základě konfigurace tzv.

[16] Dao – Data-access-object – objekt, který se stará o přístup k datům

SessionFactory, která při inicializaci načte všechny mapovací soubory, jejich podrobnosti a vlastnosti související s databází. Vytvoření tohoto objektu je velmi náročné na výpočetní výkon, proto je výhodné využít na přístup k tomuto objektu návrhový vzor[17] Jedináček (anglicky Singleton). Tento návrhový vzor nám řeší situaci, kdy potřebujeme, aby v celém programu běžela pouze jedna instance dané třídy.

```
public class NHibernateHelper
{
        private static ISessionFactory _factory;
        private static ISession _s;
        public static ISessionFactory Factory
        {
            get
            {
                if (_factory == null)
                {
                    var cfg = new Configuration();
                    _factory = cfg.Configure().BuildSessionFactory();
                }
                return _factory;
            }
        }

        public static ISession OpenSession()
        {
            return Factory.OpenSession();
        }
}
```

Kód 3 - Inicializace nHibernate Factory

Zdroj: vlastní zpracování

Posledním krokem pro úspěšnou práci s databází, je vytvořit tzv. mapovací soubory, ze kterých se databázovému frameworku udává, kterou tabulku má namapovat na příslušnou modelovou třídu. Jsou zde uvedeny všechny atributy třídy a k nim přiřazeny příslušné sloupce v databázové tabulce. Samozřejmostí je i identifikace názvu třídy a názvu tabulky. Doplňujícími, ale též neméně podstatnými, informacemi jsou atributy *assembly* a *namespace*, které určují, kde se mají příslušné třídy vyhledávat. Mapovací soubory jsou v projektu uloženy v adresáři *Mappings* a jejich název má daný formát: *názevTřídy.hbm.xml*.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                   assembly="WebApiDataAccess"
                   namespace="WebApiDataAccess.Model">

  <class name="UserRequirements" table="UserRequirements_Log">
```

[17] Návrhový vzor – popis řešení problému nebo šablona, která se dá využít při různých situacích

```
    <id name="Id" column="id">
      <generator class="native" />
    </id>
    <property name="UserId" column="userid"/>
    <property name="DateCreate" column="datecreate"/>
    <property name="Method" column="method"/>
    <property name="Action" column="action"/>
    <property name="Val" column="val"/>
    <property name="Inputs" column="inputs" type="StringClob"/>
    <property name="Results" column="results" type="StringClob"/>
    <property name="Token" column="token"/>
    <property name="Authorized" column="authorized"/>

  </class>
</hibernate-mapping>
```

Kód 4 - Mapovací soubor UserRequirements.hbm.xml

Zdroj: vlastní zpracování

V neposlední řadě musíme vytvořit ještě modelové třídy, do kterých nám nHibernate namapuje výsledná data. Tyto modelové třídy uložíme do adresáře *Model* a obsahují atributy, které mohou korespondovat se sloupci v databázi.

```
public class UserRequirements : IEntity
{
        public virtual int Id { get; set; }
        public virtual string UserId { get; set; }
        public virtual string Action { get; set; }
        public virtual string Method { get; set; }
        public virtual DateTime DateCreate { get; set; }
        public virtual string Val { get; set; }
        public virtual string Results { get; set; }
        public virtual string Inputs { get; set; }
        public virtual string Token { get; set; }
        public virtual bool Authorized { get; set; }
}
```

Kód 5 - Ukázka modelové třídy UserRequirements

Zdroj: vlastní zpracování

Každá modelová třída implementuje rozhraní *IEntity*, které nám určuje, že každá entita musí obsahovat atribut *Id*, který nám udává primární klíč. Bez tohoto klíče nHibernate nelze použít.

5.1.2. Práce s daty – Dao třídy

Abychom mohli jednoduše přistupovat k úložišti dané třídy (v tomto případě se jedná o databázi), je vhodné vytvořit pro každou entitu v našem systému tzv. Dao[18] třídu. Dao třída je v podstatě objekt nebo rozhraní, které poskytuje přístup úložnému prostoru nebo databázi. Tyto třídy pak definují operace pro práci s daným objektem, zejména se jedná o základní operace vytvoření, editace, smazání a další. Abychom si usnadnili vývoj a nevytvářeli Dao třídy pro konkrétní entity se stále stejnými metodami, vytvoříme si obecnou třídu[19] (anglicky generic class) s názvem *DaoBase* a do ní vytvoříme obecné metody pro práci s obecným typem objektu. Následně všechny vytvářené Dao třídy, které budou specifické pro konkrétní typ objektu, budou dědit z této *DaoBase* třídy. Tím zdědí i její metody a výsledný kód bude udržitelnější. Více informací o této problematice lze nalézt na [34].

```
public class DaoBase<T> : IDaoBase<T> where T : class, IEntity
{

    protected DaoBase()
    {

    }

    public virtual object Create(T entity)
    {
        object o;
        using (ISession session = NHibernateHelper.OpenSession())
        {
            using (ITransaction transaction = session.BeginTransaction())
            {
                o = session.Save(entity);
                transaction.Commit();
            }
        }
        return o;
    }

    public virtual void Update(T entity)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            using (ITransaction transaction = session.BeginTransaction())
            {
                session.Update(entity);
                transaction.Commit();
            }
        }
  }
     … //Další metody
}
```

[18] Dao – Data-access-object – objekt, který se stará o přístup k datům
[19] Obecné třídy – zapouzdřují operace, které nejsou specifické pro určitý datový typ

Kód 6 - Ukázka části třídy BaseDao.cs

Zdroj: vlastní zpracování

5.1.3. Business vrstva

V předchozí podkapitole jsme si popsali vytvoření Dao tříd pro práci s úložištěm, které nám propojují databázi s entitami. Nyní si vytvoříme tzv. **Business vrstvu**, respektive servisní vrstvu, do které budeme implementovat potřebné operace pro práci s daty tzv.: aplikační logiku. Tato vrstva by měla obsahovat minimálně třídy s jejich atributy a metody, které byly navrženy v návrhovém modelu tříd z kapitoly 4.4. Každá servisní třída spolupracuje s nižší vrstvou, která se stará o data, a dále předává data do aplikační vrstvy.

V následující ukázce kódu je představena servisní třída *Furlough*, která se stará o práci s entitou dovolenky. Tato třída implementuje rozhraní *IDoc*, které specifikuje metody a atributy, jaké třída musí implementovat. Jedná se o metody *Approve* a *Reject*, které slouží ke schválení a zamítnutí záznamu. Dále třída obsahuje atributy, které jsou viditelné a pouze pro čtení do vyšší vrstvy, a statické metody *GetById* a *GetAllByUserId*. Metoda *GetById* se stará o získání záznamu dle určitého id spolu s ověřením, zda-li záznam může zobrazit příslušná autorizace. Metoda *GetAllByUserId* vrací seznam všech záznamů, nad kterém neproběhlo schválení/zamítnutí.

```
  public class Furlough : IDoc<Furlough>
  {

      private FurloughDB _item { get; set; }
      public int Id { get { return _item.Id; } }
      public string Depart { get { return _item.Depart; } }
      public string DepartCis { get { return _item.DepartCis; } }
      public DateTime End { get { return _item.End; } }
      public string Name { get { return _item.Name; } }
      public DateTime DateCreate { get { return _item.DateCreate; } }
      public double Days { get { return _item.Days; } }
      public string Place { get { return _item.Place; } }
      public string SapCis { get { return _item.SapCis; } }
      public DateTime Start { get { return _item.Start; } }
      public int ApproveUserId { get { return _item.ApproveUserId; } }
      private Decision decision = null;
      public Decision Decision […]
public bool IsExists […]

  public static Furlough GetById(int id, IAuthorization auth) […]
      public bool Approve(int userId) […]
      public bool Reject(int userId, string text)
```

```
        {
            FurloughDao dao = new FurloughDao();

Decision decision = Decision.Insert(userId,     Enum.DecisionStatus.REJECT,
text);
            _item.DecisionId = decision.Id;
            dao.Update(_item);
            return true;
        }

 public JSON.Furlough ToJson()[…]
   public static IList<Furlough> GetAllByUserId(IAuthorization auth) […]
   }
```

Kód 7 - Ukázka servisní třídy pro entitu dovolenky

Zdroj: vlastní zpracování

5.1.4. Vystavení RESTových služeb

Po implementaci všech potřebných podvrstev, které jsme se dozvěděli v kapitolách výše, nyní můžeme implementovat funkce pro volání RESTových služeb. V projektu *WebApi* tedy vytvoříme nový *Controller* (třída) pod názvem *ServiceController*, který bude dědit z controlleru *ApiController*. Tato třída bude obsahovat metody (služby) pro přihlášení uživatele, získání dat nebo schválení/zamítnutí záznamů. Každá metoda, až na výjimku Auth, bude obsahovat ověření uživatele pomocí atributu token a příslušný výkonný kód. V případě, že se nepodaří ověřit totožnost příslušeného dotazu, bude vrácen stavový HTTP kód[20] 401 Unauthorized.

Seznam implementovaných metod s argumenty:

- **Auth**([FromBody] AuthorizationViewModel auth)
- **GetAppVersion**(int platform, string token)
- **GetAll**(string token)
- **GetFurlough**(string token)
- **SetFurlough**([FromBody] FurloughViewModel furloughItems, string token)
- **GetTravelOrder**(string token)
- **SetTravelOrder**([FromBody] TravelOrderViewModel travelOrderItems, string token)

[20] Stavový HTTP kód – je součástí hlavičky odpovědi a upřesňuje, jakým způsobem byl požadavek vyřízen.

- **GetP04Requirement**(string token)
- **SetP04Requirement**([FromBody] P04RequirementViewModel p04ReqItem, string token)
- **GetP04Internal**(string token)
- **SetP04Internal**([FromBody] P04InternalViewModel p04InItem, string token)

5.1.5. Routování

Webové aplikace na architektuře MVC nám umožňují zpracovávat friendly URL (přátelská URL[21]) a delegovat je správným controllerům a jejich metodám včetně předání příslušných parametrů. Tato možnost se nazývá „routování“. Seznam všech rout pro webovou aplikaci je uložen v tzv. *RouteCollection* a jejich nastavení se provádí v souboru *RouteConfig.cs*, který je typicky uložen v adresáři *App_Start*. Pokud ovšem používáme webovou aplikaci i jako API, nelze použít toto nastavení, jelikož API využívá vlastní uložení rout. Nastavení pro API se provádí v souboru *WebApiConfig.cs* a též v adresáři *App_Start*. V tomto souboru tedy vytvoříme vlastní routu. Více informací o této problematice lze nalézt na [14].

```
config.Routes.MapHttpRoute(
  name: "DefaultApiWithToken",
  routeTemplate: "api/{controller}/{token}/{action}/",
  defaults: new {  }
);
```

Kód 8 - Ukázka vlastní routy

Zdroj: vlastní zpracování

Každý záznam musí obsahovat jedinečné jméno v celé kolekci rout. Kód výše definuje strukturu URL volaných služeb a obstará nám obhospodaření všech metod, které obsahují parametr *token*, kdy:

- {controler} – definuje název controlleru (v našem případě se jedná o kontroler *Service*),

[21] Přátelská URL – je webová adresa, která je snadno čitelná pro uživatele a obsahuje slova popisující obsah (službu) webové stránky

- {token} – definuje hodnotu proměnné, kterou předává do příslušných metod pod parametrem token
- {action} – definuje název metody v controlleru (např. *GetFurlough*)

Reálný příklad volané URL je:

http://localhost:58761/api/Service/af855cbd1cdc67e45dcd8047cd6d3be123b454107d5239396156905ab8f6117c/GetFurlough

Změnu routování můžeme provádět i na úrovni metody v controlleru pomocí Data Annotations. Tento způsob je nazýván Attribude Routing a je k dispozici od verze MVC 5. Je vhodný na definování výjimek. Tento způsob zápisu využijeme u metody Auth, která neobsahuje parametr token a tím neplní routovací podmínku výše. Pomocí data atributu [Route(„api/Service/Auth“)] přiřadíme metodě identifikátor URL.

```
/// <summary>
/// Metoda, která slouží k authorizaci uživatele
/// </summary>
/// <param name="auth">Modelová třída pro authorizaci</param>
/// <returns>Vrací JSON objekt s tokenem, pro následnou komunikaci</returns>
[HttpPost]
[Route("api/Service/Auth")]
[ResponseType(typeof(JSONReturnAuth))]
public HttpResponseMessage Auth([FromBody] AuthorizationViewModel auth)
 {
```

Kód 9 - Ukázka změny routování pomocí data annotations

Zdroj: vlastní zpracování

5.1.6. HelpPage – dokumentace k API

Při vytváření webové aplikace s API je užitečné vytvořit i stránku s dokumentací, aby ostatní vývojáři věděli, jak volat příslušné API. Tuto dokumentaci můžete vytvořit ručně jako statickou stránku HTML, nebo využít automatické generování dokumentace přímo z aplikace. K tomu ASP.NET Web API poskytuje knihovnu pro automatické vytváření dokumentace při běhu aplikace. Tato knihovna se nazývá **HelpPage** a do projektu jí můžeme přidat pomocí správce balíčku NuGet. Po úspěšném nainstalování balíčku si ale musíme dát

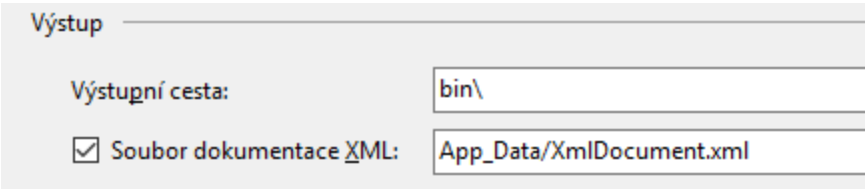
pozor na to, aby v souboru *Global.asax* byl následující kód, který registruje všechny oblasti a inicializuje jej do projektu.

```
protected void Application_Start()
{
       AreaRegistration.RegisterAllAreas();
       …
}
```

Kód 10 - Registrace oblastí potřebné k inicializaci HelpPage knihovny

Zdroj: vlastní zpracování

Po přidání výše uvedeného kódu do aplikace je nutné nastavit generování dokumentace do XML souboru. To provedeme ve vlastnostech projektu v sekci *sestavení* a v části *výstup* nastavíme zaškrtávací tlačítko u položky „Soubor dokumentace XML“ a zapíšeme cestu k souboru „*App_Data/XmlDocument.xml*“. Tuto cestu zkontrolujeme nebo změníme i v souboru konfigurace knihovny HelpPage v souboru *Areas/HelpPage/App_Start/HelpPageConfig.cs*.

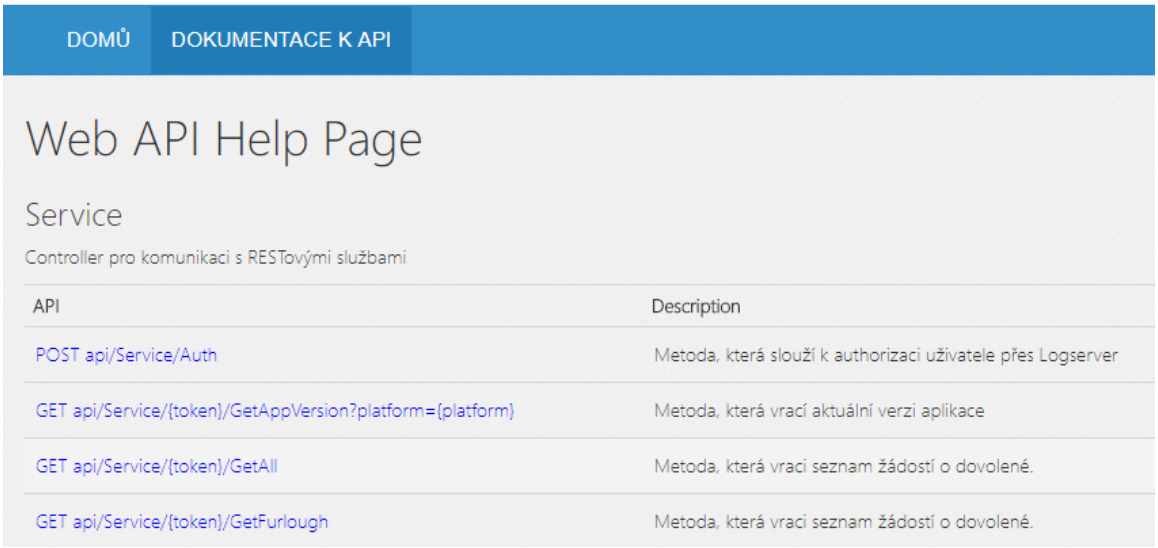


Obrázek 16 - Nastavení výstupního souboru dokumentace

Zdroj: vlastní zpracování

Po úspěšném nastavení všech cest, můžeme dle potřeb upravit výstupní vzhled dokumentace ve sdíleném souboru *Areas/HelpPage/Views/Help/Index.cshtml*.

Bakalářská práce
Martin Hůlek
schvalování dokumentů



Obrázek 17 - Ukázka výsledné HelpPage dokumentace

Zdroj: vlastní zpracování

5.2. Vývoj klientské android aplikace

V této kapitole si představíme vlastní vývoj klientské aplikace pro nativní systém Android pomocí frameworku Xamarin v programovacím jazyku C#. Vývoj vychází z analýzy, která byla vysvětlena v kapitole 4. Na začátku vývoje nastavíme verzi Android API, ve které budeme vyvíjet aplikaci. Toto nastavení provedeme ve vlastnostech projektu nebo v souboru s názvem *AndroidManifest.xml*. Tato aplikace je cílena na Android 5.1 (Lollipop), která je definována verzí API 22. Po zvolení cílové architektury je nutné nastavit (specifikovat) oprávnění, která bude aplikace požadovat vůči zařízení na kterém bude nainstalována. Ta bude využívat přístup k internetu, z toho důvodu nastavíme požadované oprávnění *INTERNET* a *ACCESS_NETWORK_STATE*. V neposlední řadě nastavíme název aplikace, název balíčku, ikonu a preferované úložiště.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="cs.hulek.appropoval" android:versionCode="1" android:versionName="1.3"
android:installLocation="internalOnly">
  <uses-sdk android:minSdkVersion="10" android:targetSdkVersion="22" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
  <application android:label="Bakalářská práce Hůlek Martin"
android:icon="@drawable/Icon"></application>
</manifest>
```

Kód 11 - Ukázka souboru AndroidManifest.xml

Zdroj: vlastní zpracování

5.2.1. Přihlášení uživatele

Po nastavení prostředí můžeme přejít k vývoji první aktivity, která je po spuštění aplikace volána. Jedná se o aktivitu pod názvem *MainActivity.cs*. V této aktivitě se ověřuje identita uživatele, zda-li je uživatel přihlášen či nikoliv. Z toho důvodu, tato aktivita bude přepínat mezi dvěma layouty. V případě, že uživatel není přihlášen, aktivita zobrazí layout tvořený logem aplikace, textovými prvky pro zadání uživatelského jména a hesla a tlačítkem na přihlášení. Na toto tlačítko nastavíme událost, ve které zavoláme naše webové API a pokusíme se ověřit uživatele.

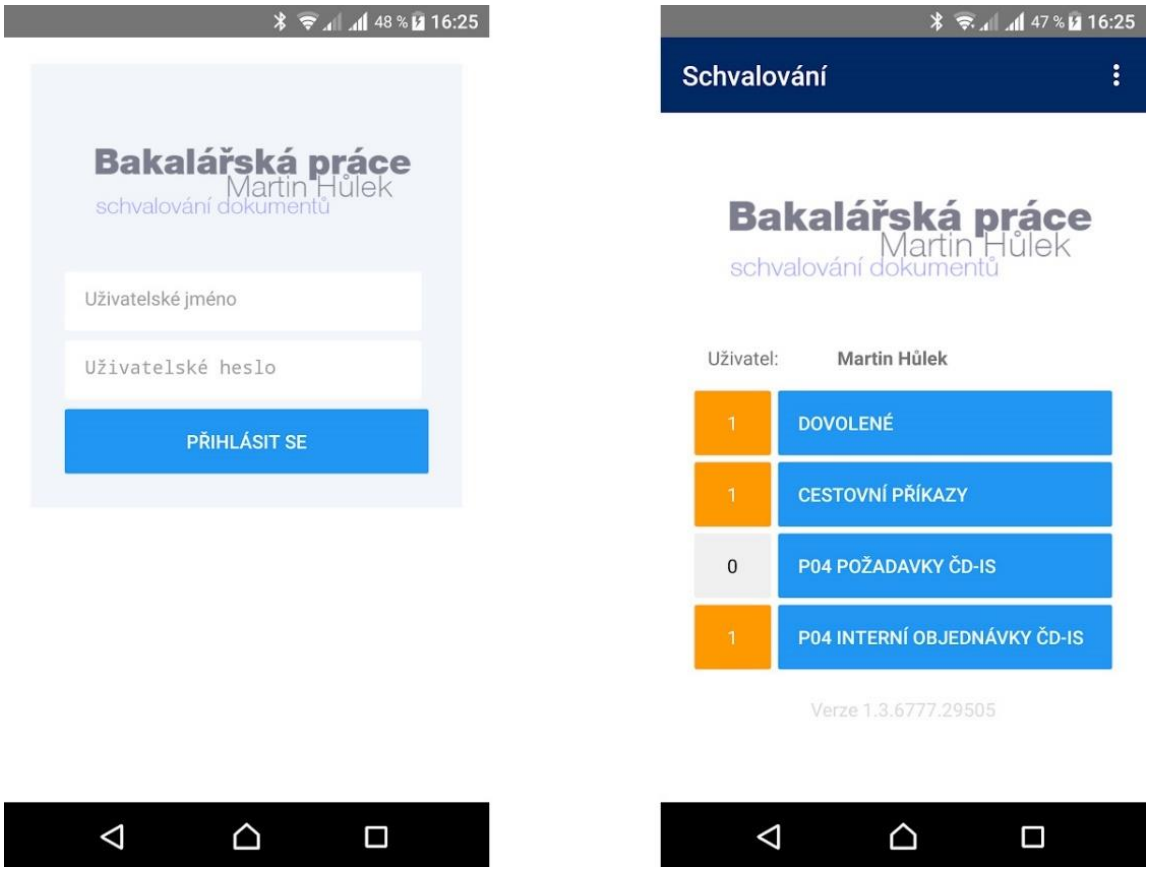
```
SetContentView(Resource.Layout.MainLogin);

Button buttonLogin = FindViewById<Button>(Resource.Id.buttonLogin);
buttonLogin.Click -= buttonLogin_Click;
buttonLogin.Click += buttonLogin_Click;
```

Kód 12 - Část souboru MainActivity.cs, která nastavuje událost na tlačítko a layout s přihlašovacím formulářem

Zdroj: vlastní zpracování

V případě, že se podařila ověřit totožnost uživatele, povolíme uživateli vstup do aplikace tím, že mu zobrazíme druhý layout s úvodní obrazovkou. V horní části je zobrazeno logo, pod ním jméno přihlášeného uživatele a 4 tlačítka spolu s počtem neschválených záznamů pro daný typ dokumentu. V současné době se vykoná dotaz na pozadí, který zjistí aktuální vydanou verzi aplikace. V případě, že aplikace je zastaralá, zobrazí informační okno s informací o nutnosti aktualizace aplikace.



Obrázek 18 - Printscreen aplikace s přihlašovacím formulářem a úvodní obrazovka po úspěšné autorizaci

Zdroj: vlastní zpracování

Kliknutím na jedno ze čtyř tlačítek, aplikace spustí novou aktivitu, která zobrazuje seznam neschválených záznamů, který si popíšeme v následující podkapitole.

```
Button buttonTravelOrder =
FindViewById<Button>(Resource.Id.buttonTravelOrder);
buttonTravelOrder.Click += (object sender, EventArgs e) =>
{
       var activity = new Intent(this, typeof(TravelOrderActivity));
         StartActivity(activity);
};
```

Kód 13 - Spuštění nové aktivity pro schvalování cestovních příkazů

Zdroj: vlastní zpracování

5.2.2. Seznam záznamů

V této podkapitole si popíšeme zpracování několika záznamů a vykreslení pomocí komponenty *ListView*[22]. Komponenta zobrazuje rolovací seznam položek, které se vkládají pomocí tzv. adaptéru. *Adaptér*[23] je jakýsi prostředník mezi komponentou *ListView* a zobrazovanými daty, kde je zodpovědný za vytváření *View*[24] pro jednotlivé záznamy. *View* je základní stavební blok pro vykreslení uživatelského rozhraní.

Na začátek vytvoříme nový layout, který bude obsahovat komponentu *ListView* a tlačítka pro hromadné schválení/zamítnutí záznamů, ten pojmenujeme *ListLayout*. Tento layout bude společný pro všechny aktivity starající se o jiné typy dokumentů, ale každá aktivita bude implementovat jiný adaptér. Následně vytvoříme 4 nové layouty, každý bude obsahovat jiný vzhled pro konkrétní typ dokumentu. Názvy souborů budeme pojmenovávat podle označení dokumentu a příznakem „ListView_item“. Ke každému z těchto layoutů je potřeba vytvořit i již zmíněný *Adaptér*, který jej bude ovládat a plnit konkrétními daty. Jedná se o třídu, která bude dědit z třídy *Android.Widget.BaseAdapter* s referencí na modelovou třídu.

```
class TravelOrderListViewAdapter : BaseAdapter<TravelOrder> {
     private List<TravelOrder> mItems;
     private Context mContext;
     private Dictionary<int, bool> checkedItems = new Dictionary<int, bool>();
     public List<TravelOrder> CheckedItems[…]
     public TravelOrderListViewAdapter(Context context, List<TravelOrder>
items) […]
     public override long GetItemId(int position) […]
     public override TravelOrder this[int position] […]
     public override View GetView(int position, View convertView, ViewGroup
parent) […]
     public override int Count […]
     public void ToggleClick(int position) […]
}
```

Kód 14 - Ukázka adaptéru pro výpis položek cestovních příkazů

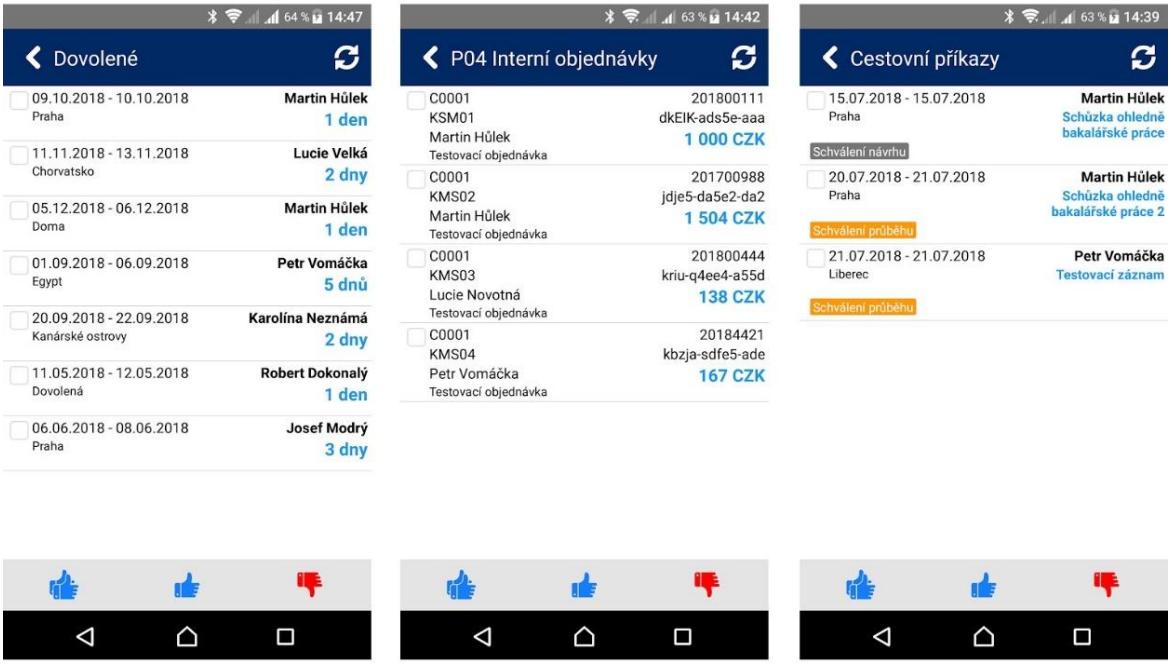
Zdroj: vlastní zpracování

Při vytváření adaptéru je nutné implementovat několik metod: *GetItemId*, *this*, *Count a GetView*. První tři jsou triviální k implementaci, většinou pracují s polem a slouží pro

[22] ListView dokumentace dostupná z *https://developer.android.com/guide/topics/ui/layout/listview*
[23] Adaptér dokumentace dostupná z *https://developer.android.com/reference/android/widget/Adapter*
[24] View dokumentace dostupná z *https://developer.android.com/reference/android/view/View*

výběr konkrétní položky ze seznamu. Naopak metoda *GetView* se stará o generování již zmíněného *View* a propojení atributů z modelové třídy s komponenty daného *View*.

Na závěr vytvoříme čtyři různé aktivity, které budeme spouštět po kliku na tlačítku z úvodní obrazovky. Každá aktivita bude implementovat adaptér patřící k danému typu dokumentu. Aktivity dále budou obsahovat i výkonný kód pro obsluhu tlačítek ke schválení a zamítnutí označených záznamů, obsluhu kliku na daný záznam, který spustí novou aktivitu s detailem záznamu a v neposlední řadě i kód pro opakované načítání záznamů.



Obrázek 19 - Printscreen z aplikace seznamu záznamů pro Dovolenky, P04 Interní objednávky a Cestovní příkazy

Zdroj: vlastní zpracování

6. Výsledek a závěr

Cílem této bakalářské práce bylo navrhnout a vyvinout mobilní aplikaci pro schvalování dokumentů z různých interních systémů. Pro vývoj mobilní aplikace byl použit nástroj Xamarin. K úspěšné implementaci této aplikace bylo zapotřebí vyvinout webové API, které sdružuje interní systémy do jednoho komunikačního API. Sdružení systémů nám poskytuje větší bezpečnost při komunikaci s interními systémy, kdy s nimi komunikuje pouze jeden server, a proto tyto interní systémy mohou být umístěny v demilitarizované zóně, a tím odstíněny od případného útoku z prostředí internetu. API bylo vyvinuto pomocí frameworku .NET s využitím REST architektury. Tato architektura je v současné době velmi populární hlavně kvůli její rychlosti, jednoduchosti, efektivitě a rozšiřitelnosti.

Tato práce byla v prvních částí zaměřena na představení nástroje Xamarin, kde jsme se dozvěděli jeho historii a podstatu tohoto nástroje, operačního systému Android a vývoj nativních aplikací pro operační systém Android. Nemalá teoretická část se pak věnovala již zmíněné REST architektuře, kde byly popsány její hlavní rysy, příklady a výhody či nevýhody použití této architektury. Praktická část se zabývala analytickým vývojem aplikace, kde byly představeny požadavky na aplikaci a byl proveden její návrh, na jehož základě probíhal konečný vývoj aplikací.

Obě aplikace (mobilní aplikaci a webové API) se úspěšně podařilo vyvinout a v současné době jsou nasazeny v nejmenované společnosti. Aplikace je aktivně využívána zaměstnanci společnosti a do budoucna se předpokládá její rozšíření o integraci dalších interních systémů a vytvoření mobilní aplikace pro platformu iOS.

Seznam zdrojů

[1] BALASUBRAMANIAN, CARLYLE, ERL and PAUTASSO: *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutins with REST* [online]. WhatisREST.com. 2012 [cit.2017-08-19] ISBN: 0137012519. Dostupné z: http://whatisrest.com/rest_constraints/client_server

[2] BURNS, Amy: *Přehled sdílení kódů* [online], Microsoft, 23. 3. 2017, [cit. 2018-07-15]. Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/cross-platform/app-fundamentals/code-sharing#Net_Standard

[3] BURNS, Amy: *Úvod do Xamarin* [online], Microsoft, 28. 3. 2017, [cit. 2018-07-15]. Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/cross-platform/get-started/introduction-to-mobile-development#Introduction_to_Xamarin

[4] ČÁPKA, David: *Lekce 2 – UML – Use Case Diagram* [online], ITnetwork.cz, [cit. 2018-07-12]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-use-case-diagram

[5] ČÁPKA, David: *Lekce 3 – UML – Use Case Specifikace* [online], ITnetwork.cz, [cit. 2018-07-10]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-use-case-specifikace-diagram

[6] ČÁPKA, David*: Lekce 4 - UML - Doménový model* [online], ITnetwork.cz, [cit. 2018-07-13]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-domenovy-model-diagram

[7] ČÁPKA, David: *Lekce 5 – UML – Class diagram* [online], ITnetwork.cz, [cit. 2018-07-15]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-class-diagram-tridni-model

[8] ČÁPKA, David: *Singleton (jedináček)* [online], ITnetwork.cz, [cit. 2018-07-08]. Dostupné z: https://www.itnetwork.cz/navrh/navrhove-vzory/gof/gof-vzory-pro-vytvareni/singleton-navrhovy-vzor/

[9] FIELDING, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures* [online], 2000 [cit. 2017-08-18]. Dissertation's thesis. University of California, Irvine. Chair of the Disertation Committee: Professor Richard N. Taylor. Dostupné z http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[10] FISHER, Roman: Xamarin: *Úvod do Xamarin Forms* [online], Skeleton, 1. 3. 2017, [cit. 2018-07-14]. Dostupné z: https://www.skeleton.cz/uvod-do-xamarin-forms

[11] GOOGLE DEVELOPERS, Android: *Activity* [online], 6. 6. 2018, [cit. 2018-07-15]. Dostupné z: https://developer.android.com/reference/android/app/Activity

[12] GOOGLE DEVELOPERS, Android: *Android.app* [online], 6. 6. 2018, [cit. 2018-07-15]. Dostupné z: https://developer.android.com/reference/android/app/package-summary

[13] GOOGLE DEVELOPERS, Android: *The Android Source Code* [online], 14. 11. 2017, [cit. 2018-07-10]. Dostupné z: https://developer.android.com/reference/android/app/Activity

[14] HOLEC, Miroslav: *MVC Routing – Úvod* [online], 8. 9. 2014, [cit. 2018-06-25]. Dostupné z: https://www.miroslavholec.cz/blog/mvc-routing-uvod

[15] HOUŠKA, Petr: *Začátky s Androidem aneb Co to vlastně je?* [online], Android Market, 24. 12. 2012, [cit. 2018-06-20]. Dostupné z: http://androidmarket.cz/ruzne/zacatky-s-androidem-aneb-co-to-vlastne-je/

[16] KARCH, Marziah: *What Is Google Andoid?* [online], Lifewire, 2. 10. 2017, [cit. 2018-06-18]. Dostupné z: https://www.lifewire.com/what-is-google-android-1616887

[17] KONEČNÝ, Matěj: *Vyvíjíme pro Android: První krůčky* [online], 22. 6. 2012, Zdroják.cz, [cit. 2018-06-15]. Dostupné z: https://www.zdrojak.cz/clanky/vyvijime-pro-android-prvni-krucky/

[18] KONEČNÝ, Matěj: *Vyvíjíme pro Android: Začínáme* [online], 15. 6. 2012, Zdroják.cz, [cit. 2018-06-15]. Dostupné z: https://www.zdrojak.cz/clanky/vyvijime-pro-android-zaciname/

[19] MÁDR, Vojtěch: *Xamarin: Představujeme nástroj pro multiplatformní vývoj mobilních aplikací (díl 1)* [online], Eman, 29. 2. 2016, [cit. 2018-07-19]. Dostupné z: https://www.eman.cz/blog/xamarin-predstavujeme-nastroj-pro-multiplatformni-vyvoj-mobilnich-aplikaci-dil-1/

[20] MALÝ, Martin: *REST: architektura pro webové API* [online], 3.8.2009, Zdroják.cz, [cit. 2018-03-11]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

[21] MANAGMENTMANIA: *Nativní aplikace* [online], 5. 4. 2017, Wilmington, [cit. 2018-07-11]. Dostupné z: https://managementmania.com/cs/nativni-aplikace-native-application

[22] MCLEMORE, Mark: *Android prostředky* [online], Microsoft, 1. 2. 2018, [cit. 2018-07-12]. Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/resources-in-android/?tabs=vswin

[23] MCLEMORE, Mark: *Principy úrovní rozhraní API systému Android* [online], Microsoft, 2. 7. 2018, [cit. 2018-07-08]. Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/android-api-levels?tabs=vswin

[24] MCLEMORE, Mark: *Životní cyklus aktivity* [online], Microsoft, 1. 2. 2018, [cit. 2018-07-12]. Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/android/app-fundamentals/activity-lifecycle/

[25] MICROSOFT DEVELOPERS: *Visual Studio 2015 a Xamarin* [online], Microsoft, 2018, [cit. 2018-07-20]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/mt299001.aspx

[26] MICROSOFT: *Chapter 11: Server-Side Implementation část* [online], 28 . 8. 2012, [cit. 2018-06-30]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh404093(v=pandp.10)

[27] MONTEGRIFFO, Nicholas: *What is an APK file and how do you install one?* [online], Androidpit, 22. 1. 2018, [cit. 2018-07-15]. Dostupné z: https://www.androidpit.com/android-for-beginners-what-is-an-apk-file

[28] PAŽOUREK, Tomáš: *Webové služby v architektuře REST na platformě .NET*, Brno, 2012. Bakalářská práce. Masarykova Univerzita: Fakulta informatiky. Vedoucí práce Mgr. Filip Jurnečka. Dostupné z: https://is.muni.cz/th/u2ra1/thesis.pdf

[29] READMEBLOG: *The History of REST APIs* [online], 15.10.2016 [cit. 2017-08-18]. Dostupné z: https://blog.readme.io/the-history-of-rest-apis/.

[30] ROUSE, Margaret: *Native app* [online], březen 2018, Techtarget, [cit. 2018-07-12]. Dostupné z: https://searchsoftwarequality.techtarget.com/definition/native-application-native-app

[31] SETHUMADHAVAN, Arun: *What is Xamarin? – Part 2* [online], Xamarin.Android, 23. 6. 2014, [cit. 2018-07-05]. Dostupné z: http://xamarinandroid.blogspot.com/2014/06/what-is-xamarin-part-i.html

[32] SCHMIDL, Tomáš: *Návrh a implementace RESTových rozhraní*, Brno, 2016. Bakalářská práce. Masarykova Univerzita: Fakulta informatiky. Vedoucí práce: doc. RNDr. Petr Sojka, Ph.D.. Dostupné z: https://is.muni.cz/th/uo42j/navrh-a-implementace-restovych-rozhrani.pdf

[33] SCHMIDT, Cory: *What is Android? Here is a complete quide for beginners* [online], Androidpit, 24. 6. 2016, [cit. 2018-06-19]. Dostupné z: https://www.androidpit.com/what-is-android

[34] ŠTĚPÁNEK, Jiří: *ASP.net MVC tutorial 5 - Přístup k datům* [online], YouTube, 25. 2. 2015, [cit. 2018-02-13]. Dostupné z: https://www.youtube.com/watch?v=KFpVxndVEh0

[35] TECHTERMS: *Friendly URL Definition* [online], 11. 2. 2011, [cit. 2018-07-18]. Dostupné z: https://techterms.com/definition/friendly_url

[36] THIJSSEN, Josua: *Caching your REST API* [online], Restcookbook.com [cit. 2017-08-20]. Dostupné z: http://restcookbook.com/Basics/caching/.

[37] VERMA, Sanja: *APIs versus web services* [online], Mulesoft, 18. 1. 2018, [cit. 2018-07-13]. Dostupné z: https://blogs.mulesoft.com/dev/api-dev/apis-versus-web-services/

[38] WASSON, Mike: *Vytváření stránek nápovědy pro rozhraní ASP.NET Web API* [online], Micerosoft, 1. 4. 2013, [cit. 2018-07-19]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/web-api/overview/getting-started-with-aspnet-web-api/creating-api-help-pages

[39] WIGMORE, Ivy: *Resource-oriented architecture (ROA)* [online], Whatis.com, July 2012 [cit. 2017-08-19]. Dostupné z: http://whatis.techtarget.com/definition/resource-oriented-architecture-ROA.

[40] WIKIPEDIE: *CRUD* [online], 6. 1. 2015 [cit. 2017-08-19]. Dostupné z: https://cs.wikipedia.org/wiki/CRUD.

[41] WIKIPEDIE: *Singleton* [online], 11. 2. 2018 [cit. 2018-07-05]. Dostupné z: https://cs.wikipedia.org/wiki/Singleton

[42] WIKIPEDIE: *Webová služba* [online], 4. 10. 2017 [cit. 2018-08-2]. Dostupné z: https://cs.wikipedia.org/wiki/Webov%C3%A1_slu%C5%BEba.

[43] WIKIPEDIE: *Xamarin* [online], 2. 6. 2018 [cit. 2018-08-3]. Dostupné z: https://en.wikipedia.org/wiki/Xamarin.

[44] ZIMMEROVÁ, B.: *Analytický model tříd – 1. část* [online], PV167 Projekt z obj. návrhu IS, 26.3.2008, [cit. 2018-07-05]. Dostupné z: https://www.fi.muni.cz/~buhnova/PV167/06_AnalytickyModelTrid_I.pdf

Seznam obrázků

Seznam tabulek

Seznam kódů

Přílohy

1) Příloha 1 – Struktura přiloženého DVD

Adresářová struktura přiloženého DVD:

- **Bakalarska-prace**
  - **Bakalarska-prace-martin-hulek.pdf** – text bakalářské práce
- **Databaze**
  - **Struktura.sql** – script pro vytvoření struktur databáze
- **MobileApp –** kompletní projekt

Zadání práce

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2016/2017

Studijní program: Aplikovaná informatika
Forma: Kombinovaná
Obor/komb.: Aplikovaná informatika (ai3-k)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Hůlek Martin	Pod Velbabou 1044, Úpice	I14503

TÉMA ČESKY:

Vývoj mobilní aplikace pro schvalování dokumentů v prostředí Xamarin

TÉMA ANGLICKY:

Development of mobile application for approval of documents in the Xamarin

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl:
Cílem je návrh a vývoj mobilní aplikace pro schvalování dokumentů z různých interních systémů s využitím technologie Xamarin.

Osnova:
1. Úvod
2. Xamarin a webové služby
3. Systém Android, nativní aplikace
4. Analýza a návrh řešení
5. Vlastní vývoj řešení
6. Výsledky, závěr
7. Literatura

SEZNAM DOPORUČENÉ LITERATURY:

Google Scholar

Podpis studenta: ..... **Datum:** .....

Podpis vedoucího práce: ..... **Datum:** .....

(c) IS/STAG , Portál - Podklad kvalifikační práce , I14503 , 20.10.2016 08:50