UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2019 Milan Suchomel

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Mobilní klientská aplikace pro geosociální sítě

Milan Suchomel

Diplomová práce

2019

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2018/2019**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Milan Suchomel**
Osobní číslo:	**I17221**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Mobilní klientská aplikace pro geosociální sítě**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce bude v teoretické části charakterizovat a popsat geosociální sítě a analyzovat využitelnost jejich API pro vlastní mobilní aplikaci. Součástí práce bude i přehled rozhraní API geosociálních sítí, identifikace jejich výhod a nevýhod. Druhým cílem v praktické části diplomové práce bude navrhnout a realizovat mobilní aplikaci (volitelně pro operační systém Android/iOS/Windows Mobile) či multiplatformní mobilní aplikaci v Adobe PhoneGap. Ta bude využívat API vybrané geosociální sítě a geolokační technologie pro lokalizaci uživatele a bude inovativním způsobem využívat získaná geodata v okolí uživatele.

Rozsah grafických prací:

Rozsah pracovní zprávy: **50-60**

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

Seznam odborné literatury:

KYSELA, J.: Comparison of Web Applications Geolocation Service. In: IEEE 15th International Symposium on Computational Intelligence and Informatics (CINTI 2014). Budapešť: Óbuda University, 2014. ISBN 978-1-4799-5338-7
KYSELA, J., HORÁLEK, J., HOLÍK, F.: Measuring Information Quality of Geosocial Networks. In: New Trends in Intelligent Information and Database Systems. Springer, 2015. ISBN 978-3-319-16211-9
PEACOCK, M.: PHP 5 Social Networking. Packt Publishing, 2010. ISBN: 978-1-849512-38-1

Vedoucí diplomové práce: **Ing. Jiří Kysela, Ph.D.**
Katedra informačních technologií

Datum zadání diplomové práce: **22. října 2018**
Termín odevzdání diplomové práce: **18. května 2019**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 17. listopadu 2018

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 15. 5. 2019

Milan Suchomel

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat svému vedoucímu Ing. Jiřímu Kyselovi, Ph.D. za odborné vedení mé práce. Chtěl bych také poděkovat své přítelkyni, rodině, přátelům a hlavně skupině Ninja Arena za velkou podporu při studiu.

ANOTACE

Cílem této diplomové práce je vytvoření mobilní aplikace pro mobilní platformu Android, která bude využívat otevřených API vybraných geosociálních sítí a zobrazovat tak události či zajímavá místa v okolí uživatele.

KLÍČOVÁ SLOVA

Android, Kotlin, geosociální síť, mobilní aplikace, otevřené API

TITLE

A Mobile Geosocial Networking Client Application

ANNOTATION

The aim of this thesis is to create a mobile application for the Android mobile platform, which will use open APIs of selected geosocial networks to show events or places of interest around the user. The aim of this thesis is to create a mobile application for the Android mobile platform, which will use open APIs of selected geosocial networks to show events or places of interest around the user.

KEYWORDS

Android, Kotlin, geosocial network, mobile application, open API

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM GRAFŮ

SEZNAM TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK

API	Application Programming Interface
ART	Android Runtime
CSS	Cascading Style Sheet
GCM	Google Cloud Messaging
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
MVVM	Model View ViewModel
NFC	Near Field Communication
QR	Quick Response
REST	Representational state transfer
RSS	Rich Site Summary
SD	Secure Digital
SDK	Software Development Kit
SIM	Subsciber Identity Module
SMS	Short Message Service
VoIP	Voice over Internet Protocol
WiFi	Wireless Fidelity

ÚVOD

V poslední době se staly mobilní zařízení velice rozšířené. Denně proto vzniká nespočet aplikací pro tuto platformu. A to je také cílem této diplomové práce, navrhnout a realizovat mobilní aplikaci pro operační systém Android. Tato aplikace bude využívat veřejná data určitých geosociálních sítí. Přesto, že je více možností vývoje pro mobilní platformy, pro tuto diplomovou práci byl vybrán vývoj nativní aplikace pro operační systém Android. Operační systém Android je nejrozšířenějším systémem v mobilních platformách a nejen proto byl vybrán jako zaměření této práce.

Diplomová práce je rozdělena na dvě části, a to na část teoretickou a praktickou. V teoretické části jsou nejprve vysvětleny pojmy spojené s geolokací. Poté je část věnována operačnímu systému Android a vývoji pro tuto platformu, představeny jsou také další možnosti vývoje pro mobilní zařízení. Poslední kapitola teoretické části se zabývá webovými API vybraných geosociálních sítí a jejich využitím.

Praktická část se bude zabývat vytvořením již zmíněné aplikace. Nejprve bude zpracována analýza, která se skládá z požadavků na aplikaci. Následně budou popsány některé z využitých technologií pro její vývoj. A nakonec bude představena realizace samotné aplikace, skládající se z popisu projektu a implementované funkcionality.

Výstupem práce je aplikace, jež využívá otevřená data různých geosociálních sítí. Cílem této práce je také čtenáře seznámit s možnostmi práce s těmito daty a s vývojem pro mobilní platformu celkově.

1 GEOLOKACE

Geolokace je metoda umožňující zjištění geografické polohy určitého objektu. K určení této polohy lze využít různé technologie. Použitá technologie má přímou závislost na přesnost získané polohy. Mezi zařízení poskytující geolokace patří například mobilní telefon, smartphone, notebook nebo stolní počítač. Výstupem geolokace může být zeměpisná šířka a výška pozičního systém GPS nebo již přeložená fyzická adresa. Geolokaci objektu lze získat z více zdrojů, jako jsou:

- GPS,
- Mobilní signál,
- IP adresa,
- WiFi.

Zdroj: [1], [2].

1.1 Využití

Geolokace již dnes má spoustu využití a další budou přibývat. Mezi nynější příklady využití geolokace patří například:

- Rozšířená navigace,
- lokace blízkých veřejných míst (restaurace, obchody, kina, bankomaty, památky atd.),
- zobrazení aktuální dopravní situace,
- zjišťování polohy osob či objektů,
- zvýšené zabezpečení (platebních karet, účtů atd.),
- hraní her ve skutečném světě.

Zdroj: [1], [2].

1.2 Geocoding

Přesto, že je k dispozici mnoho metod na označení konkrétního místa, lidé většinou používají popis určující místo pomocí názvu města, ulice a míst, které jsou pro člověka snadno zpracovatelné. Takto udávané pozice však nejsou vhodné pro použití v informatice, proto vznikly metody pro převod těchto popisů míst na určení místa v souřadnicovém systému.

Zdroj: [3].

1.2.1 Reverzní geocoding

Reverzní geokódování je procesem opačným než geocoding, souřadnice převádí do fyzických adres. Tím můžeme určit město či stát, ve kterém se daný uživatel nachází nebo dokonce ulici s číslem popisným. Takový údaj je mnohem čitelnější pro koncového uživatele.

Službu pro reverzní geokódování poskytuje například Google prostřednictvím veřejné API. Tato služba požaduje souřadnice a vrací nám údaje o nejbližší ulici, městu, čtvrti a státu.

Zdroj: [3].

1.3 Geotagging

Geotagging je přidávání geografických metadat k různým mediím jako jsou fotografie, videa, webové stránky, SMS, QR kódy nebo RSS zdroje. Tyto data obsahují obvykle zeměpisnou šířku a výšku, ale mohou obsahovat nadmořskou výšku, přesnost nebo také čas záznamu.

Zdroj: [4].

1.4 Geosociální síť

Geosociální síť je typ sociální sítě, ve které se využívají geografické principy, jako je geokódování a geotagging. Uživatelské informace o poloze nebo geolokace umožňuje sociálním sítím cílit obsah v závislosti na lokaci uživateli. Výsledkem je, že uživateli jsou zobrazovány místa, události nebo také i osoby, jež se nacházejí v okolí. Mezi geosociální sítě patří například Foursquare, Flickr, TripAdvisor a další. Některé budou blíže představeny v kapitole Přehled otevřených API.

Zdroj: [5], [6].

2 MOBILNÍ PLATFORMY

2.1 Vývoj pro mobilní platformy

2.1.1 Webové aplikace

Webové aplikace jsou aplikace běžící na webu a uloženy na vzdálených serverech. Tyto aplikace jsou pak spouštěny prostřednictvím webového prohlížeče.

Přestože v některých případech mohou být takovéto aplikace vhodné, mají spoustu úskalí. Prvním z problémů je, že k použití takovéto aplikace je nutný stálý přístup k internetu, protože nejsou uloženy na lokálním zařízení. Takováto aplikace je dosti omezená, nemá přístup k periferiím (fotoaparát, senzory atd.) a úložišti.

Zdroj: [7].

2.1.2 Hybridní aplikace napříč platformami

Hybridní aplikace pro více platforem jsou to samé jako aplikace webové s tím rozdílem, že jejich obsah je zabalen do nativního kontejneru. Aplikace se tak tváří jako aplikace nativní. Pro vývoj takovýchto aplikací se používá zejména HTML5, JavaScript a CSS.

Tento typ vývoje řeší neduhy webových aplikací. Tyto aplikace nutně nepotřebují přístup k internetu, a tím že se tváří jako nativní aplikace, mohou být vystavovány v aplikacích jako Google Play či App Store. Tím se zvyšuje jejich možnost k objevení a úměrně s tím roste i počet uživatelů. Nástroji pro tento typ vývoje jsou:

- Apache Cordova,
- Ionic,
- Adope PhoneGap.

Zdroj: [7].

2.1.3 Nativní aplikace

Nativní aplikací je myšlena aplikace vyvinutá pro určitou platformu a operační systém.

Aplikace pro iOS jsou napsány v jazyce Swift nebo Objective-C, naopak pro psaní aplikací pro Android využijeme Javu či Kotlin. Vývojáři vyvíjející aplikace pro iOS používají jako IDE Xcode a na druhé straně vývojáři pro Android používají Android Studio.

Zdroj: [7].

2.1.4 Nativní aplikace napříč platformami

Nástroje pro nativní aplikace napříč platformami umožňují napsat kód aplikace pouze jednou, který je poté přeložen do nativního kódu různých operačních systémů. Tím je zpřístupněna možnost vydání aplikace na více platformách. Nativní aplikace pro více platforem jsou skvělou kombinací mezi hybridními a nativními aplikacemi. Používanými technologiemi pro tento typ vývoje jsou například:

- React Native,
- Xamarin,
- Titanium.

Zdroj: [7].

2.2 Operační systém Android

2.2.1 Historie Androidu

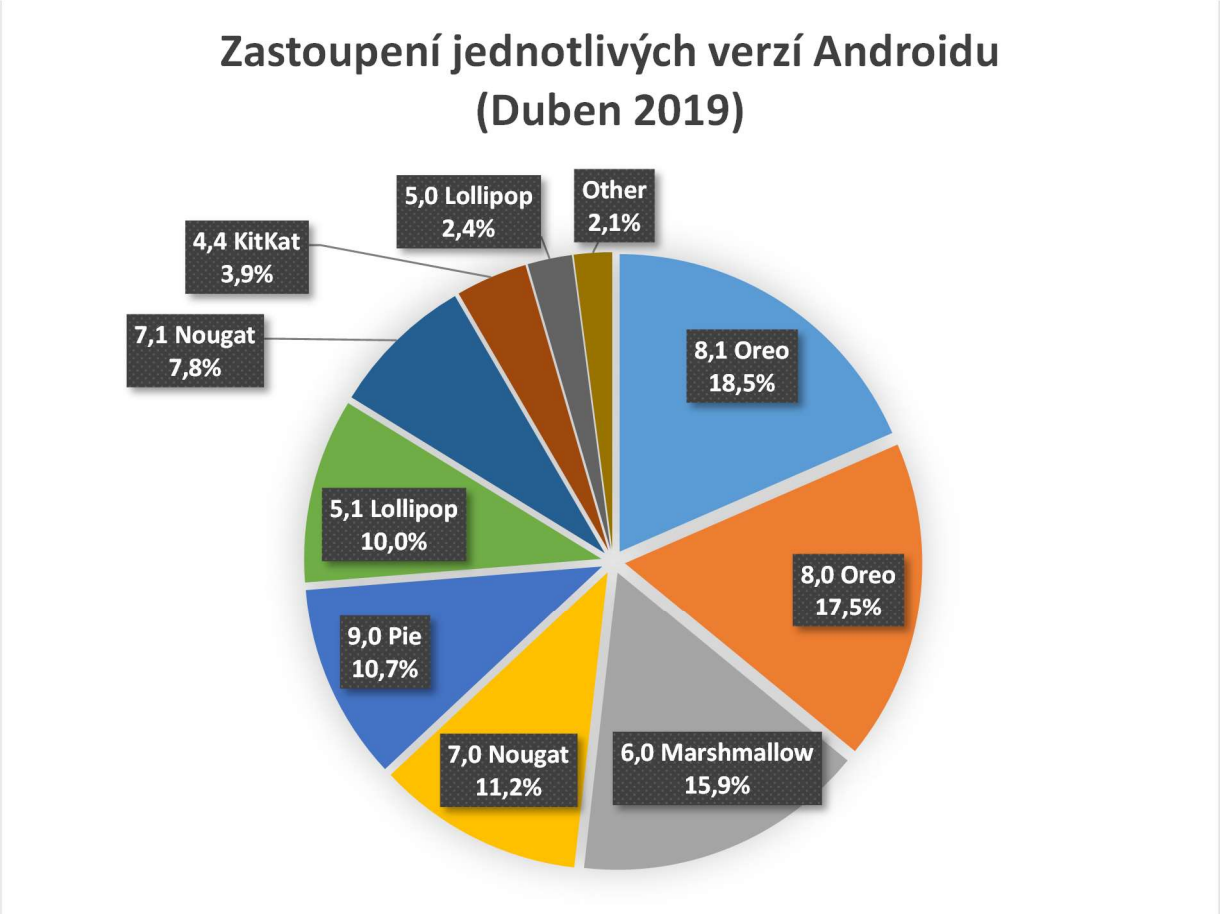
Android byl vyvinut společností Android Inc., která byla založena v říjnu roku 2003. Tento operační systém není vyvinut ani navržen společností Google. Avšak v srpnu roku 2005 byla tato společnost odkoupena společností Google. Výkupní cena byla 50 milionů dolarů, dnes má cenu vyšší o několik řádů. Za vývojem od roku 2007 stojí Open Handset Alliance. V tentýž rok vyšel také vývojářský kit SDK. V září 2008 přišel na trh první chytrý telefon s Androidem 1.0, byl to model HTC Dream (G1). V únoru 2009 byla vydána verze Android 1.1 jako aktualizace pro G1. V dubnu téhož roku byl vydán Android ve verzi 1.5, první veřejně přístupný Android.

Zdroj: [8].

2.2.2 Verze Androidu

Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwitch, Jelly Bean, KitKat, Lollipop, Marshmallow, Nougat, Oreo, Pie..., takto jsou označeny jednotlivé verze Androidu. Na první pohled je možné si všimnout předpisu pro pojmenovávání nových verzí, který Google používá. Jedná se o názvy cukrovinek a jsou řazeny dle abecedy. První dvě verze z části nedodržují tento předpis, jsou označeny jako Alpha a Beta.

V následujícím grafu jsou zobrazeny zastoupení jednotlivých verzí Androidu statistiky jsou z dubna roku 2019. Data jsou převzata z webu StatCounter, který je získává analýzou z více jak deset miliard přístupů do více jak dvou milionů stránek.



Graf 1 - Zastoupení verzí Androidu

V následující části kapitoly budou stručně představeny jednotlivé verze operačního systému Android. U každé verze budou zmíněné novinky, jež verze přinesla.

Android 1.5 Cupcake (30. 4. 2009)

Podpora softwarové klávesnice se slovníky, nahrávání a přehrávání videa, možnost kopírování a vkládání obsahu přes schránku, widgety na domovské obrazovce, animace přechodů mezi obrazovkami.

Android 1.6 Donut (1. 9. 2009)

Vylepšený Android Market (obrázky, hodnocení), práce s více soubory (galerie, mazání souborů), univerzální vyhledávač na domovské obrazovce, bezplatná navigace od Google, vylepšené hlasové vyhledávání, podpora displejů s větším rozlišením (WCGA), podpora VPN.

Android 2.0/2.1 Eclair (26. 10. 2009)

Podpora více velikostí rozlišení displeje, optimalizace výkonu, podpora Bluetooth 2.1, podpora HTML 5, animované tapety na domovské obrazovce, podpora Google Maps 3.x, podpora osvětlovací diody, nový design uživatelského prostředí.

Android 2.2 Froyo (20. 5. 2010)

Možnost instalovat aplikace na SD kartu, modem USB, možnost vytvoření Wi-Fi hotspotu, automatické aktualizace aplikací z Android Marketu, významná optimalizace celkového výkonu a využití paměti, vylepšené zálohování.

Android 2.3 Gingerbread (6. 12. 2010)

Zlepšena správa baterie, možnost zařízení s více fotoaparáty, NFC, podpora senzorů (gyroskop, barometr, …), VoIP, vylepšená softwarová klávesnice.

Android 3.0 Honeycomb (22. 2. 2011)

Optimalizace na tablety, vylepšené uživatelské prostředí, action bar, přizpůsobitelná domovská obrazovka, zjednodušené notifikace, hardwarová akcelerace.

Android 4.0 Ice Cream Sandwitch (19. 10. 2011)

Odstraňuje rozdíly mezi verzemi pro mobilní zařízení a verzemi pro tablet, aplikace na zamykací obrazovce, nedávno spuštěné aplikace, adresáře, zrušení notifikací, možnost odemykání pomocí tváře, převod hlasu na text, integrace kontaktů ze sociálních sítí.

Android 4.2 Jelly Bean (9. 7. 2012)

Možnost přepínání uživatelských účtů, vylepšené notifikace, GCM, rozpoznání hlasu i offline, zrychlení vykreslování obrazu.

Android 4.4 KitKat (3. 9. 2013)

Vyšší výkon, lepší podpora multitaskingu, režim fullscreen pro některé aplikace, podpora krokoměru, zlepšená podpora vícejádrových procesorů, infračervené ovládání, automatické vypínání procesů na pozadí.

Android 5.0/5.1 Lollipop (25. 6. 2014)

Material Design, ART, lepší správa baterie, více uživatelů na jednom zařízení (mód host), správa notifikací na zamykací obrazovce, funkce Smart Lock, podpora více SIM, lepší zvuk.

Android 6.0 Marshmallow (5. 10. 2015)

Oprávnění aplikací jsou uživatelem definovány. Prodloužení výdrže baterie, podpora USB-C, rozpoznání otisků prstů.

Android 7.0/7.1 Nougat (22. 8. 2016)

Možnost rozdělení obrazovky na dvě, nová API pro VR, notifikace při nadměrném vytížení určitou aplikací.

Android 8.0/8.1 Oreo (21. 8. 2017)

Zjištění notifikací po dlouhém kliknutí na ikonu, podpora více obrazovek, nativní podpora tisku, zrychlení spuštění zařízení, omezení aplikací běžících na pozadí.

Android 9.0 Pie (6. 8. 2018)

Upravená navigační lišta, větší množství informací v notifikacích, změna UIX, statistiky využívání telefonu a jeho aplikací, mód nerušit.

Zdroj: [8].

3 WEBOVÉ API

3.1 Otevřené API

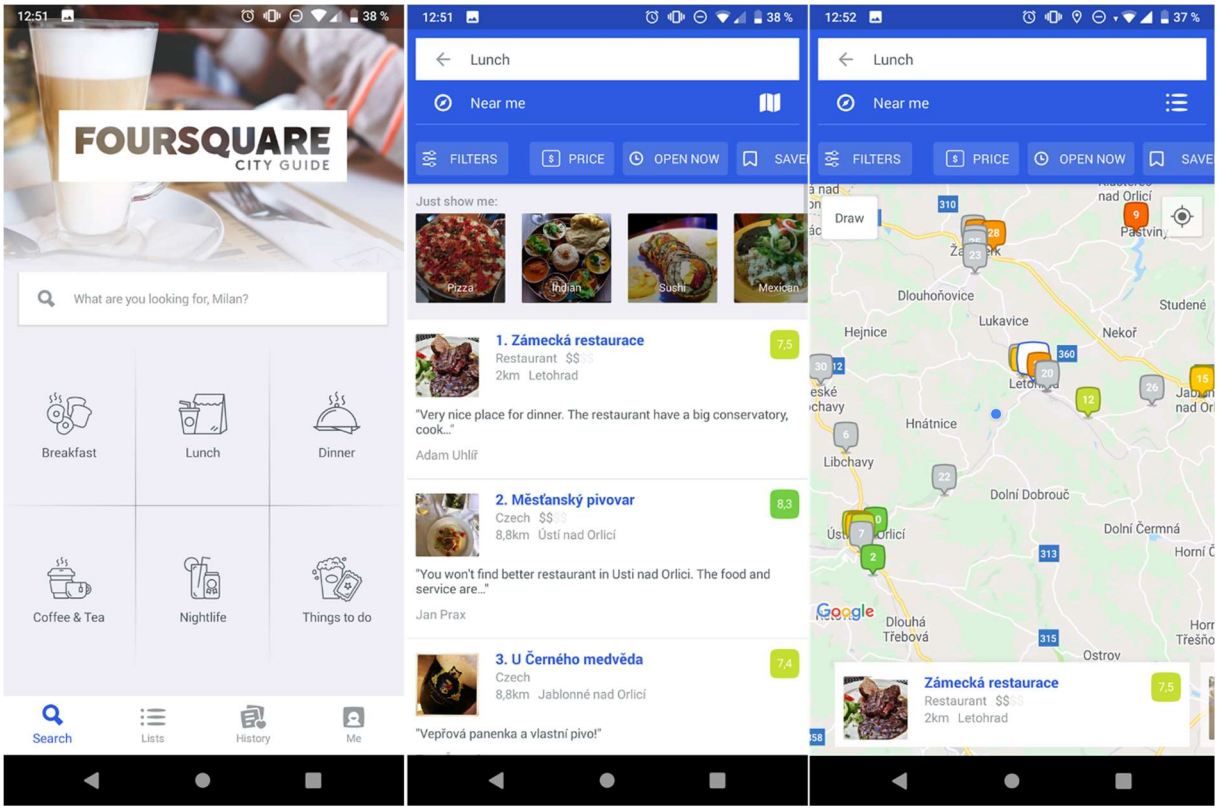
Otevřené API nebo také veřejné API, je veřejně dostupné aplikační programovací rozhraní, které poskytuje vývojářům přístup k softwarové aplikaci nebo webové službě.

3.2 Přehled otevřených API

V následující kapitole budou představeny některé z otevřených API, které obsahují informace o geolokaci k jednotlivým příspěvkům. U jednotlivých API bude představena mobilní aplikace, která ji využívá. Dále bude popsána dostupnost a případná cena za využití.

3.2.1 Foursquare

Foursquare je služba, která poskytuje vyhledávání míst v okolí jako jsou restaurace, noční podniky, obchody a další zajímavá míst. Jednotlivá místa jsou ohodnocena uživateli a jsou k nim připojeny fotky a tipy taktéž od uživatelů.



Obrázek 1 - Ukázka aplikace Foursquare

Přehled balíčků pro vývojáře

V následující tabulce jsou popsány jednotlivé balíčky služby Foursquare pro vývojáře. Obsahuje tři druhy balíčků. Prvním je balíček Personal, který je velmi omezený. Dalším je balíček Start-Up, který jak název napovídá, je určen pro začínající firmy s větší aplikací, který je měsíčně účtován a k tomu jsou placeny jednotlivé přístupy v přepočtu cca 13,5 tisíce Kč měsíčně plus jednotlivé přístupy. Posledním balíčkem je balíček Enterprise. Balíček je přizpůsobitelný požadavkům zákazníka a většina jeho vlastností je na dohodě. Podle toho je také účtován.

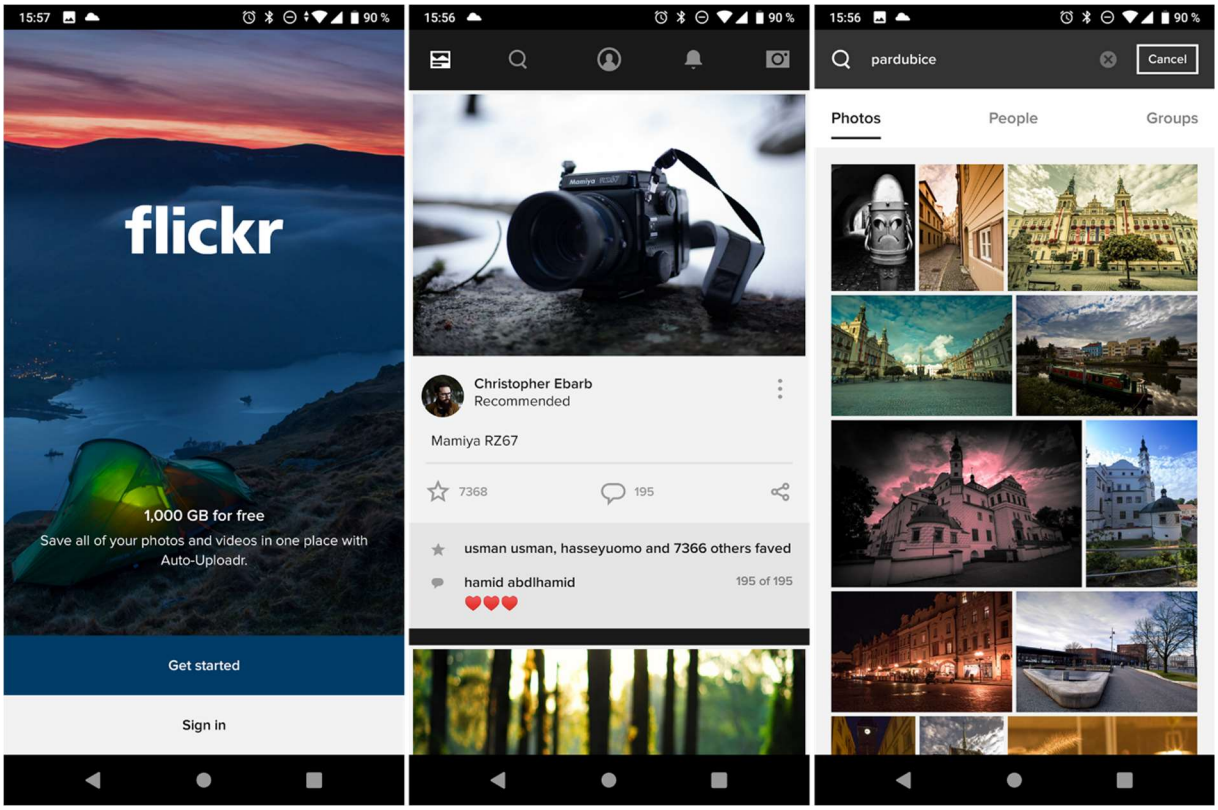
Tabulka 1 - Srovnání balíčku Foursquare

Srovnání	**Personal** **Non-Commercial**	**Start-Up** **Commercial**	**Enterprise** **Commercial**
Přístup k 105M místům	✔	✔	✔
Využití API	✔	✔	✔
Komerční využití	✘	✔	✔
Počet aplikací	1	Neomezený	Neomezený
Počet fotek k místu	2	4	Po dohodě
Počet tipů k místu	2	4	Po dohodě
Podpora	StackOverflow	Email do 72 hodin	Po dohodě
Běžné API volání / den	99 500	Neomezené ($0,001 / volání)	Zvýhodněné ceny
Prémiové API volání / den	500	Neomezené ($0,06 / volání)	Zvýhodněné ceny
Měsíční poplatek	ZDARMA	$599	Zvýhodněné ceny

Zdroj: [9], [10].

3.2.2 Flickr

Flickr je aplikace pro správu a sdílení fotografií po světě. Na Flickr uživatelé nahrávají fotky, které jsou doplněny o metadata, jako je geografické umístění, licence, lidé, značky a další. V mobilní aplikaci Flickr není jakýkoliv způsobem zpracována geolokace příspěvku.



Obrázek 2 - Ukázka aplikace Flickr

Přístupnost

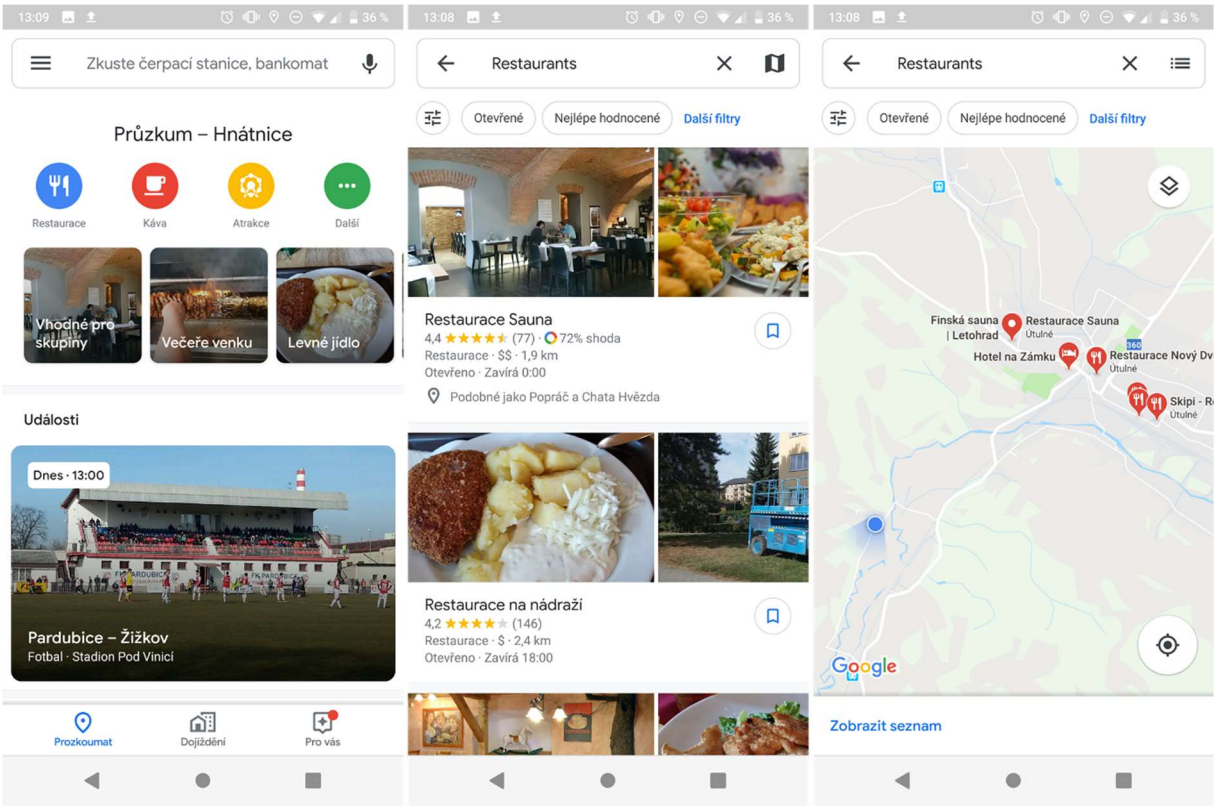
Hlavní výhodou je, že API je zdarma bez většího omezení. Pro autentizaci aplikace je zvolen protokol OAuth. Jsou zde také limity pro volání, které jsou nastaveny na 3600 volání za hodinu. Tento limit je nastaven kvůli předejití útokům na server. Po překročení tohoto limitu může být přístup zrušen nebo pozastaven.

Zdroj: [11].

3.2.3 Google Places

Google Places je otevřené API od společnosti Google shromažďující lokální firmy. Poskytuje informace o poloze, je tedy možno lokalizovat firmu na mapě. Nové firmy jsou vkládány převážně vlastníky firmy. Kromě umístění firmy obsahují další informace jako webové stránky, otevírací doby, fotky a další. Uživatelé mohou tyto místa hodnotit a komentovat. Tyto firmy jsou kategorizovány například jako restaurace, hotely, bary. Podle těchto kategorií je často vyhledávání filtrováno pro relevantnější informace.

Tato API nemá samostatnou oficiální aplikaci jako ostatní z uvedených, proto pro ukázku je představena aplikace Google Maps obsahující informace právě z této API.



Obrázek 3 - Ukázka aplikace Google Maps

Přístupnost

Google places poskytují následující sady volání:

- Place Search - vrátí seznam míst na základě umístění nebo filtru.
- Place Details - vrátí podrobnější informace o konkrétním místě.
- Place Photos - poskytuje přístup k milionům fotografií vázané k určitým místům.
- Place Autocomplete - doplňuje rozepsaný text místa či adresy.
- Query Autocomplete - poskytuje službu predikce dotazů pro textová vyhledávání.

Při vytvoření přístupu k této API je nutné vyplnit validní platební kartu. Přístup k této API je totiž placený, každý měsíc je však k dispozici kredit $200, který je dostačující pro menší aplikace. Jednotlivé dotazy jsou zpoplatněny dle jistých pravidel.

Zdroj: [12].

3.2.4 Porovnání otevřených API

Tato kapitola se zabývá porovnáním předešlých API, které jsou porovnávány dle ceny, přístupnosti, objemu dat a dalších hledisek.

Obsah těchto API není totožný, Flickr obsahuje příspěvky uživatelů, ke kterým je připojena lokace jako nepovinný atribut, naopak u druhých dvou je lokace základním prvkem dat. Google Places a Foursquare obsahují informace o místech, jako jsou například restaurace, jejich hodnocení, fotky a další. Tím je způsobeno, že počet příspěvků v následující tabulce nemusí být směrodatný.

Důležitým parametrem využitelnosti těchto API je cena, ta je závislá především na počtu dotazů. Flickr je zcela zdarma a to je jeho velká výhoda. Foursquare jak už bylo popsáno obsahuje balíčky, které mohou být placené. Mimo to dotazy na server jsou rozděleny do dvou typů dotazů a to na běžné a prémiové, dle toho jsou také pak zpoplatněny. Mezi běžné dotazy patří například vyhledání míst v okolí se základními informacemi. Naopak prémioví dotaz je nutný k získání podrobnějších informací, které obsahují například fotky a tipy k danému místu. Prémiový dotaz je šedesátkrát dražší než běžný dotaz. Google Places obsahuje komplexní platební plán, kde jsou zpoplatněny jak dotazy tak požadovaná data. Součástí dotazu mohou být parametry ovlivňující výstup, a proto jsou zpoplatněny jak dotazy, tak jeho data. Mezi zpoplatněná data patří například kontaktní údaje. Ačkoliv API Google Places je zpoplatněna již od prvního použití, lze ho využívat zdarma díky jeho měsíčnímu kreditu. V nastavení lze určit cenu, jakou může aplikace či jeden uživatel spotřebovat.

Endpointy jsou adresy, jež jsou využity pro přístup k datům. Jejich počet může vypovídat o rozmanitosti dostupných dat. V následující tabulce jsou děleny dle nutnosti autentizace, která je prováděna pomocí protokolu OAuth, který se stará o identifikaci uživatele v rámci komunikace. Autentizace začíná přihlášením uživatele, při kterém je získán autorizační token, který je součástí další komunikace.

V následující tabulce jsou jednotlivé API porovnány podle několika parametrů.

Tabulka 2 - Porovnání jednotlivých API

Porovnnání API		Flickr	Google Places	Foursquare
Zcela zdarma		✔	✘	✘
Počet balíčků		✘	✘	3
Bez omezení dat *		✔	✔	✘ **
Stránkování odpovědí		✔	✘	✘
Endpoints	Bez autentizace	96	7	19
	S autentizací	126	0	28
	Celkově	222	7	47
Počet příspěvků ***		1243	130	208

* data jako jsou fotografie či komentáře od uživatelů
** data u balíčku Enterprise nemusí být omezeny po dohodě
*** dle autorova průzkumu

V posledním řádku předešlé tabulky je uvedeno, že hodnoty jsou získány z autorova průzkumu. Tímto průzkumem je myšleno získání příspěvků z několika míst v okolí Pardubic pomocí jednotlivých API. Hodnota pak určuje počet těchto příspěvků. Tento počet je pouze orientační a slouží pouze k přibližné představě o množství dat.

Pro samotného vývojáře je také velmi důležitou součástí API dokumentace. Takovéto dokumentace jsou veřejně přístupné, a proto by měly být vytvořeny přehledně a měla by být popsána kompletní dostupná funkcionalita. Všechny dokumentace zmíněných API jsou komplexní a přehledné. Nejlepší dokumentací dle autora je dokumentace od API Foursquare, kde jsou parametry dotazu přehledně popsány v tabulkách. Součástí této dokumentace je také takzvaný API Explorer, kde lze dotazy vyzkoušet.

Zdroj: [9], [10], [11], [12].

4 REALIZACE APLIKACE

4.1 Požadavky

Prvotní specifikace aplikace umožňuje stanovení požadavků na aplikaci. Požadavky lze rozdělit na dva druhy požadavků, a to na funkční a nefunkční.

4.1.1 Funkční požadavky

Funkční požadavky definují, co by výsledný systém či aplikace měla dělat. V této aplikaci jsou funkční požadavky následující:

- Aplikace bude sledovat a zobrazovat polohu uživatele.
- Aplikace bude využívat dat z geosociálních sítí.
- Aplikace bude zobrazovat události, místa a příspěvky v okolí.
- Aplikace bude schopna filtrovat jejich obsah.

4.1.2 Nefunkční požadavky

Nefunkční požadavky jsou sada podmínek omezující daný systém či aplikaci. V rámci dané aplikaci jsou definované následující:

- Aplikace bude napsána v jazyce Kotlin.
- Aplikace bude vyvíjena v prostředí Android Studio.
- Aplikace bude komunikovat se servery pomocí knihovny Retrofit.

Zdroj: [13].

4.2 Použité technologie

4.2.1 Retrofit

Retrofit je bezpečný HTTP klient pro Android a Javu, vyvinutý společností Square, která stojí také za vývojem frameworku Dagger pro dependency injection nebo také OkHttp. Díky tomuto frameworku je relativně snadné komunikovat s webovou službou založenou na RESTu. Komunikace se skládá z načítání a nahrávání strukturovaných dat, většinou je to JSON či XML. V programu je nutné definovat konvertor, který bude data serializovat a deseralizovat. Typicky pro JSON je použit Gson, je však možné definovat vlastní konvertory pro zpracování XML nebo jiných protokolů. Pro http dotazy Retrofit používá již zmíněnou knihovnu OkHttp.

4.2.1.1 Použití Retrofitu

Pro práci s Retrofitem je nutné definovat následující typy tříd:

- Modelové třídy, vzniklé překonvertovaním JSON souboru (lze využít online generátor).
- Rozhraní, které definuje jednotlivé volání.
- Třída Retrofit.Builder je instance, která využívá předešlé rozhraní a adresu API k sestavení celé adresy pro HTTP volání.

Modelové třídy

Modelové třídy jsou datové třídy zrcadlící obsah strukturovaného souboru. V tomto případě je to JSON. Tyto třídy do sebe mohou být různě vnořovány. Vstupní JSON by mohl vypadat zjednodušeně takto.

```
{
    id: 2753,
    first_name: "Milan",
    last_name: "Suchomel"
}
```

Kód 1 - Ukázka odpovědi JSON

Pro tento soubor typu JSON je nutné vytvořit třídy ze stejnými atributy. Existují konvertory, které ze vstupního JSON soboru vytvoří potřebné třídy. Výsledek vypadá následovně.

```
data class User(
    val id: Int,
    val first_name: String,
    val last_name: String
)
```

Kód 2 - Překonvertovaný JSON do datových tříd

API rozhraní

Jednotlivé metody zmíněného rozhraní představují právě jedno možné volání. Musí mít jednu z anotací HTTP pro specifikaci typu dotazu a jsou následující:

- GET – slouží k získání dat a neměl by mít žádný vliv na data,
- POST – používá se k nahrání dat,
- PUT – nahrazuje stávající data,
- DELETE – odstraní určitý zdroj dat,
- HEAD – stejné jako metoda GET pouze bez těla odpovědi.

V této anotaci je specifikovaná relativní adresa URL. Návratovou hodnotou této funkce je objekt typu *Call* s typem očekávaného objektu.

```
@GET("users")
fun getUsers(): Call<List<User>>
```

Kód 3 - Jednoduchá metoda GET Users

Tyto metody mohou obsahovat parametry s různými anotacemi. Některé z nich budou popsány dále. Jedním z takových je nahraditelný blok. V rámci relativní URL je definován blok pomocí *{attr}*, který je poté nahrazen parametrem. Tento parametr je označen anotací *@Path*.

```
@GET("users/{name}/commits")
fun getCommitsByName(@Path("name") name: String): Call<List<Commit>>
```

Kód 4 - Metoda GET s anotací Path

Další anotací je anotace *@Query*. Touto anotací jsou označovány parametry dotazu. Jsou tedy automaticky přidány na konec adresy URL. Výsledná URL adresa následujícího dotazu by mohla vypadat takto *http://www.upce.cz/api/users?id={id}*.

```
@GET("users")
fun getUserById(@Query("id") id: Int?): Call<User>
```

Kód 5 - Metoda GET s anotací Query

Pro použití kódovaných parametrů jsou potřeba dva typy anotace. Kódovaná data se odesílají, pokud je před metodou přítomna anotace *@FormUrlEncoded*. Při definici parametrů se pak používá anotace *@Field*.

```
@FormUrlEncoded
@POST("user/edit")
fun updateUser(@Field("first_name") first: String): Call<User>
```

Kód 6 - Metoda POST s kódovaným parametrem

Poslední představenou anotací je anotace *@Body*. Tímto parametrem je nahrazeno tělo dotazu.

```
@POST("users")
fun postUser(@Body user: User): Call<User>
```

Kód 7 - Metoda POST s anotací Body

Další anotace jsou použity například pro list objektů, tyto anotace vycházejí z některých předešlých (*QueryMap*, *FieldMap*). Posledními zmíněnými anotacemi jsou anotace parametrů pracující s hlavičkou dotazu (*Header*, *HeaderMap*).

Instance Retrofit

Posledním krokem je inicializace instance API, nad kterou bude možné volat jednotlivé funkce. Pro inicializaci je nutné definovat URL použité API a také konvertor, pro tento případ je využit již zmíněný Gson konvertor.

```
val retrofit = Retrofit.Builder()
    .baseUrl(https://www.ninjaarena.cz/api/)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
api = retrofit.create(UpceApi::class.java)
```

Kód 8 - Inicializace instance Retrofitu

Nyní je k dispozici objekt reprezentující API, nad kterým lze provést volání API. V ukázce níže je představeno blokující volání.

```
val request = api.getUserById(50)
val response: Response<User>
val user: User?
try {
    response = request.execute()
    user = response.body()
} catch (ex: Exception) { }
```

Kód 9 - Synchronní volání

Synchronní volání je velmi snadné, stačí vytvořit *request*, poté ho spustit zavoláním metody *execute()*. Tato metoda vrací objekt typu *Response<User>*, který obsahuje námi požadovaný objekt typu *User*. Přestože je volání velmi snadné, častěji se používá neblokující asynchronní volání.

```
val request = api.getUserById(50)
request.enqueue(object : Callback<User> {
    override fun onFailure(call: Call<User>, t: Throwable) {
        t.printStackTrace()
    }
    override fun onResponse(call: Call<User>, response: Response<User>) {
        val user = response.body()
    }
})
```

Kód 10 - Asynchronní volání

Namísto metody *execute()* je použita metoda *enqueue()*, která jako parametr očekává objekt implementující rozhraní *Callback<User>*. Toto rozhraní obsahuje dva předpisy metod, a to pro úspěšný a neúspěšný dotaz.

Zdroj: [14], [15], [16], [17].

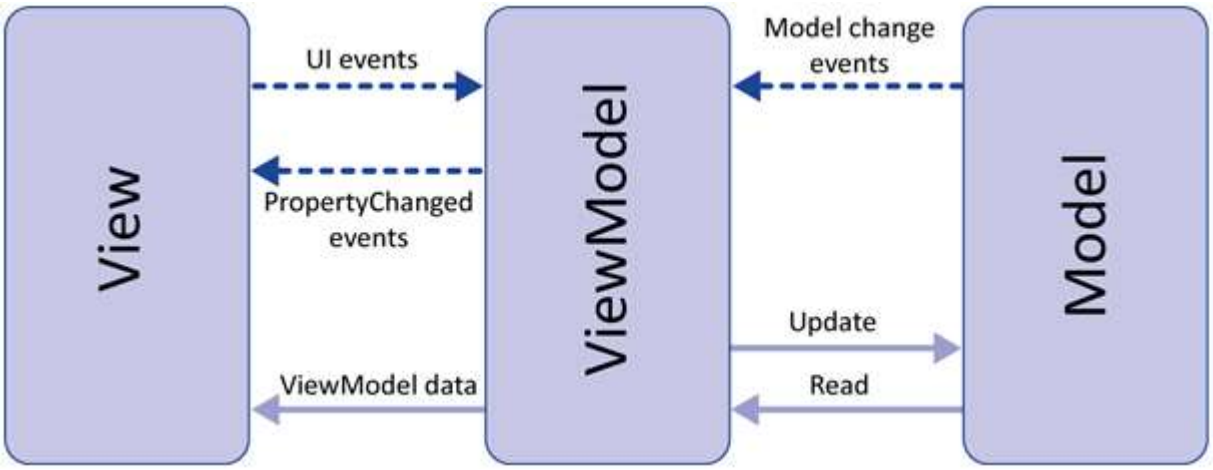
4.2.2 MVVM

Model-View-ViewModel je jednou ze softwarových architektur, která umožňuje oddělit logiku uživatelského rozhraní od té logické. Aplikace je poté mnohem přehlednější a je strukturována dle tohoto principu. Touto strukturou lze také jednoduše vyřešit problém volání metod na hlavním vlákně.

Hlavní myšlenka je tedy prostá. Vytvoření objektu, který drží stav aplikace a není závislý na jednotlivých lifecyklech aktivity.

4.2.2.1 Vrstvy MVVM

Následující obrázek vyjadřuje jednotlivé provázání níže popsaných vrstev.



Obrázek 4 - MVVM diagram

Model

Model představuje data a logiku aplikace. Může tedy obsahovat třídy pracující s lokální databází nebo například se serverovou částí aplikace. Model nezná stav ovládacích prvků. Jednou z doporučených implementací je poskytovat data prostřednictvím pozorovatelných položek, které jsou odděleny od pozorovatelů.

ViewModel

ViewModel je vrstva spolupracující přímo s vrstvou Model. Tato vrstva dále připravuje pozorovatelná data pro vrstvu View. Taktéž vrstva ViewModel zpracovává požadavky od View a provolává funkce dále do vrstvy Model. Důležitou strategií této vrstvy je odpojení od vrstvy View, tedy ViewModel si není vědom, komu předkládá výstupní data.

View

View je vrstva, která zobrazuje data a interaguje s uživatelem. Tato vrstva by neměla obsahovat žádnou logiku. Hlavní úlohou této vrstvy je reagovat na změny ve vrstvě ViewModel.

Zdroj: [18].

4.2.3 LiveData

LiveData jsou jednou z nově zavedených komponent. V rámci zmíněné architektury MVVM jsou součástí vrstvy ViewModel a vrstvy Model. LiveData schraňují data, která jsou pozorována jiným objektem. Tím je umožněno komponentám v aplikaci sledovat objekty LiveData, jejich hodnoty a změny, aniž by bylo nutné vytvoření explicitní závislosti. Tím je docíleno oddělní vlastníka objektu LiveData a spotřebitele těchto dat.

Velkou výhodou komponenty LiveData je to, že respektuje životní cykly aktivity a její stav. Tím je docíleno toho, že data jen tak nezmizí. Pokud tedy není aktivita v aktivním stavu, nebudou jí ani provolávány změny. Naopak při zničení aktivity je tato aktivita automaticky odvázána jako pozorovatel.

Vlastní vytvoření je jednoduché, potřeba je znát pouze typ dat, které budou LiveData obsahovat. V následujícím příkladu je vyobrazeno vytvoření z listu uživatelů a jejich změna.

```
val liveData = MutableLiveData<ArrayList<User>>()
liveData.value = ArrayList()
```

Kód 11 - Vytvoření objektu LiveData

Ve vrstvě View je pak potřeba na tato data vytvořit pozorování. Je nutné definovat aktivitu a metodu volající se po změně daných dat.

```
viewModel.liveData.observe(activity, Observer { list ->
    list?.let { showUsers(it) }
})
```

Kód 12 - Vytvoření spojení s LiveDaty

Zdroj: [18].

4.2.4 Google Maps SDK

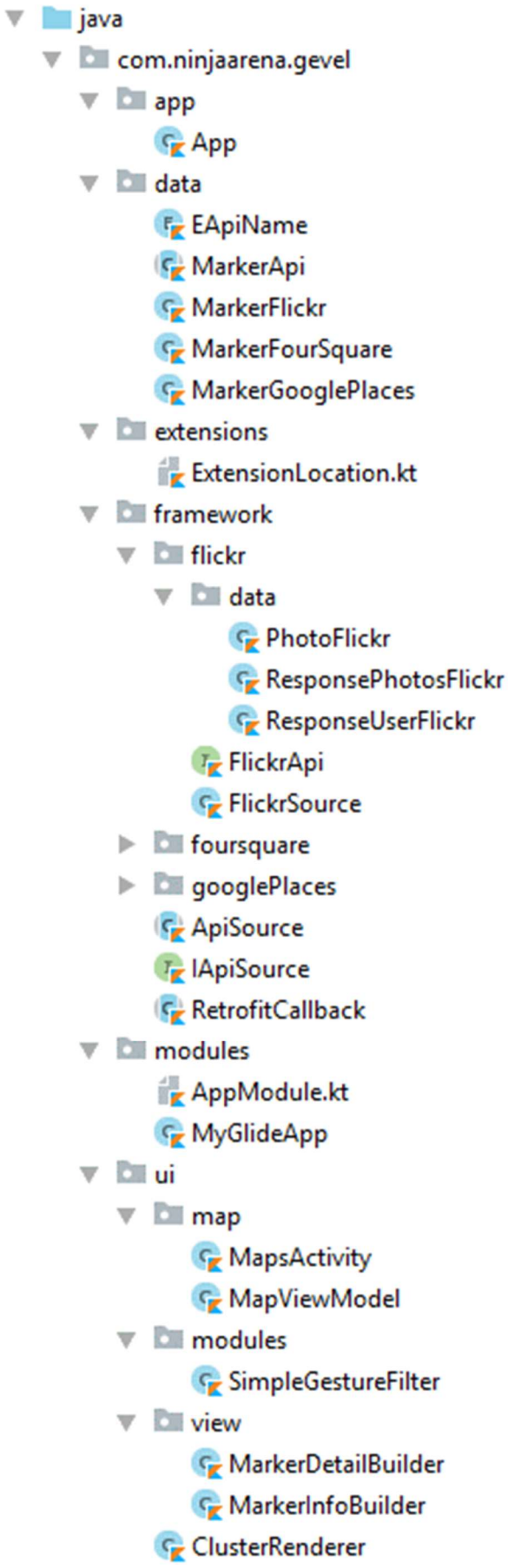
Pomocí tohoto SDK lze do aplikace s operačním systémem Android vložit mapy založené na klasickým Google Maps. Součástí je komunikace se serverem služby Google Maps, stahování dat, zobrazení mapy a reakce na gesta provedené nad mapou. Nad mapou jsou také zpřístupněny grafické operace jako například vykreslení značky, kruhu či polygonu ukotvené na určitou pozici. Na tyto objekty mohou být aplikované události, jež se provedou například po kliknutí. Pro přístup je nutný API klíč, který jednoznačně definuje aplikaci. Přístup nemusí být placený, záleží na využívání a druhu použití.

Zdroj: [19], [18].

4.3 Struktura zdrojových kódů

V této kapitole bude popsána struktura zdrojových kódů projektu vytvořeném ve vývojovém prostředí Android Studio. Budou popsány všechny důležité součásti tohoto projektu skládající se z balíčků, tříd a rozhraní. U důležitých tříd bude popsána struktura a obsah pomocí UML diagramu a ukázek kódu.

Největší důraz bude kladen na třídy nacházející se v balíčcích *framework* a *ui*, ve kterých se nachází hlavní byznys logika aplikace. Balíček *framework* obsahuje vše, co se týče serverové komunikace s různými API. Oproti tomu balíček *ui* zprostředkovává zobrazení těchto dat a interakci s uživatelem. Obrázek na následující straně vyobrazuje strukturu projektu v Android Studiu.



Obrázek 5 - Struktura zdrojových kódů

4.3.1 App

Balíček *app* obsahuje pouze jedinou třídu, obsahem malou avšak tato třída je velmi důležitá. Třída nese stejný název jako balíček tedy *App*. Tato třída je potomkem třídy *Application*, zastává celou aplikaci a při její inicializace v metodě *onCreate()* voláme inicializace důležitých modulů. V tomto případě obsahuje inicializace dvou modulů a to knihovny Koin a Timber. Timber je logovací knihovnou, Koin je určen pro vytváření a držení jednotlivých instancí objektů. Tyto instance jsou pak brány jako singletony a dle nastavení je buďto tato instance držena po celou dobu života aplikace, a nebo může být zničena a znovu vytvořena v závislosti na jejím využitím.

```
class App : Application() {

    override fun onCreate() {
        super.onCreate()
        startKoin(this, listOf(appModule))
        Timber.plant(Timber.DebugTree())
    }
}
```

Kód 13 - Třída App

Takto vytvořená třída musí být definována v souboru *AndroidManifest.xml*. Cesta k této třídě se zadá do parametru *android:name*.

4.3.2 Modules

Tento balíček obsahuje dvě krátké třídy nebo spíše soubory. Prvním je soubor *AppModule*, který obsahuje pouze jeden atribut. Druhým souborem je třída, která se nazývá *MyGlideApp*, tato třída se využívá k práci s obrázky.

AppModule

Tento soubor obsahuje pouze atribut s názvem *appModule* datového typu *Module*. Uvnitř tohoto atributu jsou definovány všechny singletony využité v aplikaci. Nejdůležitějším z nich je speciální singleton typu *ViewModel*, který je komunikátorem mezi vrstvou *view* a vrstvou

model. Při inicializaci právě tohoto objektu jsou potřeba další tři singletony starající se o komunikaci s jednotlivými API.

```
val appModule = module {
    viewModel { MapViewModel(
        flickr = get(),
        googlePlaces = get(),
        foursquare = get()
        )
    }
    single { FlickrSource() }
    single { GooglePlacesSource() }
    single { FoursquareSource() }
}
```

Kód 14 - Soubor AppModule

MyGlideApp

Tato třída je potomkem třídy AppGlideModule z knihovny Glide. Knihovna je využita k práci s obrázky. Stará se o několik důležitých funkcí jako je stahování obrázku, ukládání do cache paměti, ukládání na paměť zařízení a toto vše je zprostředkováno asynchronním procesem. Po úspěšném nahrání obrázku ho načte do příslušného *ImageView*. Nad obrázky lze provádět libovolné transformace upravující obrázek.

Následující příklad demonstruje jednoduchost použití. Nejprve je obrázek asynchronně stažen z příslušné URL adresy, a poté je oříznut do tvaru kruhu. V poslední řadě je nutné vyvolání přerušení a nahrání obrázku do *ImageView* na hlavním vlákně.

```
GlideApp.with(view.context)
    .load(marker.photo.getUserIconUrl())
    .transform(CircleCrop())
    .into(userIcon)
```

Kód 15 - Ukázka stažení obrázku

4.3.3 Data

V tomto balíčku jsou sdruženy datové třídy, kterých aplikace využívá. Obsahuje abstraktní třídu *MarkerApi*, která je použita k zobrazení značek na mapě. Tato třída dědí ze třídy *ClusterItem*, kterou je nutno použít pro funkcionalitu klastrování neboli shlukování značek. Od třídy *MarkerApi* však záměrně nejde vytvořit instance, protože neobsahuje nosné informace. Proto jsou vytvořeny třídy pro jednotlivé použité API. Tyto třídy jsou následovné:

- MarkerFlickr,
- MarkerFoursquare,
- MarkerGooglePlaces.

Názvy těchto tříd jsou vytvořeny spojením slova *Marker* a názvem určité API. Tyto třídy jsou pouze rozšířením abstraktní třídy *MarkerApi* o nosné informace. Například třída *MarkerFlickr* je rozšířena o objekt typu *PhotoFlickr* nacházející se v balíčku *framework* popsaném níže.

Další třídou v tomto balíčku je výčtový typ *EApiName*. Tato třída je použita napříč celou aplikací k jednoznačnému přiřazení jednotlivých objektů k použité API. Jedním z důležitých využití je například přiřazení tlačítek pro filtrování. Proto také obsahuje identifikační číslo tohoto tlačítka a rozšíření o *companion* funkci, která rozliší API podle tohoto identifikačního čísla.

```
enum class EApiName(val resourceId: Int) {
    FLICKR(R.id.filterApiFlickr),
    GOOGLE_PLACES(R.id.filterApiGooglePlaces),
    FOUR_SQUARE(R.id.filterApiFoursquare);

    fun getFilterResourceId(): Int {
        return resourceId
    }

    companion object {
        fun valueOfResource(resourceId: Int): EApiName {
            return when (resourceId) {
                R.id.filterApiFlickr -> FLICKR
                R.id.filterApiGooglePlaces -> GOOGLE_PLACES
                R.id.filterApiFoursquare -> FOUR_SQUARE
                else -> throw InvalidParameterException()
            }
        }
    }
}
```

Kód 16 - Třída EApiName

4.3.4 Extensions

Programovací jazyk Kotlin umožnuje vytváření takzvaných rozšiřovacích funkcí. Tyto funkce mohou rozšiřovat jakoukoli třídu. Takovýmto funkcím může být definována viditelnost jako klasickým metodám. Proto tu je pouze jedna funkce, která rozšiřuje třídu *Location*, ostatní jsou definovány jinde. V rámci některých API je práce s lokací odlišná, proto je definovaný převod na objekt typu *LatLng*. V následující ukázce je jak funkce tak ukázka volání této funkce.

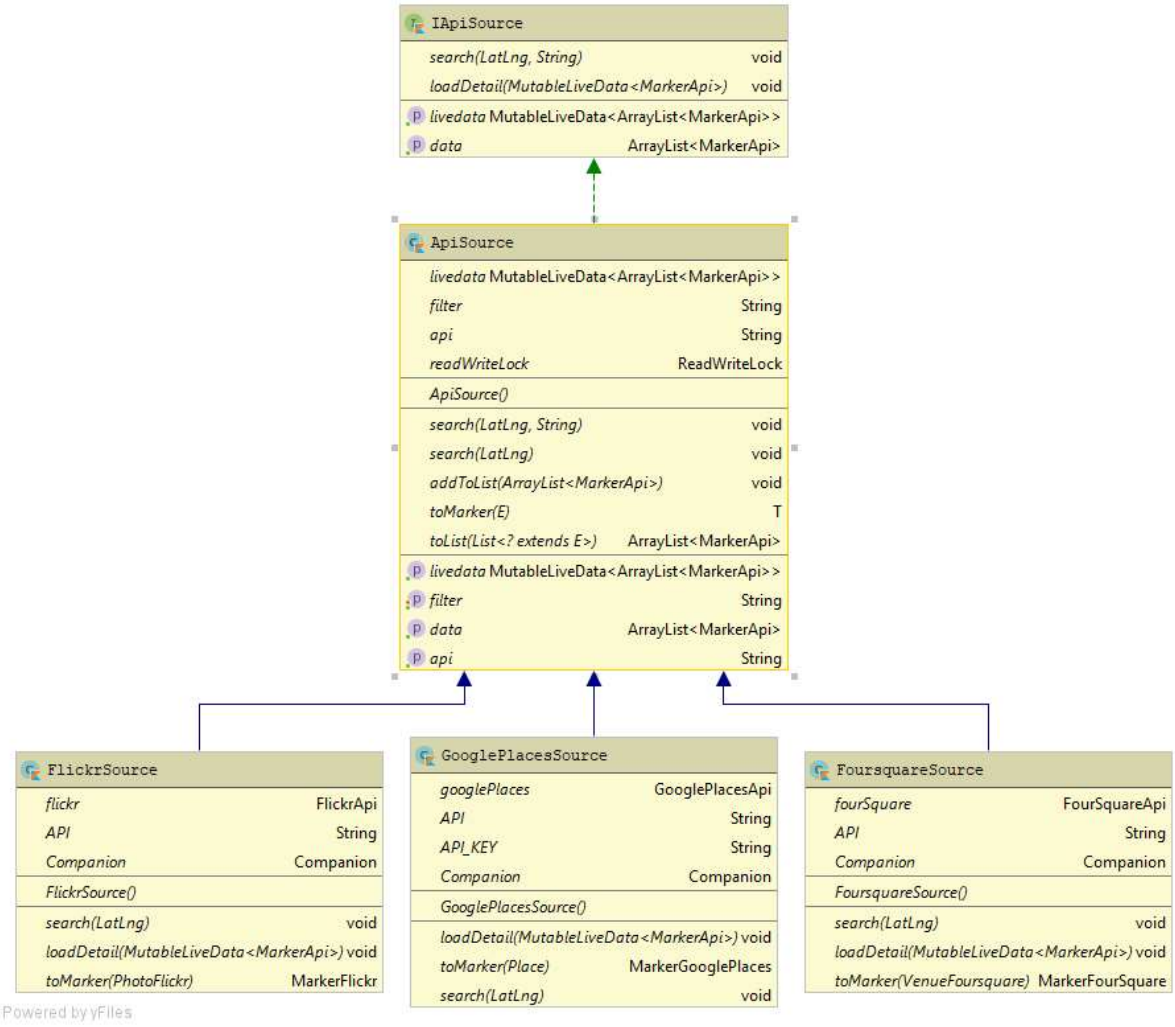
```
fun Location.toLatLng(): LatLng {
    return LatLng(this.latitude, longitude)
}

val latlng: LatLng = location.toLatLng()
```

Kód 17 - Ukázka rozšířené funkce

4.3.5 Framework

Tento balíček jak již bylo popsáno, obsahuje třídy zajišťující komunikaci s jednotlivými API. Tyto třídy jsou shlukovány podle jednotlivých API do balíčků. Přímo v tomto balíčku jsou pouze dvě třídy s jedním rozhraním, předpisy pro samotné konkrétní třídy a třída definující obecnou návratovou hodnotu z dotazu na server.

V následujícím UML diagramu jsou vyobrazeny třídy a rozhraní starající se o serverovou komunikaci s jednotlivými API.



Obrázek 6 - UML diagram tříd ApiSource

IApiSource

Nejobecnější strukturou je rozhraní definující metody pro serverový zdroj dat. Samotné rozhraní pracuje pouze s třemi metodami a jednou hodnotou. První metodou je metoda *getData()*, ta má za úkol pouze vrátit aktuální uložené hodnoty. Další je metoda pro získání dat ze serveru v závislosti na lokaci a filtru. Předávání dat zpět do uživatelského prostředí je

zprostředkováno pomocí objektu livedata. Tento objekt se stará o udržování dat a při případné změně informuje odběratele, kterým je konkrétní aktivita. Poslední metodou tohoto rozhraní je metoda zajišťující získání dodatečných dat pro detail obsahu. Opět tato metoda pracuje z livedaty, které předává parametrem.

```
interface IApiSource {

    val livedata: MutableLiveData<ArrayList<MarkerApi>>

    fun getData(): ArrayList<MarkerApi>
    fun search(location: LatLng, filter:String? = null)
    fun loadDetail(markerDetail: MutableLiveData<MarkerApi>)

}
```

Kód 18 - Rozhraní IApiSource

ApiSource

ApiSource je abstraktní třídou implementující předešlé rozhraní *IApiSource*. Tato třída se stará o obecné zpracování dat, jejich filtraci a mapování. Využívá generických datových typů pro obecné řešení. Prvním je typ definující vnitřní data označený jako *E*, může jím být například třída *PhotoFlickr*. Tím druhým je typ definující samotnou mapovou značku, který musí dědit z abstraktní třídy *MarkerApi*.

Děděním z této třídy je třída potomka oproštěna od obecné práce s kolekcí a filtrem. Pro tuto funkcionalitu obsahuje několik metod, ty nejdůležitější budou dále představeny. První důležitou metodou je metoda *search()* implementována z rozhraní *IApiSource*, zajišťující resetování cache paměti při změně filtru. Také zajišťuje následné zavolání abstraktní funkce implementované potomkem pro samotné vyhledání dat na určitém místě. Druhou metodou je metoda *addToList()*, která slouží k ukládání nových dat do listu bez duplicit.

Kompletní zdrojový kód této třídy viz. Příloha A – *Třída ApiSource*.

RetrofitCallback

Pro asynchronní komunikaci s rozhraním API přes knihovnu Retrofit je nutné využít rozhraní *Callback<T>* definující dvě metody. První metodou je metoda *onResponse()*, jež se provede při úspěšné komunikaci ze serverem, tedy při přijetí HTTP odpovědi. Druhou metodou je metoda opačná, která nastane při chybě spojení. Může nastat například, když zařízení nemá přístup k internetu. Tato metoda se nazývá *OnFailure()*. Implementováním tohoto rozhraní při každém dotazu na server vzniká mnoho duplicitního kódu, z toho důvodu pro tento projekt byla vytvořena odlehčená abstraktní třída implementující toto rozhraní.

V původní metodě *onResponse()* je několik způsobů, jak taková odpověď může vypadat. Tyto případy jsou ošetřeny a jsou z nich vyfiltrovány jen odpovědi, které jsou pro nás úspěšné. Ostatní odpovědi, tedy ty neúspěšné, jsou zaznamenány v logu. Díky tomu v konkrétní implementaci může být pouze jeden řádek kódu.

```
abstract class RetrofitCallback<E> : Callback<E> {

    override fun onFailure(call: Call<E>, t: Throwable) {
        t.printStackTrace()
    }

    override fun onResponse(call: Call<E>, response: Response<E>) {
        if (response.isSuccessful) {
            response.body()?.let {
                Timber.d("Success - %s", call.request().url())
                onSuccessfulResponse(it)
            }
            if (response.body() == null) {
                Timber.e("Response body is null")
            }
        } else {
            Timber.e(response.message())
        }
    }

    abstract fun onSuccessfulResponse(response: E)
}
```

Kód 19 - Abstraktní třída ResponseCallback

4.3.5.1 Foursquare

Pro každou využitou API je vytvořen samostatný balíček pro lepší přehlednost. Obsah takového balíčku bude popsán konkrétně na balíčku *Foursquare*. Tento balíček jak název napovídá zprostředkovává komunikaci s API Foursquare. Komunikace je zprostředkována knihovnou Retrofit, proto je nutné definovat rozhraní zastupující API. Toto rozhraní je pojmenováno *FoursquareApi*. Další nutnou součástí je třída, která navazuje spojení, pro ní je již vytvořena rodičovská abstraktní třída *ApiSource*. Vytvořená instance třídy je v aplikaci vedena jako singleton pomocí knihovny Koin. V této třídě je také nutné inicializovat objekt retrofit. Další nezbytnou součástí tohoto balíčku je několik tříd zastupující objekty v JSON odpovědi.

FoursquareApi

Jak již bylo popsáno v představení knihovny Retrofit, toto rozhraní obsahuje metody definující dotazy na server. V tomto případě jsou to dvě metody. Ukázková metoda je pro stažení dat o místech definované lokací a textovým filtrem. Jak je vidět z následující ukázky kódu, jedná se o dotaz HTTP metodou GET a očekává běžné parametry *@Query*, které jsou při dotazu součástí URL adresy. Na výstupu očekává objekt typu *ResponseVenuesFoursquare* obalený

v rozhraní *Call<T>*. Toto rozhraní umožňuje vytvořit dotaz jak v aktuálním vlákně, tak ve vlákně novém, pomocí metod *execute()* resp. *enqueue()*. Třída *ResponseVenuesFoursquare* blíže popsána v další části.

```
@GET("venues/search")
fun searchVenues(
    @Query("query") query: String?,
    @Query("ll") ll: String,
    @Query("venuePhotos") venuePhotos: Int = 1,
    @Query("v") version: String = VERSION,
    @Query("client_id") clientID: String = CLIENT_ID,
    @Query("client_secret") clientSecret: String = CLIENT_SECRET
): Call<ResponseVenuesFoursquare>
```

Kód 20 - Foursquare API searchVenues()

ResponseVenuesFoursquare

Tato třída je odrazem JSON souboru a jeho objektů při dotazu na vyhledávání míst pomocí API Foursquare. Přesněji tato konkrétní třída představuje pouze kořenový prvek takovéhoto souboru. Pro celkové naparsování celého JSON souboru do Kotlin objektů je nutná kaskáda datových tříd. Tato třída je obsahově velmi malá, jak je vidět na následující ukázce.

```
data class ResponseVenuesFoursquare(
    val response: Response
) {
    data class Response(
        val venues: ArrayList<VenueFoursquare> = ArrayList()
    )
}
```

Kód 21 - Třída ResponseVenuesFoursquare

Ekvivalentem této třídy pro dotaz detailních informací o určitém místě je třída s podobným názvem *ResponseVenueFoursquare*. Tato třída se liší pouze v tom, že atribut vnitřní třídy *Response* je typu *VenueFoursquare*, není to tedy list. Zároveň se liší ve jméně tohoto atributu, který je v jednotném čísle.

Veškeré další informace pak jsou ve třídě *VenueFoursquare* a jejích jedenácti vnitřních třídách. Následující ukázka obsahuje pouze jednu vnitřní třídu a samotnou třídu *VenueFoursquare*.

```
data class VenueFoursquare(
    val id: String,
    val name: String,
    val contact: Contact,
    val location: Location,
    val categories: ArrayList<Category>,
    val stats: Stats,
    val likes: Likes,
    val rating: Double,
    val ratingColor: String?,
    val tips: Tips,
    val hours: Hours,
    val bestPhoto: Photo?
) {
    ...
    data class Location(
        val lat: Double,
        val lng: Double,
        val formattedAddress: ArrayList<String>?
    )
    ...
}
```

Kód 22 - Třída VenueFoursquare

FoursquareSource

Tato třída dědí z již popsané třídy *ApiSource*. Proto v této třídě je řešena pouze komunikace s API a od všeho jiného je odstíněna. Prvním úkolem je vytvoření objektu implementující rozhraní FoursquareApi, k tomu je potřeba nejdříve objekt typu Retrofit. Pro takový objekt je nutné definovat URL, ke kterému API přistupuje a také konvertor, s kterým chceme pracovat pro parsování JSON souboru. V tomto případě je to Gson konvertor.

```
companion object {
    private const val API = "https://api.foursquare.com/v2/"
}

private val foursquare: FoursquareApi

init {
    val retrofit = Retrofit.Builder()
        .baseUrl(API)
        .addConverterFactory(GsonConverterFactory.create())
        .build()

    foursquare = retrofit.create(FoursquareApi::class.java)
}
```

Kód 23 - Inicializace FoursquareSource

V rámci aplikace se tato inicializace provede pouze jednou, protože si aplikace drží pouze jednu instanci této třídy a tu dále poskytuje.

Zásadní funkcí této třídy jsou dotazy, jež má vytvářet. Přesněji, jedná se pouze o dva dotazy pro vyhledání míst v okolí a pro získání detailních informací. Pro popsání byla vybrána první metoda. Nejprve je nutné lokaci transformovat do čitelné podoby pro API. Dalším krokem je vytvoření dotazu nad objektem *foursquare* a příslušnými předpřipravenými parametry.

Posledním krokem je paralelní spuštění dotazu. Tato metoda požaduje parametr implementující rozhraní *Callback<E>*, pro tuto situaci je v projektu vytvořena třída *RetrofitCallback<E>*, ze které je nutné vytvořit anonymní třídu a implementovat její metodu *onSuccessfulResponse()*.

```
override fun search(location: LatLng) {
    val ll = "${location.latitude},${location.longitude}"
    val request = foursquare.searchVenues(filter, ll)

    request.enqueue(object : RetrofitCallback<ResponseVenuesFou…>() {
        override fun onSuccessfulResponse(response: ResponseVenuesFou…) {
            addToList(response.response.venues.toList())
        }
    })
}
```

Kód 24 - FoursquareSource metoda search()

4.3.6 UI

Posledním balíčkem je, jak název napovídá, balíček starající se o uživatelské rozhraní. Součástí něho je pouze jedna aktivita obsahující Google mapu. Pod pojmem aktivita lze chápat jednu obrazovku v mobilní aplikaci. Dále obsahuje několik doprovodných tříd k vytváření UI, které budou popsány níže. Nezbytnou součástí tohoto balíčku je také třída *MapViewModel*.

MapViewModel

Třída *MapViewModel* dědí z abstraktní třídy ViewModel. Úkolem této třídy je zprostředkovat komunikaci mezi vrstvou View a modelovými třídami. Vrstva View je zastoupena aktivitou *MapsActivity*. Modelovými třídami jsou potomci třídy *ApiSource*. Jak z architektury MVVM vypovídá, ViewModel nemá přístup k aktivitě. Toto opatření je z toho důvodu, že aktivita jako samotná může kdykoliv zaniknout a následně se znovu vytvořit a ViewModel by pak měl neplatnou referenci. Naopak ViewModel má přímý přístup k modelovým třídám. Tyto třídy jsou požadované při inicializaci.

Modelové třídy jsou nejprve obaleny třídou *Api* a poté seskupeny do hashmapy, ve které jsou klíčovány podle enum třídy *EApiName*. Obalová třída *Api* je vytvořena z důvodu aplikace filtru. Filtrem je myšleno to, zda uživatel požaduje data právě z této API. Proto je součástí této třídy atribut *enabled* datového typu *MutableLiveData<Boolean>*. Tento atribut indikuje zda má být API použita pro vyhledávání, či nikoliv.

Třída jako taková obsahuje několik atributů typu *MutableLiveData*, tyto atributy udržují stav aktivity při jejím znovuvytvoření. Tím nejobsáhlejším je atribut obsahující všechny mapové značky a jejich data. Dále jsou to atributy pro udržení nastavení ke sledování polohy nebo udržující textový filtr.

ViewModel se také stará o to, aby procesy, jež nevyužívají UI, zbytečně neběželi na hlavním vlákně. Je více způsobů pro práci z vlákny, v tomto případě byly využity coroutines. Použití coroutines je znázorněno v následující ukázce na metodě *searchPlaces()*.

```
fun searchPlaces(location: LatLng) {
    GlobalScope.launch(Dispatchers.Default) {
        updatePlaces()
        sources.forEach { _, u ->
            GlobalScope.launch(Dispatchers.Default) {
                u.search(location, searchFilter.value)
            }
        }
    }
}
```

Kód 25 - Ukázka metody ve ViewModelu

MapsActivity

Hlavním prvkem této aktivity je Google mapa. Tato mapa zobrazuje zajímavá místa v okolí. Ve stávajícím stavu aplikace umožňuje stahovat data ze třech různých API a ty pak zobrazit na mapě. Na mapě jsou pak zobrazeny značky, které označují lokaci určitého místa.

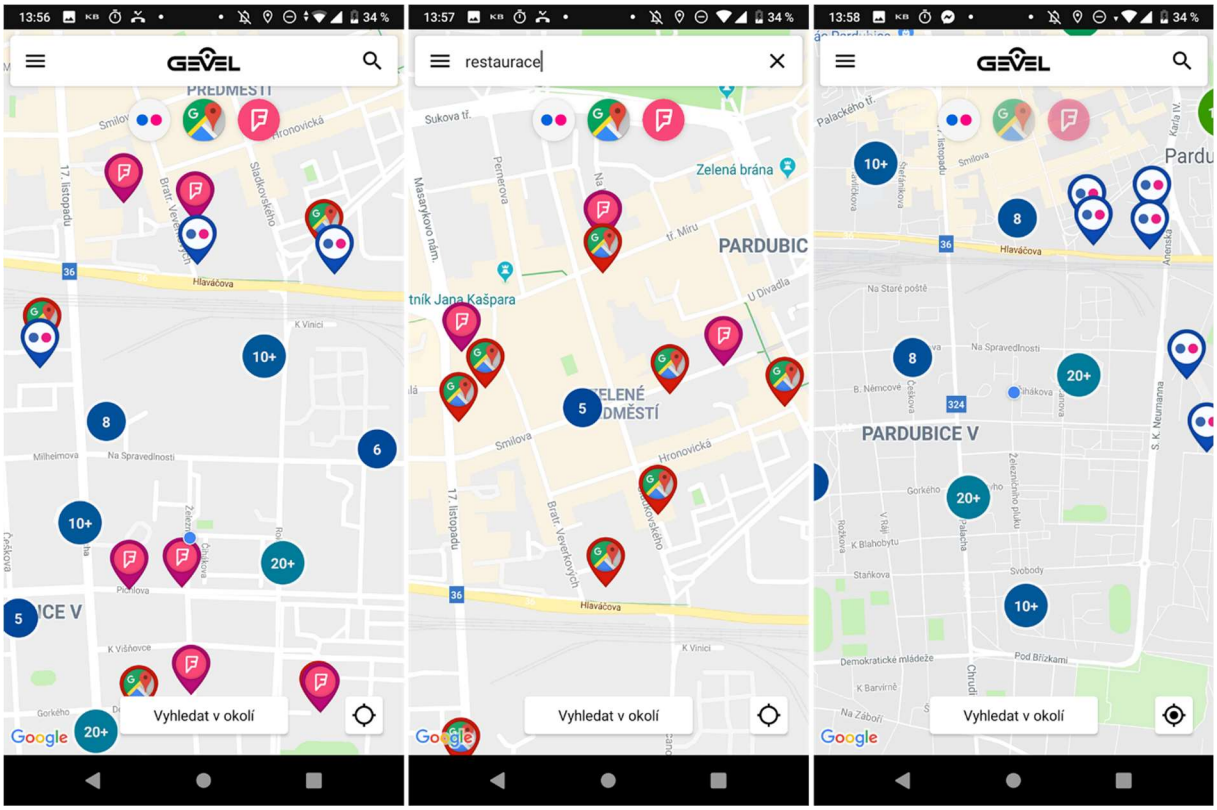
Dalším výrazným prvkem je toolbar, který ve výchozím nastavení zobrazuje pouze název aplikace. Po kliknutí na tlačítko lupy vpravo se toolbar přepne do módu pro filtrování. To uživateli zpřístupňuje možnost filtrovat například pouze restaurace.

Pod zmíněným toolbarem jsou kruhové ikony. Tyto ikony symbolizují jednotlivé API, jsou klikatelné a při jejich kliknutí lze API aktivovat nebo deaktivovat. To znamená, že aplikace pak nedotazuje určité API ke stahování dat. Funkce aplikovaná při dlouhém kliknutí na tyto ikony, aktivuje vybranou API a ostatní deaktivuje, pro snazší a rychlejší filtrování obsahu. Aktivaci či deaktivaci API lze poznat dle průhlednosti ikony. Navíc při jakékoliv změně těchto filtrů je zavoláno vyhledávání míst pro aktuální nastavení.

V dolní části obrazovky se nacházejí dvě tlačítka. Větším tlačítkem ve středu obrazovky je tlačítko pro samotné vyhledávání při změně lokace, tato cesta aktualizace byla zvolena z důvodu menšího vytížení API. V pravé dolní části je tlačítko pro zaměření polohy, při

jednoduchém kliku je zaměřena poloha zařízení pokud je dostupná. Při delším kliknutí je pak nastaveno sledování polohy, mapa se tak automaticky pohybuje s lokací zařízení.

Představené části obrazovky jsou ukázány na obrázcích níže. Obsahují jak výchozí stav toolbaru, tak i ten vyhledávací. Dále jsou různě aktivované API, a také je ukázána změna tlačítka pro sledování pozice zařízení.



Obrázek 7 - Obrazovka MapsActivity

Aktivita dále obsahuje náhledy a detaily jednotlivých míst, to však je popsáno v dalších částech této kapitoly. Přesněji v částech zaměřující se na třídu *MarkerDetailBuilder*, která zobrazuje detail místa přes celou obrazovku a třída *MarkerInfoBuilder*, která zobrazuje náhled místa v dolní části obrazovky.

Nyní bude popsána samotná třída *MapsActivity*. Tato třída má za úkol inicializace jednotlivých UI komponent a jejich správu. Události nad tlačítky předává ViewModelu jež k ní je připojen. Při inicializaci je nutná kontrola oprávnění aplikace. Toto oprávnění se týká přístupu k poloze zařízení.

Po propojení s ViewModelem je nutné vytvořit pozorovatele pro všechny atributy typu LiveData. Takovéto propojení se vytváří v *onCreate()* metodě aktivity. Pro příklad byl zvolen

pozorovatel nad všemi dostupnými značkami míst. Při změně těchto dat se překreslí značky míst na mapě.

```
viewModel.liveData.observe(this, Observer { list ->
    list?.let { showMarkers(it) }
})
```

Kód 26 - Ukázka pozorovatele nad livedaty

Pro práci s mapou bylo využito Google Maps SDK. Toto SDK zpřístupňuje fragment, v němž je obsažena mapa, nad kterou zpracovává gesta. O veškerou komunikaci ze serverem Google Maps je postaráno a vývojář se s ní pak nemusí zabývat. Tento fragment pak lze různě modifikovat. Mezi tyto modifikace patří zobrazování či schovávání výchozích tlačítek pro ovládání mapy nebo nahrazování událostí, které nastanou při interakci s mapou.

Mapa samotná umožňuje zobrazování značek na mapě. To však je v této aplikaci rozšířené o takzvané klastrování, tedy shlukování jednotlivých značek. Pokud je tedy v malém prostoru zobrazeno více značek, tak jsou automaticky sloučeny do jediné značky, označené počtem značek jež obsahuje. Metoda pro zobrazení značek na mapě je ukázána níže.

```
private fun showMarkers(list: ArrayList<MarkerApi>) {
    if (!::clusterManager.isInitialized) return

    readWriteLock.writeLock().lock()
    try {
        clusterManager.clearItems()
        clusterManager.addItems(list)
    } catch (ex: Exception) {
        ex.printStackTrace()
    } finally {
        readWriteLock.writeLock().unlock()
    }
    clusterManager.cluster()
}
```

Kód 27 - Metoda pro zobrazení značek

ClusterRenderer

ClusterRenderer je jednoduchou třídou starající se o způsob vykreslení jednotlivých značek na mapě. V tomto případě se jedná o výběr ikony na základě druhu využité API. Na obrázku níže jsou značky pro jednotlivé API, vlevo je ikona pro Flickr, pak Google Places a nakonec Foursquare.

Obrázek 8 - Značky jednotlivých API

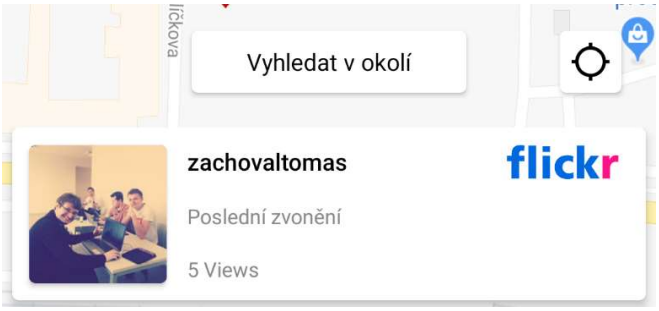
Pro takovéto rozhodování je určeno v jazyce Kotlin použití klíčového slova *when*. Je to ekvivalentem klíčového slova *switch* v jiných jazycích. Pomocí tohoto lze také přímo přiřazovat do proměnné jak je ukázáno v tomto příkladu zdrojového kódu.

```
val icon = when (item?.api) {
    EApiName.FLICKR -> flickrMarker
    EApiName.GOOGLE_PLACES -> googlePlacesMarker
    EApiName.FOUR_SQUARE -> foursquareMarker
    else -> throw InvalidParameterException()
}

markerOptions?.icon(icon)
```

Kód 28 - Přiřazení značce ikonu

MarkerInfoBuilder

Třída *MarkerInfoBuilder* je použita k vytvoření náhledu jednotlivých značek. Po kliknutí na nějakou značku je v dolní části zobrazen *FrameLayout*, jež obsahuje základní informace o daném místě. Mezi tyto informace může patřit miniatura obrázku, jméno místa nebo například kategorie do nichž spadá.



Obrázek 9 - Ukázka náhledu Flickr

Funkcí této třídy je tedy inicializace jednotlivých komponent a následné vyplnění daty. Pro každou z API je vytvořena vlastní metoda pro naplnění dat. V následujícím kódu je ukázáno jak tato metoda je implementována pro Flickr API.

```
private fun fillContent(view: View, marker: MarkerFlickr) {
    val title = view.findViewById<TextView>(R.id.title)
    val username = view.findViewById<TextView>(R.id.username)
    val stats = view.findViewById<TextView>(R.id.stats)
    val image = view.findViewById<ImageView>(R.id.image_thumbnail)
    val apiImage = view.findViewById<ImageView>(R.id.api_image)

    username.text = marker.photo.ownername
    title.text = marker.photo.title
    stats.text = "${marker.photo.views} Views"
    apiImage.setImageResource(R.drawable.flickr_logo)

    GlideApp.with(view.context)
        .load(marker.photo.getSquareLargeUrl())
        .transform(RoundedCorners(10))
        .into(image)
}
```

Kód 29 - Naplnění dat do rozvržení

Rozvržení neboli layout může mít každé API definované vlastní, v tomto případě však mají stejné. Definice rozvržení jsou psané ve značkovacím jazyce XML pomocí speciálních elementů, které jsou modifikovány atributy.

Elementem v takovém souboru může být textové pole, které může obsahovat mnoho atributů. Tyto atributy mohou určovat velikost textového pole nebo třeba barvu textu. Tento konkrétní element nemůže obsahovat hodnotu. Hodnotou je totiž v tomto souboru definována hiearchie komponent.

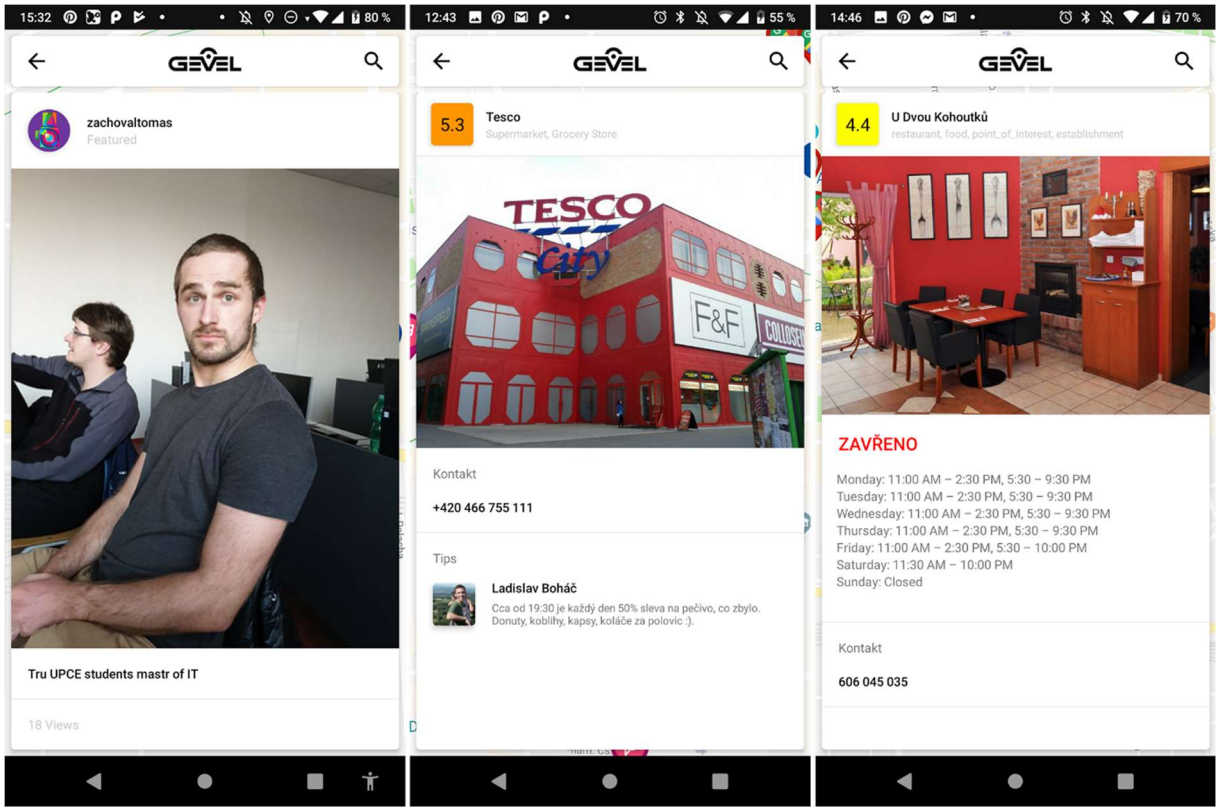
Příklad takovéhoto rozvržení viz. Příloha B – *Layout list_item_default*.

MarkerDetailBuilder

Poslední představenou třídou je třída starající se o zobrazení detailních informací o místě. Jako předešlá třída inicializuje rozvržení a plní ho daty. Pro každé API je definováno jiné rozvržení, z důvodu odlišných dat.

K místu je ve většině případů připojena fotografie, jež je hlavním prvkem tohoto rozvržení. Dostupná data jsou závislá na využité API a oblíbenosti místa. Tato data mohou obsahovat například hodnocení, kontakt, otevírací doba či tipy od uživatelů.

V následujících snímcích obrazovky jsou zobrazeny jednotlivé detaily. Pořadí uvedených snímků je následovné Flickr, Google Places, Foursquare.



Obrázek 10 - Snímky obrazovky detail

ZÁVĚR

Cílem této diplomové práce bylo vytvoření mobilní aplikace pro operační systém Android v jazyce Kotlin. Výstupem této práce se stala aplikace „*Gevel*“, která shromažďuje data o okolních místech z různých zdrojů. K těmto místům je ve většině případů připojena fotografie a další doprovodné informace podle využitého zdroje. Uživateli je také zpřístupněna možnost filtrace těchto míst.

Ve stávajícím stavu aplikace shromažďuje data pouze ze tří zdrojů a to Flickr, Google Places a Foursquare. Aplikace je však navržena pro snadné přidání dalších zdrojů používající Rest API jako přístup k datům. Aktuálně je pracováno pouze s veřejnými daty těchto zdrojů, pro lepší integraci je nutné implementovat správu uživatelských účtů pro jednotlivé API. Poté by bylo možné například vytvořit jeden příspěvek a jednoduše ho sdílet se všemi těmito sítěmi.

Nejen ve sdílení, ale také v zobrazování obsahu, autor vidí největší potencionál této aplikace. Při dalším vývoji by bylo však nutné zvážit nevýhody a úskalí této aplikace. Aplikace je přímo závislá na poskytovaných datech z externího zdroje. Poskytovateli této API by mohlo přijít určité využití nevhodné a mohl by zrušit celkový přístup aplikace k jeho datům. Další nevýhodou je to, že poskytovatel nemusí zpřístupňovat veškerá data. Posledním a pro autora nejvážnějším úskalím aplikace tohoto typu je velké množství placených služeb. Tyto služby by dle jeho názoru nebylo možné pokrýt z jakéhokoliv možného výdělku aplikace.

Tato práce autorovi přinesla další rozvoj ve vývoji pro mobilní platformu. Autor aplikoval architektonický vzor MVVM, a tím si rozšířil znalosti v jeho použití pro operační systém Android. Druh takovéto aplikace by měl využití u osob, jež často využívá více jak jednu sociální síť. Přestože potenciál z role uživatele je dle autorova názoru vysoký, z pohledu vývojáře tomu tak bohužel není.

POUŽITÁ LITERATURA

[1] **FRANKENFIELD, Jake**. Geolocation. *Investopedia* [online]. [cit. 2019-05-08]. Dostupné z: https://www.investopedia.com/terms/g/geolocation.asp

[2] **GRAVITATE ADMIN**. Geolocating Carmen Sandiego. *Gravitate* [online]. [cit. 2019-05-08]. Dostupné z: https://www.gravitatedesign.com/blog/what-is-geolocation/

[3] Geocoding API. *Google Developers* [online]. 19. 12. 2018 [cit. 2019-05-08]. Dostupné z: https://developers.google.com/maps/documentation/geocoding

[4] **ROUSE, Margaret**. Geotagging. *WhatIs* [online]. [cit. 2019-05-08]. Dostupné z: https://whatis.techtarget.com/definition/geotagging

[5] **KYSELA, Jiří**. Stručný úvod do geosociálních sítí. *Internet pro všechny* [online]. 2013 [cit. 2019-05-08]. Dostupné z: http://www.internetprovsechny.cz/strucny-uvod-do-geosocialnich-siti/

[6] Geosocial networking. *Wikipedia* [online]. 2019 [cit. 2019-05-08]. Dostupné z: https://en.wikipedia.org/wiki/Geosocial_networking

[7] **ELHADY, Hady**. *Your Guide to Cross-Platform Mobile App Development Tools* [online]. 1. 10. 2018 [cit. 2019-04-20]. Dostupné z: https://instabug.com/blog/cross-platform-development/

[8] **LACKO, Ľuboslav**. *Mistrovství - Android*. Brno: Computer Press, 2017. Mistrovství. ISBN 978-80-251-4875-4.

[9] Compare All Tiers. *Foursquare* [online]. [cit. 2019-04-27]. Dostupné z: https://foursquare.com/developers/upgrade

[10] Documentation. *Foursquare Developers* [online]. [cit. 2019-04-27]. Dostupné z: https://developer.foursquare.com/docs/api

[11] API Documentation. *Flickr* [online]. [cit. 2019-04-27]. Dostupné z: https://www.flickr.com/services/api/

[12] Places API. *Google Developers* [online]. 19. 12. 2018 [cit. 2019-04-22]. Dostupné z: https://developers.google.com/places/web-service/intro

[13] **ARLOW, Jim a Ila NEUSTADT**. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

[14] Retrofit. *Square Open Source* [online]. [cit. 2019-04-21]. Dostupné z: https://square.github.io/retrofit/

[15] **VOGEL, Lars, Simon SCHOLZ a David WEISER**. *Using Retrofit 2.x as REST client - Tutorial* [online]. 5. 6. 2018 [cit. 2019-04-21]. Dostupné z: https://www.vogella.com/tutorials/Retrofit/article.html

[16] Introduction to Retrofit. *Baeldung* [online]. 26. 10. 2018 [cit. 2019-04-21]. Dostupné z: https://www.baeldung.com/retrofit

[17] HTTP – Methods. *tutorialspoint* [online]. [cit. 2019-04-21]. Dostupné z: https://www.tutorialspoint.com/http/http_methods.htm

[18] **SALEH, Hazem**. *MVVM architecture, ViewModel and LiveData (Part 1)* [online]. 31. 5. 2017 [cit. 2019-04-21]. Dostupné z: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1

[19] Maps SDK for Android. *Google Developers* [online]. 19. 12. 2018 [cit. 2019-04-22]. Dostupné z: https://developers.google.com/maps/documentation/android-sdk

[20] **ALLEN, Grant**. *Android 4: průvodce programováním mobilních aplikací*. Brno: Computer Press, 2013. ISBN 978-80-251-3782-6.

[21] *Kotlin Programming Language* [online]. [cit. 2019-05-08]. Dostupné z: https://kotlinlang.org/

[22] **TAHIR, Nish**. *Android Development with Kotlin*. Velká Británie: Packt Publishing Limited, 2017. ISBN 9781787123687.

PŘÍLOHY

Příloha A – *Třída ApiSource*

```
abstract class ApiSource<E, T : MarkerApi> : IApiSource {

    final override val livedata = MutableLiveData<ArrayList<MarkerApi>>()
    var filter: String? = null
    val api = javaClass.simpleName
    private val readWriteLock: ReadWriteLock = ReentrantReadWriteLock()

    init {
        livedata.value = arrayListOf()
    }

    final override fun search(location: LatLng, filter: String?){
        if(this.filter != filter){
            livedata.value?.clear()
            this.filter = filter
        }
        search(location)
    }

    abstract fun search(location: LatLng)

    override fun getData(): ArrayList<MarkerApi> {
        return livedata.value ?: arrayListOf()
    }

    protected fun addToList(list: ArrayList<MarkerApi>) {
        readWriteLock.writeLock().lock()
        val source = arrayListOf<MarkerApi>()
        source.addAll(livedata.value ?: arrayListOf())

        val size = source.size
        for (it in list) {
            if (!source.contains(it)) {
                source.add(it)
            }
        }
        readWriteLock.writeLock().unlock()
    }

    abstract fun E.toMarker(): T

    protected fun List<E>.toList(): ArrayList<MarkerApi> {
        val list = arrayListOf<MarkerApi>()

        for (it in this) {
            list.add(it.toMarker())
        }

        return list
    }
}
```

Příloha B – *Layout list_item_default*

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginBottom="6dp"
        android:layout_marginStart="6dp"
        android:layout_marginEnd="6dp">

    <FrameLayout
            android:id="@+id/image_thumbnail_layout"
            android:layout_width="80dp"
            android:layout_height="80dp"
            ... />

        <ImageView
                android:id="@+id/image_thumbnail"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:contentDescription="@string/image_thumbnail"/>

    </FrameLayout>


    <TextView
            android:id="@+id/username"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:fontFamily="sans-serif-medium"
            android:textColor="@color/black"
            android:textSize="16sp"
            android:maxLines="1"
            android:ellipsize="end"
            ... />

    ...

    <ImageView
            android:id="@+id/api_image"
            android:layout_width="wrap_content"
            android:layout_height="0dp"
            android:maxWidth="100dp"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="@+id/username"
            app:layout_constraintBottom_toTopOf="@id/title"
            android:contentDescription="@string/api"
            android:layout_marginBottom="4dp"
            android:layout_marginStart="10dp"
            android:layout_marginEnd="16dp"
            android:adjustViewBounds="true"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```