Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Polohově zaměřená mobilní aplikace IoT

Bc. David Hořák

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

David Hořák

Systémové inženýrství a informatika
Informatika

Název práce

Polohově zaměřená mobilní aplikace IoT

Název anglicky

Location-oriented mobile application IoT

Cíle práce

Cílem práce je vytvořit modul pro mobilní aplikaci zaměřený na platformu Android OS. Pro vývoj bude použit programovací jazyk Kotlin. Modul může být využit jako základ pro mobilní aplikace, které slouží jako průvodce v prostoru pomocí polohového zaměření v reálném čase. Vytvořený modul bude následně implementován do navigační aplikace pro zoologickou zahradu.

Metodika

Tato diplomová práce bude zaměřena na vývoj modulu a mobilní aplikace pro platformu Android OS. V teoretické části se práce bude věnovat porovnání jazyků Kotlin a Java při vývoji pro Android, dále budou popsány další technologie, které jsou při vývoji použity. V praktické části se práce nejprve bude věnovat vytvořením modulu, který je následně implementován do Android aplikace v podobě průvodce po zoologické zahradě.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

Android; mobile development; Kotlin; Java; REST

Doporučené zdroje informací

Android Developers. Android Developers [online]. Dostupné z: https://developer.android.com/

Banerjee, Madhurima, Subham Bose, Aditi Kundu, and Madhuleena Mukherjee. ”A comparative study: Java vs kotlin programming in android application development.” International Journal of Advanced Research in Computer Science 9, no. 3 (2018): 41.

Ceri, Stefano, Piero Fraternali, and Aldo Bongio. ”Web Modeling Language (WebML): a modeling language for designing Web sites.” Computer Networks 33, no. 1-6 (2000): 137-157.

Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. ”Internet of Things (IoT): A vision, architectural elements, and future directions.” Future generation computer systems 29, no. 7 (2013): 1645-1660.

Kotlin Programming Language. Kotlin Programming Language [online]. Dostupné z: https://kotlinlang.org/

Moskala, Marcin, and Igor Wojda. Android Development with Kotlin. Packt Publishing Ltd, 2017.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 26. 02. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Polohově zaměřená mobilní aplikace IoT" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 30. března 2021 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu mé diplomové práce panu doc. Ing. Vojtěchovi Merunkovi, Ph.D. za věcné podněty a připomínky ke zpracování teoretické i praktické části práce a za čas, který mi věnoval při konzultacích.

Polohově zaměřená mobilní aplikace IoT

Abstrakt

Diplomová práce se věnuje vývoji Android knihovně pro navigaci v mapě a mobilní Android aplikaci pro zoologickou zahradu. Práce se skládá z teoretické a praktické části. V teoretické části je představen operační systém Android, jeho architektura a jak lze pro tuto platformu vyvíjet. Poté je přiblížen programovací jazyk Kotlin a na příkladech je porovnán s programovacím jazykem Java. Stručně jsou dále popsány technologie – API, architektura REST a pojem IoT. Praktická část se zaměřuje na analýzu, návrh, vývoj a testování Android knihovny i mobilní aplikace pro zoologickou zahradu. Oba softwarové programy jsou vytvořeny pomocí programovacího jazyku Kotlin a moderních přístupů v softwarovém inženýrství. Na závěr jsou uvedeny výsledky a poznatky plynoucí z této práce.

Klíčová slova: Android; mobilní vývoj; Kotlin; Java; REST

Location-oriented mobile application IoT

Abstract

This diploma thesis deals with the development of the Android library for map navigation and Android mobile application for a zoo. The work consists of theoretical and practical parts. The theoretical part introduces the Android operating system, also what are the possibilities of development for this platform and its architecture. Then the Kotlin programming language is presented and compared with the Java programming language using the examples. Furthermore, more technology is briefly described – API technology, REST architecture and the term IoT. The practical part focuses on the analysis, design, development and testing of Android library and mobile application for the zoo. Both software programs are created using the Kotlin programming language and modern approaches in software engineering. Finally, the results and findings from this work are presented.

Keywords: Android; mobile development; Kotlin; Java; REST

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

API – Application Programming Interface
ART – Android Runtime
DI – Dependency Injection
DSL – Domain specific language (doménově specifický jazyk)
ERD – Entity Realtionship Diagram
GPS – Global Positioning System (globální polohový systém)
GUI – Graphical User Interface (grafické uživatelské rozhraní)
IDE – Integrated development environment
IoT – Internet of Things
IT – Infomační technologie
JSON – JavaScript Object Notation
JRE – Java Runtime Environment
JVM – Java Virtual Machine (virtuální stroj Java)
NPE – NullPointerException (chyba kódu)
OS – Operační systém
REST – Representational State Transfer
RFID – Radio-frequency identification
SDK – Software Development Kit (sada vývojových nástrojů)
URL – Uniform Resource Locator
UX – User Experience
XML – Extensible Markup Language

1 Úvod

Zrození internetu s sebou přineslo 4. průmyslovou revoluci, která exponenciálně popohnala IT průmysl vpřed. Nastala tak éra digitalizace a společnosti všeho druhu se začaly postupně přizpůsobovat tomuto novému IT světu. Technologie se rychle měnila a IT řešení se konstantně zdokonalovaly. V nynější době je obtížné najít pracovní odvětví, které nějakým způsobem nevyužívá internet nebo jiné IT řešení.

Ve 21. století jsou mobilní zařízení neoddělitelnou součástí našeho života a konstantní vývoj smartphonů velkých i malých společností nepředpovídá v tomto ohledu změnu. Nyní téměř všechny smartphony mají přístup k internetu, určení současné polohy, technologii Bluetooth a další technologie. Ty z novějších řad jsou vybaveny například RFID čipy nebo čtečkami biometrických informací (otisk prstu, snímání obličeje). Za pomocí takto vybavených smartphonů jsme schopni téměř všeho; internetové bankovnictví, bezkontaktní platby přes platební terminály (RFID čipy), přesné určení současné pozice, propojit telefon s dalším chytrým zařízením atd.

Odvětví operačních systémů pro mobilní zařízení dominují hlavně dvě společnosti – Apple se systémem iOS a Google se systémem Android. Operační systém iOS se označuje jako kvalitnější v několika směrech, ale jeho cena je zároveň velmi vysoká, navíc jediný výrobce mobilních zařízení s iOS je samotná firma Apple. Naopak projekt Android se chlubí svým open-source systémem, což umožňuje ostatním společnostem zdokonalovat systém Android, tudíž je velmi rozšířený. Další výhody Androidu jsou nastavitelnost a obecná jednoduchost. Oba operační systémy jsou si podobné, liší se hlavně vzhledem a konkurují si v inovacích nových technologií telefonů. V konečném důsledku záleží jen na preferencích uživatele, který operační systém mu více vyhovuje.

Digitalizaci chopila do rukou společnost Google, která se proslavila svým vyhledávačem a dalšími svými produkty. Produkt Google Maps z roku 2005 revolucionizoval pohled na celosvětové mapy i navigaci v mapě. Nabídla funkce jako vyhledání v mapě, navigaci, Google Street View – nafocené ulice; a vylepšila mapu pomocí snímků ze satelitu i dalších vrstev map. Postupem času se tato technologie zlepšila a nyní nabízí funkcionalitu jako například vyhledání MHD spoje, zobrazení dopravní špičky nebo zobrazení prodejen v mapě. I přes tyto pokročilé funkce neumí Google Maps uspokojit všechny uživatelské požadavky – nastavit podklad mapy, upravit bod současné polohy nebo jiné pokročilejší požadavky. Tato technologická mezera dala podnět této práci.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této diplomové práce je navrhnout a naprogramovat modul pro mobilní aplikaci zaměřený na platformu Android OS. Pro vývoj bude použit programovací jazyk Kotlin, který bude popsán a porovnán s jazykem Java v teoretické části této práce. Modul neboli knihovna může být využita jako základ pro mobilní aplikace, které slouží jako průvodce v prostoru pomocí polohového zaměření v reálném čase. Vytvořený modul bude následně implementován do navigační aplikace pro zoologickou zahradu.

2.2 Metodika

V teoretické části bude představen operační systém Android, dále jak lze pro tuto platformu vyvíjet a jeho architektura. Poté bude popsán programovací jazyk Kotlin a bude porovnán s programovacím jazykem Java v příkladech. Následně budou zmíněny relevantní webové technologie API, REST, JSON a pojem IoT. Nakonec bude vysvětlen nástroj WebML a popsány jeho jednotlivé modely.

Praktická část se bude věnovat navigačnímu modulu a posléze aplikaci pro zoologickou zahradu. U obou aplikací budou zdokumentovány důležité části vývoje – analýza, návrh, implementace, testování; s hlavním důrazem na programatickou implementaci. Navigační modul bude probrán více do hloubky, bude hlavně popsána technologie mapy a algoritmus pro vyhledání nejkratší cesty v mapě, kdežto u aplikace pro zoologickou zahradu se budeme soustředit na návrh a zapojení modulu.

3 Teoretická východiska

3.1 Operační systém Android

Android je open source softwarový zásobník založený na Linuxu vytvořený pro širokou škálu zařízení. Spoléhá se na Linux jádro, které OS Android využívá pro základní funkce jako je správa vláken a paměti na nízké úrovni. Použití tohoto známého jádra také zjednodušuje výrobcům vývoj ovladačů hardwaru (Google 2020h).

Podle vývojového týmu Androidu (Google 2020i), je systém Android navržen tak, aby fungoval na mnoha různých typech zařízení, například telefony, tablety, chytré hodinky a televizory. Vývojáři tak mohou vytvářet aplikace pro široké spektrum zákazníků. Android poskytuje dynamický rámec aplikace, ve kterém lze aplikaci poskytnout prostředky specifické pro konfiguraci ve statických souborech (například různé rozložení XML pro různé velikosti obrazovky). Systém poté načte příslušné prostředky podle konfigurace zařízení. V samotné aplikaci lze také omezit, pro které zařízení je aplikace kompatibilní.

3.1.1 Historie

Chytré telefony mají za sebou již dlouhou historii 13 let. Od příchodu iPhone od společnosti Apple roku 2007 se chytré telefony každodenně zdokonalují a procházejí konstantní inovací. V OS pro chytré telefony se v podstatě přou dvě společnosti – Apple (OS iOS) a Google (OS Android).

Společnost Android Inc. byla založena v roce 2003 Andy Rubinem. Původně byla zaměřena na vývoj OS pro digitální fotoaparáty, ale postupem času se přeorientovala na tvorbu OS pro smartphony. V roce 2005 byla společnost Android Inc. zakoupena gigantem Google, o čemž se dosud mluví jako o nejlepší koupi společnosti v IT průmyslu (Looper a Daniel 2021).

OS Android si prozatím prošel 17 velkými verzemi a několika menšími, které jsou znázorněny níže. Každá z významných verzí inkrementovala Úroveň API tzv. API level. Dokumentace Android systému definuje API level jako celočíselnou hodnotu, která jednoznačně identifikuje revizi rozhraní API nabízenou verzí platformy Android (Google 2020f). Rozhraní v tomto smyslu je myšleno jako balíček možností aplikací pro interakci se základním systémem Android. Aktualizace rozhraní API jsou navrženy tak, aby nové rozhraní zůstalo kompatibilní s dřívějšími verzemi rozhraní API. To znamená, že většina změn v API je aditivní a zavádí nové nebo náhradní funkce. Pro vývojáře Android aplikací je tato hodnota velmi důležitá, protože použití novějších funkcí z rozhraní nemusí fungovat na starších verzí Androidu.

Tabulka 1: Přehled API úrovní a jejich podílu

Název verze	Verze	Úroveň API	Podíl verzí Androidu k 31.12.2020
(bez názvu)	1.0	1	-
(bez názvu)	1.1	2	-
Cupcake	1.5	3	-
Donut	1.6	4	-
Eclair	2.0	5	-
Eclair	2.0.1	6	
Eclair	2.1	7	
Froyo	2.2.x	8	-
Gingerbread	2.3 - 2.3.2	9	-
Gingerbread	2.3.3 - 2.3.7	10	
HoneyComb	3.0	11	-
HoneyComb	3.1	12	
HoneyComb	3.2.x	13	
Ice Cream Sandwich	4.0.1 - 4.0.2	14	-
Ice Cream Sandwich	4.0.3 - 4.0.4	15	
Jelly Bean	4.1.x	16	-
Jelly Bean	4.2.x	17	
Jelly Bean	4.3.x	18	
KitKat	4.4 - 4.4.4	19	0,99 %
Lollipop	5.0	21	3,29 %
Lollipop	5.1	22	
Marshmallow	6.0	23	5,96 %
Nougat	7.0	24	4,2 %
Nougat	7.1	25	3,04 %
Oreo	8.0	26	5,24 %
Oreo	8.1	27	9,96 %
Pie	9	28	21,8 %
Android 10	10	29	42,89 %
Android 11	11	30	nedostupné

Zdroj: (Google 2020k), (StatCounter 2021)

Historii Android verzí přehledně popisuje dvojice autorů Christian de Looper a Daniel Martin ve svém elektronickém článku na webu Digital Trends (Looper a Daniel 2021):

Android 1.0

První verze Androidu z roku 2008, se zdaleka nepodobala nynější verzi. Chlouba vývojářů byla možnost úpravy widgetů na domovské obrazovce, funkcionalitu, kterou Apple ve svém iOS zahrnul až v roce 2020. OS zahrnoval hlubokou integraci s tehdy začínající aplikací Gmail a obchodem Market, který byl později nahrazen aplikací Google Play.

Android 1.5 Cupcake

První významná aktualizace systému Android z roku 2009 získala nejen nové číslo verze, ale také jako první použila schéma pojmenování podle dezertů – Cupcake. Nejdůležitější přínos této verze byla klávesnice na obrazovce, doposud měla klávesnice jen hardwarovou podobu. Cupcake rozšířil funkcionalitu fotoaparátu o nahrávání videa. Další krok vpřed bylo umožnění vývojářům třetích stran vytvářet widgety.

Android 1.6 Donut

Android Donut poskytl uživatelům docela velkou aktualizaci. Například přinesl Android milionům lidí přidáním podpory pro sítě CDMA (Code-division multiple access), jako jsou Verizon, Sprint a mnoho velkých sítí v Asii.

Cílem Donutu bylo učinit Android uživatelsky přívětivějším. Donut byla první verzí systému Android, která podporovala různé velikosti obrazovek, což znamená, že výrobci mohli vytvářet zařízení s požadovanou velikostí displeje a stále používat Android. Dále bylo přidáno rychlé vyhledávací pole, uživatelé nyní mohli rychle prohledávat web, místní soubory, kontakty a další položky přímo z domovské obrazovky, aniž by museli otevírat dedikované aplikace.

Android 2.0 Eclair

Eclair přinesl obrovské změny do OS Android, z nichž mnohé jsou stále součástí současného Androidu. Obrovským přídavkem byla navigační aplikace Google Maps, která byla počátek úpadku GPS jednotek v autech. Ve verzi Eclair měla již Google Maps aplikace podrobnou navigaci, navádění hlasem a další základní funkcionality, a k tomu byla zdarma.

Internetový prohlížeč v systému Android Eclair byl také přepracován pro novou verzi. Google přidal do prohlížeče podporu HTML5 a schopnost přehrávat videa, čímž se Android vyrovnal iOS. V neposlední řadě byla vylepšena zamykací obrazovka, kterou šlo odemknout přejetím prstu.

Android 2.2 Froyo

Android Froyo byl poprvé vydán v roce 2010 a byl zpřístupněn přednostně pro Google Nexus zařízení, modelovou řadu smartphonů, konkrétně model Nexus One od společnosti HTC. Froyo se zaměřil spíše na zdokonalení prostředí Android, nabídl uživatelům pět panelů domovské obrazovky namísto tří a předvedl přepracovanou aplikaci Galerie. Dále přinesl podporu mobilních hotspotů a odemykání obrazovky telefonu pomocí PINu.

Android 2.3 Gingerbread

Verze Gingerbread z roku 2010 nepřinesla velké změny jen malé vylepšení systému Android, jako byl například redesign widgetů, domovské obrazovky a nastavitelnost klávesnice. Největší přídavek byla podpora pro přední kameru, která nechybí v žádném novodobém smartphonu.

Android 3.0 Honeycomb

Tato verze Androidu z roku 2011 byla výjimečná, jelikož necílila na smartphony, jako všechny verze doposud, ale na tablety. Honeycomb přišel například se změnou zeleného designu na modrou paletu. Ovšem nejvýznamnější změna byla přidání softwarové reprezentace spodního navigačního panelu, což mělo za dopad vynechání fyzických tlačítka na telefonech Android.

Android 4.0 Ice Cream Sandwich

Ice Cream Sandwich, zkráceně ICS, prosazoval hlavně změny z předchozí verze Honeycomb. Jako novou funkcionalitu představil odemknutí obrazovky obličejem nebo analýzu využití mobilních dat. Dále přinesl nové aplikace pro poštu a kalendář.

Android 4.1 Jelly Bean

I když se na první pohled Jelly Bean nezdá jako velké zlepšení OS Androidu, opak je pravdou. Nejdůležitější z těchto změn je přidání tzv. chytrých karet (Google Now), ke kterým lze přistupovat rychlým přejetím prstem z domovské obrazovky a přenést informace – tj. události kalendáře, e-maily, zprávy o počasí – na jediné zobrazení. Funkcionalita chytrých karet položila základy Google Asistenta pro budoucí verze Android OS.

Dalším vylepšením z roku 2012 byl Project Butter, který výrazně zdokonalil dotykový výkon systému Android ztrojnásobením vyrovnávací paměti. Uvedená aktualizace projektu eliminovala mnoho koktání v systému Android a celkově z něj udělala mnohem plynulejší zážitek. Obnovené písmo, rozšiřitelná oznámení, větší flexibilita widgetů a další funkce byly také přidány do Jelly Bean, což z něj činí jednu z nejvýznamnějších aktualizací Androidu.

Android 4.4 KitKat

Vydání KitKatu roku 2013 se shodovalo s premiérou zařízení Nexus 5 a přišlo s mnoho funkcemi. Například představoval jednu z nejvýznamnějších estetických změn Android OS k dnešnímu dni, modernizující vzhled Androidu. Modré akcenty nalezené v ICS a Jelly Bean se staly rafinovanějším bílým akcentem a několik přepracovaných aplikací pro Android zobrazovalo světlejší barevná schémata.

Kromě nového vzhledu přinesl KitKat také věci, jako je vyhledávací příkaz „OK, Google“, který uživateli umožňoval kdykoli přístup k chytrým kartám Google. Přinesl také nové vytáčení telefonních čísel, aplikace na celou obrazovku a novou aplikaci Hangouts, která nabídla podporu SMS spolu s podporou platformy pro zasílání zpráv Hangouts.

Android 5.0 Lollipop

Android Lollipop z roku 2014 debutoval po boku smartphonu Nexus 6. Byl první, který využíval filozofii Google „Material Design“. Aktualizace však nebyly jen estetické. Byl nahrazen stárnoucí virtuální počítač Dalvik za Android Runtime (ART), který se chlubil ahead-of-time kompilací. Což v podstatě znamenalo, že část procesního výkonu požadovaného pro

aplikace byla dodána před otevřením uvedených aplikací. Kromě toho uživatelé viděli několik zlepšení oznámení, přidání podpory obrázků RAW a řadu dalších vylepšení.

Android 5.0 také přinesl novou formu Androidu, přezdívaného Android TV, který přinesl Android na velkou obrazovku a dnes se stále používá v mnoha televizorech.

Android 6.0 Marshmallow

Android Marshmallow přinesl změny designu i změny pod kapotou. Nejvýznamnější designová změna byla použití bílé barvy namísto černé u nabídky aplikace. Nově přidaná funckionalita byla například vyhledávací lišta, která uživatelům umožnila rychle vyhledat nainstalovanou aplikaci, nebo správce paměti pro kontrolu využití paměti jakékoliv aplikace.

Další na řadě byly ovladače hlasitosti. V Marshmallow uživatelé získali přístup ke komplexnější sadě ovládacích prvků hlasitosti, což umožnilo měnit hlasitost zařízení, médií a budíků. Zabezpečení také dostalo velkou podporu v OS. Android oficiálně podporoval snímače otisků prstů počínaje verzí Marshmallow a oprávnění se významně vylepšila. Namísto aplikací, které při stahování požadovaly všechna oprávnění předem, byla oprávnění požadována podle potřeby, když byla požadována.

Android 7.0 Nougat

Největší dopad této verze byl nahrazení Google Now za Google Asistenta. Spolu s asistentem přinesl Nougat nový systém oznámení, který vylepšil jejich vzhled i funkcionalitu v operačním systému. Oznámení byla prezentována z obrazovky na obrazovku a na rozdíl od předchozích verzí systému Android je bylo možné pro snadnou správu seskupit. První verze funkcionality multitasking byl další úspěch verze Nougat, který přinesl režim rozdělené obrazovky pro více zobrazených aplikací najednou.

Android 8.0 Oreo

Android Oreo přinesl nové funkce multitaskingu po úspěchu z verze Nougat. Přinesl mód picture-in-picture a rozšířil nativně rozdělenou obrazovku. Oreo poskytlo mnohem větší kontrolu nad oznámeními. Uživatelům bylo umožněno zapnout nebo vypnout oznamovací kanály, filtrovat je podle důležitosti nebo i odložit na později.

Android 9.0 Pie

Deset let po uvedení Androidu na trh se smartphony byl vydán Pie, který s sebou přinesl několik vizuálních změn, díky nimž se stal nejvýznamnější aktualizací systému Android za několik let.

Nejdůležitější je, že se Pie zbavil spodní třítlačítkové navigace, které v Androidu existovalo roky, a nahradil jej knoflíkem ve tvaru pilulky spolu s tlačítkem zpět a gesty pro ovládání funkcí jako je multitasking. Nově byla přidána Digital Wellbeing, funkce, která uživatelovi zobrazuje využití jeho telefonu i nejpoužívanější aplikace. Tato funkce je zaměřena na pomoc uživatelům lépe spravovat jejich digitální životy, aby omezili závislost na smartphonu. Mezi další funkce Pie patří adaptivní baterie, která omezuje, kolik aplikací na pozadí baterie mohou používat.

Android 10

Od této verze Google oznámil rebranding Android OS, a to pouze číslování nových verzí bez sladkostí. I bez sladkého názvu přinesl Android 10 rozšířené funkce pro přizpůsobení celého systému, globální tmavý režim a odstranil navigační tlačítko zpět, které prošlo změny v předchozí verzi Pie.

Android 11

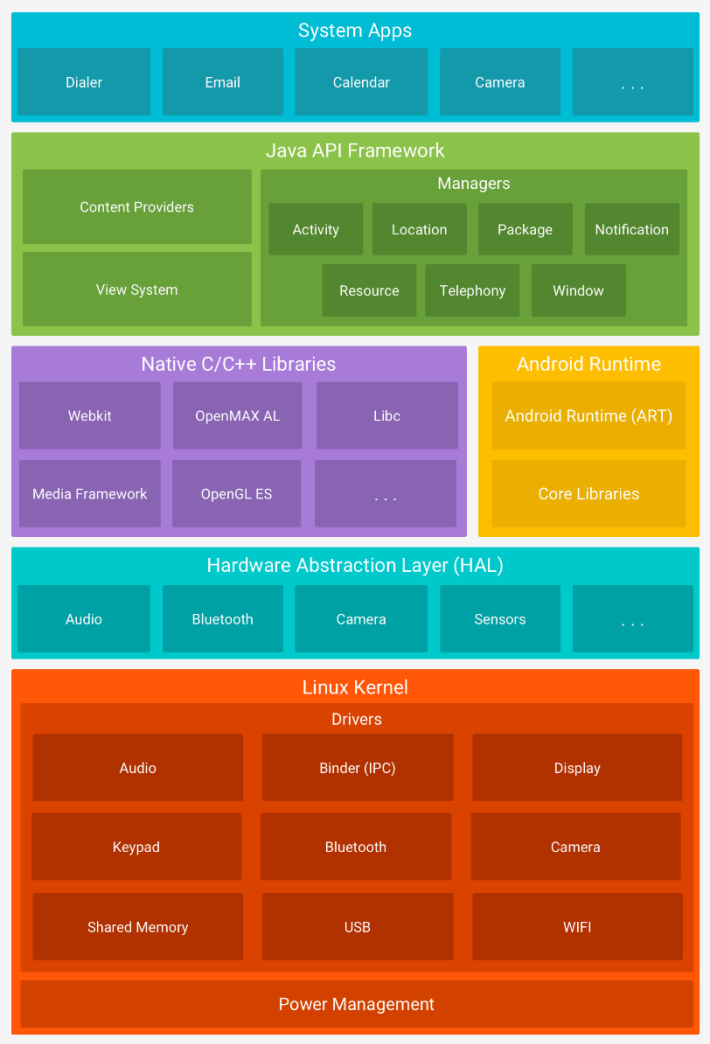
Verze Android 11 není zatím oficiálně vydána, je ale v beta verzi od června 2020 dostupná pro zařízení Google Pixel 2 a jeho modely (Looper a Daniel 2021).

Android ve své ukázce nové verze ukazuje několik nových funkcionalit (Google 2020j). Chystá například velkou změnu pro chatovací aplikace. Pro lepší konverzaci přibude možnost připnutí konverzace do menu rychlého přístupu nebo tzv. bubliny, které umožňují rychlý přístup ke konverzacím bez potřeby přerušení dosavadní činnosti na chytrém telefonu. Systém dále přichází s vestavěným záznamem obrazovky se zvukem a zobrazení dotyků na obrazovce. Velký krok vpřed v oblasti přístupnosti nové verzi je vylepšené ovládání hlasem, kterým lze ovládat systém mnohem rychleji za pomocí štítku s čísli. Menší nové funkce jsou například ovládání IoT zařízení (smart home) nebo připojení Android telefonu přímo do auta bez nutnosti použití kabelového připojení.

3.1.2 Architektura platformy Android

Android jako platforma pro zařízení lze rozčlenit na šest částí, které se starají o správný chod softwaru.

Obrázek 1: Architektura platformy Android



Zdroj: (Google 2020h)

Jak už bylo uvedeno na začátku této kapitoly, základem platformy Android je Linuxové jádro, které zajišťuje obsluhu funkcí na nejnižší úrovni. Například ART spoléhá na jádro Linuxu pro základní funkce, jako je správa vláken a paměti na nízké úrovni. Použití Linuxového jádra umožňuje systému Android využívat klíčové funkce zabezpečení a výrobcům zařízení umožňuje vyvíjet ovladače hardwaru pro známé jádro.

Vrstva hardwarové abstrakce (Hardware Abstration Layer – HAL) poskytuje standardní rozhraní, která vystavují hardwarové možnosti zařízení platformě Java API vyšší úrovně. HAL se skládá z několika knihovních modulů, z nichž každý implementuje rozhraní pro konkrétní typ hardwarové komponenty, jako je například fotoaparát nebo Bluetooth modul. Když si rozhraní API zažádá o přístup k hardwaru zařízení, systém Android načte modul knihovny pro tuto hardwarovou komponentu.

U zařízení se systémem Android verze 5.0 (Lollipop, úroveň API 21) nebo vyšší běží každá aplikace ve svém vlastním procesu a se svou vlastní instancí Android Runtime. ART je navržen pro spuštění více virtuálních strojů na zařízeních s nízkou pamětí spuštěním souborů DEX, což je formát bytecode navržený speciálně pro Android, který je optimalizován pro minimální spotřebu paměti. Uvedené informace jsme čerpali z dokumentace platformy Android (Google 2020h).

Dokumentace dále uvádí, že mnoho základních komponent a služeb systému Android, například ART a HAL, je sestaveno z nativního kódu. Tento kód vyžaduje nativní knihovny napsané v jazycích C a C++, například OpenGL ES. Platforma Android poskytuje rámec API Java, která poskytuje funkcionality některých z těchto nativních knihoven do aplikací na vyšší úrovni.

Celá sada funkcí OS Android je k dispozici prostřednictvím rozhraní API napsaných v jazyce Java. Tato rozhraní API tvoří stavební bloky, které jsou zapotřebí k vytváření aplikací pro Android. Jsou to systémové bloky:

- **View System** – rozšiřitelný systém zobrazení pro vytvoření uživatelských rozhraní, obsahuje nativní komponenty například tlačítka, texty, seznamy atd.
- **Resource Manager** – správce mimo kódových zdrojů, jako je grafika nebo překlady
- **Notification Manager** – správce oznámení pro zobrazení aplikačních upozornění
- **Activity Manager** – správce životních cyklů aktivit neboli instancí stránek (více o Aktivitách v pozdější kapitole)
- **Content Providers** – poskytovatelé obsahu, kteří umožnují přístup k datům jiné aplikace, například Kontakty

3.1.3 Architektura aplikací Android

Informace v této kapitole jsou čerpány z knihy *Professional Android* od autorů Meier Reto a Ian Lake (Meier a Lake 2018, 61-66), pokud není uvedeno jinak.

Android aplikace, dále jen aplikace, jsou softwarové programy nainstalované na Android zařízení a nativně spouštěny. Při běhu je každá aplikace spuštěna v odděleném procesu, ve své instanci ART. Pro vytvoření kvalitní aplikace je třeba porozumět komponentám, ze kterých aplikace sestává a jak s nimi nakládat.

- **Activities** – Třída *Activity*, dále aktivita, tvoří základ pro celé uživatelské rozhraní neboli obrazovku aplikace. Slouží tedy jako hlavní prezentační vrstva. UI aplikace je založeno na jedné nebo více instancí aktivit. Skládají se z tříd *Fragment* a *View*, které umožňují zobrazení informací a reagují na podněty uživatele.
- **Services** – Service komponenty jsou spuštěny bez UI, upravují zdroje dat, spouští Notifikace, a předávají Intent dále do zařízení. Používají se pro spuštění déle trvajících procesů nebo těch, které vyžadují interakci s uživatelem (procesy, které běží na pozadí).

- **Intents** – Často používané silné rozhraní, které umožňuje předávat zprávy ostatním aplikacím nainstalovaným na Android zařízení. Intent se používá hlavně při startovaní a zastavování Aktivit, Service nebo Broadcast Receiver.
- **Broadcast Receivers** – Broadcast Receivers (dále jen Receivers) slouží pro příjem Intents od ostatních aplikací, které odpovídají definovaným kritériím dané aplikace. Receivers spustí aplikaci, pro kterou je Intent určen, což z nich dělá perfektní nástroj pro aplikace řízené událostmi (event-driven applications).
- **Content Providers** – Content Providers jsou v Androidu upřednostňovaným způsobem sdílení údajů přes hranice aplikace. Aplikačním Content Providers umožňuje nastavit přístup pro ostatní aplikace do k jejím datům. Existují zabudované nativní Android Content Providers jako je například přístup ke kontaktům nebo obchod aplikací.
- **Notifications** – Notifikace umožnují aplikacím upozornit uživatele na události aplikací bez nutnosti přerušení současné aplikace. Je to doporučená technika, jak získat uživatelovu pozornost, když aplikace není aktivní. Typicky je notifikace vytvořena za pomocí Service nebo Broadcast Receiver.

Narozdíl od tradičních aplikačních platforem mají aplikace v Android platformě omezenou kontrolu nad svým vlastním životním cyklem. Aplikační komponenty musí naslouchat změnám stavu aplikace a následně reagovat. Android si agresivně střeží zdroje zařízení za cílem zaručit uživatelovi hladký chod telefonu, což znamená, že aplikace může být instantně ukončena, aby neblokovala potřebné zdroje pro aplikaci s větší prioritou. Prioritu aplikace určuje její nejdůležitější komponenta, kde jsou následující čtyři možnosti priority:

- **Aktivní proces** – *Kritická priorita*; procesy v popředí, se kterými pracuje uživatel, například otevřená aktivita aplikace. Tyto procesy se systém Android snaží upřednostňovat, tím že bere obsazené zdroje jiných méně důležitých procesů.
- **Viditelný proces** – *Vysoká priorita*; viditelné procesy, které nejsou v popředí a nereagují na vstup uživatele. Tento proces vzniká při částečném zakrytí aktivity a je zastaven jen v extrémních případech pro pokračování aktivního procesu.
- **Služba spuštěna na pozadí** – *Střední priorita*; procesy hostující služby, které nevyžadují uživatelův vstup, kvůli tomu dostávají menší prioritu.
- **Proces na pozadí** – *Nízká priorita*; procesy aktivit, které nejsou viditelné a nemají spuštěné služby. Typický proces na pozadí je při přepínání aplikací – aplikace, kterou uživatel opustí se zařadí mezi procesy na pozadí, ze kterých poté Android vybírá při uvolňování zdrojů (Meier a Lake 2018, 61).

Životní cyklus aktivity

Jak již bylo zmíněno, Android aplikace se nestará o svůj vlastní životní cyklus, ale řídí ho ART a tím taky aktivity aplikace. V případě využití aktivit v kódu aplikace se používá třída *AppCompcatActivity*, která rozšiřuje třídu *Activity* o navigační funkcionalitu. Její použití se považuje za tzv. best practice – nejlepší přístup.

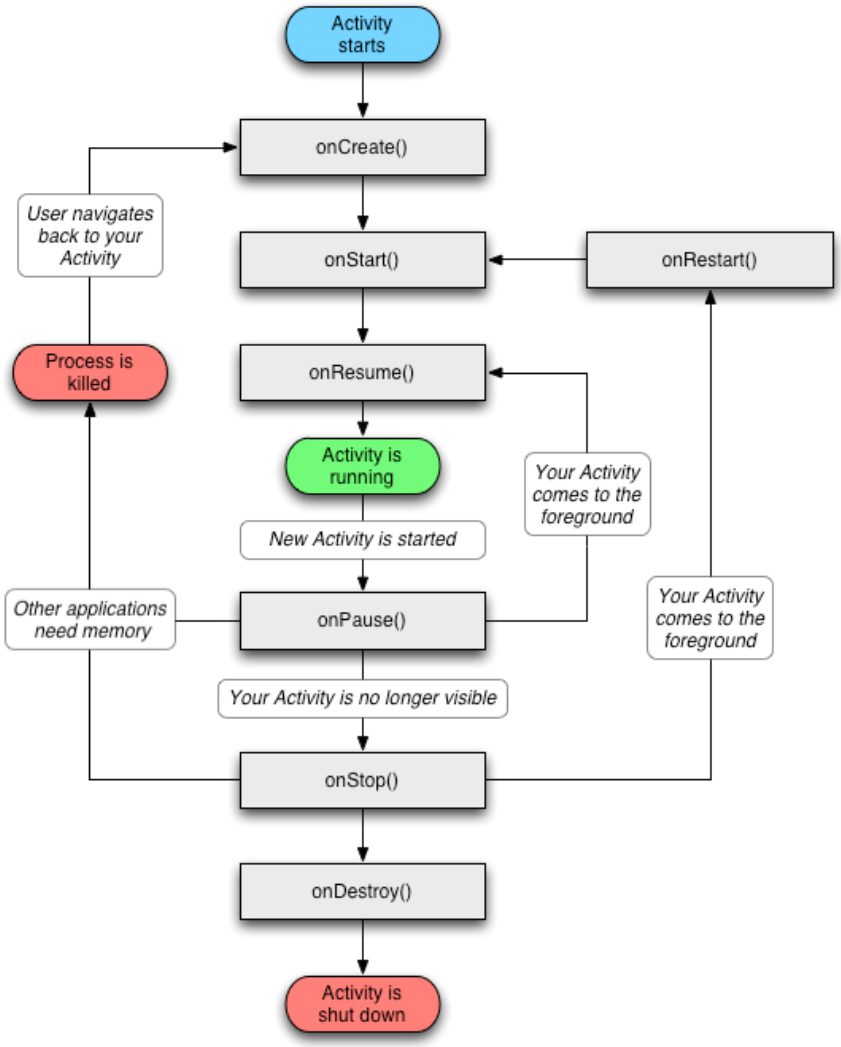
Stav aplikace se určuje podle její pozice na zásobníku aktivit, LIFO (last-in-first-out) kolekce všech běžících aktivit. Při spuštění nové aktivity je aktivita označena za zrovna aktivní a zařadí se na vrchol zásobníku. Při navigaci zpět uživatelem, nebo při uzavření aktivity v popředí, se opuštěná aktivita zruší a předchozí aktivita z vrcholu zásobníku se aktivuje (viz obrázek níže). Když aplikace nemá žádnou viditelnou aktivitu, tak se celá aplikace vloží do kolekce LRU (least-recently used), podle které Android uvolňuje zdroje zařízení.

Přechod stavů aktivit nelze přímo ovlivnit v aplikaci, záleží na uživateli a akcí systému. Aktivita má následující stavy:

- **Aktivní** – Aktivita, která je na vrcholu zásobníku aktivit, v popředí nebo reaguje na uživatelův vstup. Systém Android se snaží udržet naživu tuto aktivitu za každou cenu. V případě aktivace jiné aktivity se současná aktivita změní na pozastavenou a dále na zastavenou (viz jednotlivé stavy).
- **Pozastavena** – V některých případech je aktivita viditelná ale není zaměřena. Tohoto stavu aplikace dosáhne například při zobrazení ve více oknech, kde pozastavená aplikace je vedlejší. Android se chová k těmto aktivitám téměř totožně, jen v extrémních případech je může ukončit.
- **Zastavena** – Když není aktivita vidět, tak jí systém přiřadí stav zastavena. Aktivita sice zůstane v paměti i s potřebnými informacemi, ale je kandidát na ukončení pro uvolnění zdrojů. Při ukončení je aktivita neaktivní.
- **Neaktivní** – Po ukončení procesu aktivity je neaktivní. Neaktivní aktivita je vyřazena ze zásobníku aktivit, tudíž musí být restartována před dalším zobrazení.

Android umožňuje aplikacím naslouchat těmto přechodům mezi stavy. Při každé změně stavu aktivity předá systém aplikaci zprávu, na kterou může aplikace reagovat. Události ohledně stavů aktivit počínají vytvoření aktivity *onCreate*, start aktivity *onStart*, restart aktivity *onRestart*, pokračování v aktivitě *onResume*, pozastavení aktivity *onPause*, zastavením aktivity *onStop* a končí jejím definitivním ukončením *onDestroy*. Všechny tyto funkce lze implementovat v kódu aplikace pro speciální chování aplikace (Meier a Lake 2018, 66).

Obrázek 2: Model životních cyklů Android Aktivit



Zdroj: (Google 2020g)

Fragment

Fragment představuje opakovaně použitelnou část uživatelského rozhraní Android aplikace. Definuje a spravuje vlastní rozložení, má svůj vlastní životní cyklus a může zpracovávat své vlastní vstupní události. Fragmenty nemohou existovat samy, musí pocházet z aktivity nebo jiného fragmentu. Hodí se k definování a správě uživatelského rozhraní jedné obrazovky nebo části obrazovky. V kódu zastupuje fragment Android třída *Fragment*.

Fragmenty zavádějí do uživatelského rozhraní aktivity modularitu a opakovanou použitelnost tím, že umožňují rozdělit uživatelské rozhraní na diskrétní bloky. Lze použít více instancí stejné třídy fragmentů v rámci stejné aktivity, ve více aktivitách nebo jako podřízený prvek jiného fragmentu (Google 2020e). Fragmenty jsou často používány právě kvůli jejich modularitě.

3.1.4 Vývoj pro platformu Android

Existuje několik způsobů, jak vytvořit Android aplikaci. Přesto že v dnešní době ji lze vyvinout pomocí webové aplikace bez nutnosti znát jakýkoliv programovací jazyk, je pořád nejlepší způsob aplikaci naprogramovat. Většinou se volba technologie odvíjí od zkušenosti vývojáře, ale nejvíce rozhodnutí záleží na úloze aplikace. Článek od autora Gary Sims (Sims 2019) zmiňuje nejpopulárnější možnosti vývoje pro Android platformu:

Kotlin / Java

Programovací jazyky Kotlin i Java jsou oficiálními jazyky systému Android. Kotlin byl ovšem v roce 2019 prohlášen společností Google za oficiálně preferovaný jazyk. Aplikace napsané v těchto jazycích běží na JVM a jsou si obecně podobné. Více o jejich rozdílech bude zmíněno v kapitole programovací jazyk Kotlin v subkapitole porovnání s Javou. Vývoj pomocí těchto dvou jazyků se většinou neobejde bez podporného značkovacího jazyka XML, který se v tomto případě využívá pro definici grafického rozložení mobilní stránky tzv. layoutu.

C / C++

Kód napsaný v jazycích C/C++ se spouští na zařízení nativně, což znamená, že se program nespustí v JVM a umožňuje větší kontrolu nad zařízením například alokace paměti. S více možnostmi jako je optimalizace paměti se otevírají dveře pro lepší výkon zařízení, což má největší využití v herním průmyslu (Unreal Engine). I přes velkou kontrolu nad zařízením nepatří tato možnost k nejpopulárnějším, a to hlavně kvůli obtížnosti těchto jazyků.

C#

C# byl vyvinut společností Microsoft s cílem kombinovat sílu jazyka C++ a jednoduchosti jazyka Visual Basic. V syntaxi a optimalizaci paměti je velmi podobný jazyku Java. Síla jazyka C# vyvíjet aplikace pro Android zařízení přichází ve formě herního enginu Unity a aplikační rámec Xamarin, nebo novější alternativa .NET Multi-platform App UI (.NET MAUI).

Unity je herní engine, což je prostředek pro vykreslování 3D nebo 2D grafiky a propočet fyzikálních prvků jako je gravitace ve hrách. Umožňuje vytvářet multiplatformní hry pro Android, Apple iOS, Windows OS, herní konzole a další. Vývoj v Unity je vhodný pro úplné začátečníky i pro pokročilejší vývojáře. Jedna z výhod Unity je možnost využít Material Design od Googlu, což dává možnost vyvinout Android aplikaci v nativním vzhledu.

Microsoft představuje Xamarin jako „*projekt rozšiřující platformu .NET o nástroje a knihovny speciálně pro vytváření aplikací pro iOS, Android, macOS a další.*“ (Microsoft b.r.). Vývojářská platforma .NET je složená z nástrojů, programovacích jazyků a knihoven pro vytváření různých typů aplikací. Hlavní výhoda Xamarinu je možnost multiplatformního nativního vývoje, bez ztráty efektivnosti výkonu aplikace.

S příchodem nové verze .NET 5 v roce 2020 přichází i vylepšená verze Xamarinu – .NET MAUI. Podle Microsoftu je MAUI další krok k usnadnění multiplatformního vývoje, tím že vývojář bude pracovat na jednom projektu ale docílí funkční aplikace na všechny cílené platformy (Hunter 2020).

Flutter (Dart)

Flutter je sada SDK od společnosti Google pro vytváření pohledného a rychlého GUI pro mobilní zařízení, web a stolní počítače z jediné kódové základny. Flutter je velmi rychlý, je založen na stejné 2D grafické knihovně Skia, stejně jako webový prohlížeč Chrome a sytém Android. Použití Skia vede k nativně vypadající grafice, možnost plynulých animací a vytváření uživatelsky přívětivých widgetů. Je poháněn platformou Dart, která umožňuje kompilaci do 32bitového a 64bitového strojového kódu ARM pro Android a iOS, stejně jako JavaScript pro web a Intel x64 pro stolní zařízení. Flutter je open source a bezplatný k používání (Google 2020n).

Corona (Lua)

Dále Gary Sims (Sims 2019) zmiňuje rozhraní Corona, které nabízí další podstatně jednodušší možnost pro vývoj Adnroid aplikací, přičemž stále poskytuje slušné množství kontroly. V rozhraní Corona se programuje skriptovacím jazykem Lua. Sada Corona SDK podporuje všechny nativní knihovny a umožňuje publikovat na více platforem. Používá se z velké části k vytváření her, ale lze jej použít také mnoha jinými způsoby.

JavaScript / HTML / CSS

Největší výhoda vývoje pro Android pomocí tohoto webového tria je, že vývoj aplikace je velmi podobný webové aplikaci. Nelze však použít jen tyto technologie, ale je třeba zapojit nástroj jako je například Adobe PhoneGap nebo Ionic.

PhoneGap využívá technologii Apache Cordova a v zásadě umožňuje vytvářet aplikace pomocí stejného kódu jako pro webové aplikace. Tento kód se pak zobrazuje prostřednictvím widgetu „WebView“, který zobrazuje web v aplikaci.

Další alternativu pro vývoj mobilních aplikací nabízí technologie React Native (Facebook b.r.), která využívá sílu rozhraní React pro vývoj webových aplikací. Místo webových komponent používá jako stavební bloky nativní komponenty dle platformy. Tato technologie je populární právě kvůli podobnému vývoji pro mobilní zařízení jako pro webové aplikace.

3.2 Programovací jazyk Kotlin

Kotlin je open-source staticky typizovaný programovací jazyk, který cílí na JVM, Android, JavaScript a Native viz kapitola Kotlin/Native. Má jak objektově orientované (OO), tak funkční (FP) konstrukce. Lze jej použít jak ve stylu OO, tak ve stylu FP, nebo kombinovat prvky těchto dvou. Díky prvotřídní podpoře funkcí, jako jsou higher-order functions, function types a lambdy, je Kotlin skvělou volbou pro funkční programování. Projekt Kotlin byl zahájen v roce 2010 společností JetBrains a poprvé byl oficiálně vydán v únoru 2016 jako verze 1.0 (Ebel 2019, 24). Poslední verze jazyku Kotlin je 1.4.20 vydána 23. listopadu 2020.

Kotlin lze použít pro jakýkoliv druh vývoje, ať už na straně serveru, na straně klienta nebo v systému Android, kde byl Kotlin roku 2019 prohlášen za oficiální jazyk této platformy. S Kotlin/Native, která je v současné době ve vývoji, bude přidána plná podpora pro další platformy, jako jsou vestavěné systémy macOS a iOS. Nejvíce je Kotlin používán pro mobilní a serverové aplikace, na straně klienta s JavaScriptem nebo JavaFX. Dokumentace jazyku

Kotlin (JetBrains 2020a) dále uvádí, že je 100 % kompatibilní s JVM, což ulehčuje vývoj pro serverovou stranu, a tak lze Kotlin použít v kombinaci s existujícími rámci, jako je Spring Boot nebo Vert.x. Kromě těchto rámců existují specifické rámce napsané v jazyku Kotlin, jako je Ktor.

Kotlin/Native

Kotlin/Native je technologie pro kompilaci Kotlin kódu do nativních binárních souborů, které lze spustit bez virtuálního stroje JVM. Jedná se o backend založený na LLVM pro kompilátor Kotlin a nativní implementaci standardní knihovny Kotlin. Je primárně navržen tak, aby umožňoval kompilaci pro platformy, kde virtuální počítače nejsou žádoucí nebo možné, například vestavěná zařízení nebo iOS. Řeší situace, kdy vývojář potřebuje vytvořit samostatný program, který nevyžaduje další běhový nebo virtuální stroj.

Kotlin/Native (JetBrains 2020b) uvádí následující podporované platformy:

- iOS (arm32, arm64, simulator x86_64)
- macOS (x86_64)
- watchOS (arm32, arm64, x86)
- tvOS (arm64, x86_64)
- Android (arm32, arm64, x86, x86_64)
- Windows (mingw x86_64, x86)
- Linux (x86_64, arm32, arm64, MIPS, MIPS little endian)
- WebAssembly (wasm32)

Výhody a nevýhody

Hrubé odhady naznačují přibližně 40 % snížení počtu řádků kódu oproti ostatním jazykům. Kotlin výrazně zbezpečňuje typování objektů, např. díky podpoře typů s nulovou hodnotou je aplikace méně náchylná k chybě *NullPointerException*. Další funkce, včetně inteligentního castu, higher-order functions, extension functions a lambdy, umožňují psát expresivní kód a usnadňují vytváření DSL. Obrovská výhoda jazyku Kotlin je jeho vývojový tým, který postupně modernizuje tuto technologii a poskytuje důkladné dokumentace. Další výhoda jazyku Kotlin je, že nabízí plnou interoperabilitu s jazykem Java, tudíž zakomponovat jej do existujícího Java projektu není problém (JetBrains 2020a). Další silnou výhodu Kotlinu dává skutečnost, že Android je oficiálně tzv. Kotlin-first, což dává tomuto jazyku plnou oficiální podporu od samotných Android vývojářů (Google 2020d). Samotná společnost Google uvádí, že používá tento programovací jazyk pro vývoj více než 60 svých aplikací například Google Maps nebo Google Drive.

Tak jako každá technologie má své úskalí, tak i Kotlin má své nevýhody. Mezi nevýhody patří, že jazyk je relativně mladý, takže není tzv. battle-tested a na každou vývojářovu otázku neexistuje odpověď na internetu (ostatní jazyky zde mají výhodu). Kvůli tomu, že Kotlin není rozšířený, není tolik žádaný mezi zaměstnavateli. Vývojáři obvykle nevolí Kotlin jako svůj

první jazyk, proto musí do určité míry změnit některé ze svých programovacích zvyků (Agarwal 2019). Další problém přichází v případě data tříd, kde Kotlin automaticky generuje funkce *equals*, *hashCode*, *toString* a *copy*. Tyto funkce jsou generovány až při kompilaci kódu, což je dělá neviditelnými pro vývojáře a znepříjemňuje debugování kódu (JetBrains 2020c).

3.2.1 Asynchronní programování

Haverbeke Marijn, autor knihy *Eloquent JavaScript* (Haverbeke 2019, 182), popisuje asynchronního programování jako model prostředí, kde se věci vykonávají jedna po druhé. Když aplikace zavolá funkci, která provádí dlouhotrvající akci, může její trvání (až do jejího výsledku) pozdržet aplikaci před vykonáváním dalších akcí. V synchronním prostředí, kde se funkce požadavku vrací až poté, co dokončí svou práci, je nejjednodušší způsob provedení tohoto úkolu provést požadavky jeden po druhém. To má nevýhodu, že druhý požadavek bude spuštěn až po dokončení prvního. Celkový čas bude alespoň součet dvou dob odezvy.

Asynchronní programování umožňuje, aby se v aplikaci stalo více věcí najednou. Při zahájení akce program pokračuje ve svém běhu, po jejím dokončení je program informován a získá přístup k výsledku (například data načtená z disku). Existuje několik způsobů, jak předejít blokování aplikaci:

- **Vlákna** – Vlákna jsou zdaleka nejznámějším způsobem, jak zabránit blokováním aplikací. Dovoluje spustit část kódu v odděleném vlákně, například realizovat funkcionalitu náročnou na výkon, a přitom neblokovat uživatelské rozhraní. Ovšem mají i své nevýhody – jsou nákladné na výkon, omezené v počtu, nejsou vždy k dispozici a jsou obecně těžké pro realizaci (více vláknové programování).

- **Callback** – Cíl callbacku je předat jednu funkci (nebo část kódu) jako parametr jiné funkci a nechat ji spustit po dokončení procesu. V zásadě to vypadá jako mnohem elegantnější řešení, ale opět to má několik problémů. Obtížnost vnořených callbacků se ve složitějších případech může vymknout z ruky, což zapříčiní nečitelnost kódu a výrazně komplikuje zpracování chyb.

- **Promises, Futures** – Futures a Promises jsou jen vybrané termíny, záleží na jazyce či platformě, jak je tento princip pojmenován (dále jen Promise), dle platformy se liší i API. Obecná myšlenka je, že při zažádání o objekt se v určitém budoucím okamžiku navrátí objekt Promise, se kterým lze dále pracovat. Tento přístup vyžaduje řadu změn zejména v tom, jak vývojář programuje. Úskalí tohoto řešení je opět zpracování chyb kvůli řetězení vnořených požadavků. Komplikuje také očekávaný návratový typ v kódu, kde vrací obecný objekt Promise.

- **Reaktivní rozšíření (Rx)** – Rx byl představen Erikem Meijerem pro jazyk C#, kde byl použit pro platformu .NET. Od té doby se Rx velmi rozšířil a byl implementován pro mnoho dalších jazyků (např. RxJava, RxKotlin, RxJS atd.). Myšlenkou Rx je přejít k takzvaným pozorovatelným proudům, kde se práce s daty mění na práci s proudy dat (nekonečné množství dat) a tyto proudy lze pozorovat. Z praktického hlediska je Rx jednoduše Observer Pattern s řadou rozšíření, které nám umožňují pracovat s daty.

V přístupu je podobný Promises (viz předchozí bod), s rozdílem, že pracuje s proudy dat, umožňuje hezčí řešení chyb a má konzistentní prostředí API.

- **Coroutiny** – Přístup Kotlinu k práci s asynchronním kódem je použití coroutin, což je myšlenka pozastavitelných výpočtů, tj. že funkce může v určitém okamžiku pozastavit své provedení a pokračovat v ní později. Jednou z výhod korutin je však to, že pokud jde o vývojáře, psaní neblokujícího kódu je v podstatě stejné jako psaní blokovacího kódu. Samotný programovací model se ve skutečnosti nemění, pro použití coroutin stačí kód zabalit blokem *launch, async* nebo dalších dle cíle (JetBrains 2020d). V následující subkapitole bude tento způsob popsána více.

Coroutiny

Platforma Android (Google 2020l) představuje Kotlin coroutiny jako návrhový vzor souběžnosti, který se využívá pro splnění nároků asynchronnosti. Stejně jako vlákna mohou i coroutiny běžet paralelně, čekat na sebe a komunikovat mezi sebou. Tento vzor byl přidán do jazyku Kotlin ve verzi 1.3 a jsou založeny na zavedených konceptech z jiných jazyků. V systému Android pomáhají coroutiny spravovat dlouhotrvající úkoly, které by jinak mohly blokovat hlavní vlákno a způsobit, že aplikace přestane reagovat.

Coroutiny se vyznačují svou lehkostí, kde jich lze spustit několik na jediném vlákně díky možnosti jejich pozastavení, a tak se navzájem neblokují. Obsahují zabudovanou podporu pro zrušení své akce a vedou k méně úniků paměti. Více než 50 % profesionálních vývojářů, kteří používají coroutiny, uvedlo, že zaznamenali zvýšenou produktivitu.

Zahájit coroutinu lze velmi jednoduše blokem označeným klíčovým slovem *launch*, pozastavit jí lze metodou *delay* a to bez blokování vlákna, na kterém je coroutina spuštěna. Když chce vývojář pozastavit coroutinu hlouběji v kódu například ve vnořené funkci, tak ji musí označit slovíčkem *suspend*.

3.2.2 Porovnání s jazykem Java

Programovací jazyk Kotlin je inspirován existujícími jazyky, jako jsou Java, C#, JavaScript, Scala a Groovy. Kotlin cílí na výstižnost a přehlednost kódu. Jak už bylo zmíněno Kotlin obecně snižuje počet řádek kódu o 40 % bez ztráty výkonu nebo čitelnosti. Kotlin je hlavně porovnáván s programovacím jazyk Java, se kterou bez potíží spoluexistuje, kód Kotlinu a Javy se mohou navzájem volat bez obtíží, je tzv. interoperabilní s jazykem Java.

Jedním z hlavních návrhových pokynů, které vedly k vývoji jazyka Kotlin, je lepší zpracování nulových hodnot (null). V jazyce Java může zpracování nulových hodnot bez použití konkrétních kontrol vést k chybě *NullPointerException* (dále jen NPE).

Komparativní studie z časopisu Information and Software Technology (Ardito et al. 2020) uvádí, že až 30 % selhání Android aplikací má na svědomí právě tato chyba NPE. Mezi časté příčiny této chyby patří přístup k XML layout Android aplikace za pomocí velmi zásadních metod *setContentView* a *findViewById*. Samotná společnost Google (Google 2020d) uvádí, že při migraci svých Android aplikací z Javy na Kotlin se snížil počet řádků kódu až o 33 % a počet chyb NPE se snížil o 30 %.

Java

Společnost Oracle (Oracle 2019a) oficiálně definuje Javu jako programovací jazyk Java je univerzální, souběžný (concurrent), třídně orientovaný, objektově orientovaný jazyk. Je navržen tak, aby byl dostatečně jednoduchý na to, aby mnoho programátorů dosáhlo plynulosti jazyka. Programovací jazyk Java je podobný jazykům C a C++, ovšem opomíjí řady aspektů C a C++. Je zamýšlen jako jazyk pro produkční prostředí, nikoli jako výzkumný jazyk, a tak se vyhýbá nevyzkoušeným funkcím. Poslední verzí jazyka Javy je verze 14 vydána v březnu 2020 se dvěma subverzemi 14.0.1 a 14.0.2. Druhá ze zmíněných subverzí byl vydána 14. července 2020.

Programovací jazyk Java je silně a staticky typovaný. Tato specifikace jasně rozlišuje mezi chybami kompilace, které mohou a musí být detekovány v době kompilace, a těmi, které se vyskytnou za běhu kódu (run-time). Čas kompilace obvykle spočívá v překladu programů do strojově nezávislé reprezentace bajtového kódu. Mezi run-time aktivity patří načítání a propojování tříd potřebných ke spuštění programu, volitelné generování strojového kódu, dynamická optimalizace kódu programu a skutečné provádění programu.

Programovací jazyk Java je jazyk relativně vysoké úrovně, takže podrobnosti o strojové reprezentaci nejsou v tomto jazyce k dispozici. Zahrnuje automatickou správu úložiště / paměti, obvykle využívající garbage collector, aby se předešlo bezpečnostním problémům explicitního uvolnění, což je v C nebo C++ často obtíží. Kód napsaný v jazyku Java je obvykle kompilován do sady instrukcí bytecode a binárního formátu definovaného ve specifikaci JVM.

Java Virtual Machine (JVM)

Ke spuštění programu Java na zařízená je potřeba instalace prostředí Java Runtime Environment (JRE). Hlavní částí JRE je JVM, základní kámen platformy Java. Je to součást technologie zodpovědná za nezávislost na HW a OS, malou velikost kompilovaného kódu a schopnost chránit uživatele před škodlivými programy.

JVM je abstraktní počítač. Stejně jako skutečný výpočetní stroj má instrukční sadu a za běhu manipuluje s různými oblastmi paměti. Je relativně běžné implementovat programovací jazyk pomocí virtuálního stroje. Aktuální implementace JVM od společnosti Oracle napodobují virtuální počítač Java na mobilních, stolních a serverových zařízeních. Ovšem JVM nepředpokládá žádnou konkrétní implementační technologii, hostitelský hardware nebo operační systém. Nezná nic o programovacím jazyce Java, pouze o konkrétním binárním formátu, formátu souboru třídy (Oracle 2019b).

Porovnání

První příklad rozdílu je, že v Kotlinu je všechno objekt, dokonce i číselné hodnoty, které lze v Javě považovat i za primitivní typy (int, long, short, double, float, char, boolean, byte). Kotlin poskytuje možnost rozšířit třídu o nové funkce, aniž by daná třída musela být zděděna nebo použit jakýkoli návrhový vzor, například dekoratér, prostřednictvím speciálních deklarací nazývaných *extension* funkce a *extension* proměnné (Ebel 2019, 23). Kotlin například zpřístupňuje typ třídy v generickém zastoupení při běhu aplikace za pomocí slova *reified* (<reified T>), což Java neumožňuje. Na druhou stranu, Kotlin nemá některé vlastnosti jazyka Java, jako jsou statické členy (řeší je jinak), neveřejná proměnné a ternární operátor.

Kotlin obecně dává vývojářům více prvků moderního programování, pro vylepšení kódu a jeho čitelnosti (Ardito et al. 2020). V tomto porovnání jsou uvedeny příklady jazyka Javy verze 14 s jazykem Kotlin verze 1.4.30. Existuje mnoho příkladů, ve kterých lze tyto jazyky porovnávat, zde jsou základní z nich:

- **Deklarace proměnných**

Už v případě deklarace proměnných Kotlin ušetřuje kód na rozdíl od Javy. V případě Javy vývojář musí deklarovat datový typ proměnné, výjimka přichází v případě lokálních proměnných, kde se typ určí automaticky podle přiřazené hodnoty. V Kotlinu nezáleží na tom, kde je proměnná deklarována a automaticky zjistí datový typ podle přiřazené hodnoty. Pokud proměnná nemá mít počáteční hodnotu, je třeba datový typ deklarovat. Jazyk Java označuje finální proměnné (mutable variables; proměnné, které nemohou být přepsány) slovem *final* a jazyk Kotlin slovem *val*. Proměnné, které mohou měnit svou hodnotu po své inicializaci v Kotlinu označujeme slovem *var*.

Kód 1 – Java

```java
Boolean myBoolean;
String myString = "string";
final int myInt = 1;
var localString = "localString"; // jen lokální proměnné, Java 11+
```

Zdroj: vlastní zpracování

Kód 2 – Kotlin

```kotlin
var myBoolean: Boolean?
var myString = "string"
val myInt = 1
val localString = "localString"
```

Zdroj: vlastní zpracování

- **Deklarace metod**

Rozdíl deklarace metod je v zásadě syntaktický cukr. Kotlin nabízí mnohé rozšíření pro funkce, například *infix, inline, tailrec* funkce a vnořené funkce pro lokální použití. Možnosti funkcí v Kotlinu jsou velmi rozmanité oproti metod v jazyku Java.

V případě potřeby přidání nepovinného parametru do metody nebo konstruktoru nám oba jazyky dávají své možnosti. Java nepřidává speciální úpravu, zatímco Kotlin ulehčuje programování lehkou syntaxí přímo u parametrů v metodě. Metoda se tak může volat bez uvedení daného parametru. Kotlin také umožňuje předávat parametry do funkcí podle jmen parametrů bez ohledu na pozici v deklaraci funkce, tuto možnost jazyk Java nemá.

Kód 3 – Java

```java
int addition(int a, int b) {
  return a + b;
}

int plusOne(int a) {
  return addition(a, 1);
}
```

Zdroj: vlastní zpracování

Kód 4 – Kotlin

```kotlin
fun addition(a: Int, b: Int = 1): Int = a + b
```

Zdroj: vlastní zpracování

- **Zacházení s hodnotami null**

U objektově orientovaného programování je často problém zacházení s nulovými hodnotami tzv. null. V softwarovém inženýrství se slovem null (někdy *nil* nebo jinak) označuje absence hodnoty nebo neinicializovanou hodnotu, to záleží na programovacím jazyce. Mnoho programovacích jazyků, včetně Javy i Kotlinu, má v sobě schválně zabudovanou funkcionalitu zastavit kód při neošetřeném přístupu k null hodnotě. Je však na jazyce, jaké možnosti nabídne vývojářům těmto chybám předcházet.

Kód 5 – Java

```java
Animal dog = null;
...
if (dog != null && dog.getKingdom() != null) {
  return dog.getKingdom().getName();
}
return null;
```

Zdroj: vlastní zpracování

Kód 6 – Kotlin

```kotlin
var dog: Animal? = null
...
return dog?.kingdom?.name
```

Zdroj: vlastní zpracování

Java nenabízí žádnou speciální funkcionalitu pro kontrolu null hodnot a vývojář musí kontrolovat hodnoty manuálně. Java kód je tak náchylný na NPE. Zatímco Kotlin poskytuje mnohem lepší ochranu před NPE, a to například již při deklaraci proměnné musí vývojář určit, jestli proměnná může nabývat null hodnoty. Dále při přistupování k atributům null proměnné bez kontroly její hodnoty je nutné přidat symbol *‚?‘* nebo *‚!!‘*. Symbol otazníku říká překladači, aby automaticky zkontroloval hodnotu a řetězec symbolů *‚!!‘* naopak říká překladači, aby vyhodil chybu NPE, jestli je hodnota null (Bose et al. 2018).

Velmi užitečná syntaxe je tzv. Elvis operátor, značen *‚?:‘* (podle účesu slavného zpěváka). Tento operátor zkracuje Java ternární operátor o jednu větev podmínky. Funguje jako *else* větev, když levá strana (vlevo od Elvis operátoru) nabývá hodnoty null. Naopak Kotlin nenabízí ternární operátor (Ardito et al. 2020).

Kód 7 – Kotlin

```kotlin
val name = null
var nameLength: Int = name?.length ?: -1
```

Zdroj: vlastní zpracování

- **Statické členy (proměnné)**

Statické členy ze syntaktického hlediska lépe zpracovává jazyk Java. Kotlin tuto funkcionalitu řeší pomocí *companion* objektu. *Companion* objekt umožňuje statickou tvorbu funkcí i proměnných pro třídu, přístup mimo tuto třídu je poté stejný jako pro Java kód například *MyClass.staticMethod*. Použití *companion* object je vhodný spíše pro použití mimo tuto třídu.

Kód 8 – Java

```java
static final int ONE = 1;

static String staticMethod() {
  return "static";
}
```

Zdroj: vlastní zpracování

Kód 9 – Kotlin

```kotlin
companion object {
  @JvmStatic
  fun staticMethod() = "static"

  val ONE = 1
}
```

Zdroj: vlastní zpracování

- **Kolekce**

Kotlin striktně rozeznává měnitelné (mutable) a neměnitelné (immutable) kolekce, omezením datového typu proměnné například *MutableList* vs *List*. Výhoda je opět větší kontrola nad kódem, v tomto případě nad proměnnými. Java tyto kolekce umí rozlišit, ale dostatečně neapeluje na jejich použití.

Kotlin dále poskytuje pokročilejší operace nad kolekcemi bez nafukování kódu. Tyto operace jsou dostupné bez nutnosti používání dalších tříd. V Javě se tyto operace nejefektivněji řeší pomocí proudu dat (stream), které ale vyžadují externí třídy a několik řetězených metod.

Kód 10 – Java

```java
final List<Integer> numbers = List.of(1, 2, 3);
final List<Integer> names = List.of("Jane", "Alex", "Bob");

final List<Integer> filtered = numbers
  .stream()
  .filter(n -> n > 3)
  .collect(Collectors.toList());
final Map<String, Integer> groups = numbers
  .stream()
  .collect(Collectors.groupingBy(n -> (n % 2 == 0 ? "even" : "odd")));
final List<String> pairs = IntStream
  .range(0, Math.min(names.size(), numbers.size()))
  .mapToObj(i -> names.get(i) + ":" + numbers.get(i))
  .collect(Collectors.toList());
```

Zdroj: vlastní zpracování

Kód 11 – Kotlin

```kotlin
val numbers = listOf(1, 2, 3)
val names = listOf("Jane", "Alex", "Bob")

val filtered = numbers.filter { it > 2 }
val groups = numbers.groupBy { if (it % 2 == 0) "even" else "odd" }
val pairs = numbers.zip(names)
```

Zdroj: vlastní zpracování

- **Datová třída**

Datová třída pomáhá Kotlinu drasticky redukovat množství kódu při zachování stejné funkcionality. Jazyk Java doporučuje přidání metod tzv. getterů a setterů, pomocí kterých lze operovat nad privátní proměnnou, což kód třídy nafukuje (JetBrains 2020c). Tento tzv. boilerplate kód ovšem lze omezit použitím anotačních procesorů, například knihovna Lombok.

Kód 12 – Java

```java
public class Animal {
  private String name;
  private Kingdom kingdom;
  private int limbCount;

  public Animal(String name, Kingdom kingdom, int limbCount) {
    this.name = name;
    this.kingdom = kingdom;
    this.limbCount = limbCount;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
```

```
  }

  public Kingdom getKingdom() {
    return kingdom;
  }

  public void setKingdom(Kingdom kingdom) {
    this.kingdom = kingdom;
  }

  public int getLimbCount() {
    return limbCount;
  }

  public void setLimbCount(int limbCount) {
    this.limbCount = limbCount;
  }
}
```

Zdroj: vlastní zpracování

Kód 13 – Kotlin

```
data class Animal(var name: String,
                  var kingdom: Kingdom,
                  var limbCount: Int)
```

Zdroj: vlastní zpracování

3.3 API

API (Application Programming Interface) je sada funkcí a pravidel, která existují uvnitř softwarového programu (aplikace) umožňující interakci s ním prostřednictvím softwaru – na rozdíl od lidského uživatelského rozhraní. Na API lze pohlížet jako na jednoduchou smlouvu (rozhraní) mezi aplikací, která jej nabízí, a dalšími položkami, jako je software nebo hardware třetích stran (Mozilla Contributors 2020). API lze klasifikovat podle systémů, pro které jsou určeny – databázové API, API operačních systémů, remote API a webová API.

Databázové API umožňují komunikaci mezi aplikací a databázovým systémem. Příklad takového API je software, který sjednocuje syntaxe různých databází do obecné syntaxe pro jednodušší použití.

API operačních systémů definují, jak aplikace využívá prostředky a služby daného systému. Každý operační systém má svoji sadu rozhraní API, například Android API nebo Linux API.

Remote API definují standardy interakce pro aplikace běžící na různých zařízeních. Jinými slovy, jeden software přistupuje ke zdrojům umístěným mimo zařízení, které je požaduje. Vzhledem k tomu, že dvě vzdáleně umístěné aplikace jsou připojeny přes komunikační síť, zejména internet, většina vzdálených API je napsána na základě webových standardů. Do této třídy patří například Java Database Connectivity (JDBC) API.

Webová API, jsou nejběžnějším výskytem API. Poskytují strojově čitelná data a přenos funkcí mezi webovými systémy, které představují architekturu klient-server. Tato rozhraní API

doručují hlavně požadavky z webových aplikací a odpovědi ze serverů pomocí protokolu Hypertext Transfer Protocol (HTTP). Vývojáři mohou pomocí webových rozhraní API rozšířit funkčnost svých aplikací nebo webů. Příklad webových API je Google Maps API nebo Twitter API (AltexSoft 2019).

3.4 REST

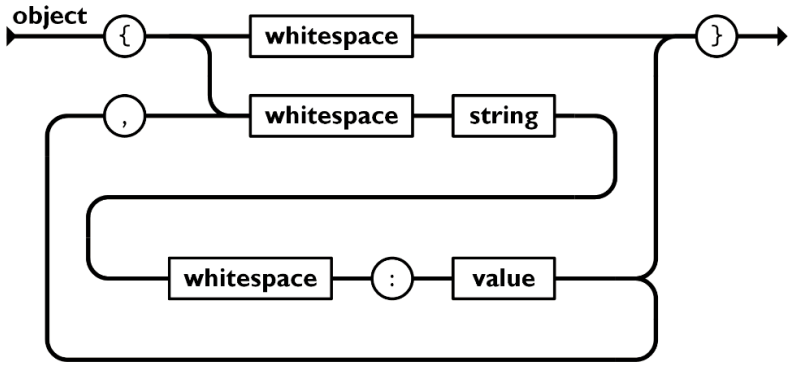
Pojem REST (Representational State Transfer) byl poprvé zmíněn ve disertační práci *Architectural Styles and the Design of Network-based Software Architectures* od Roy T. Fielding (Fielding 2000). Fielding ve své práci definoval REST jako skupinu softwarových architektur navržených pro distribuované systémy komunikujících pomocí protokolu HTTP. Tento design se soustředí hlavně na použití v client-server architekuře. Myšlenkou REST designu je, že zdroj, např. dokument nebo obrázek, je přenášen se svým stavem a vztahy (hypertext) prostřednictvím dobře definovaných, standardizovaných operací a formátů. Pro dodržení RESTful architektury se v HTTP protokolu používají hlavně metody GET/PUT/POST/DELETE, které zastávají CRUD operace (create, remove, update, delete). Nejznámější alternativou REST je protokol SOAP (Simple Object Access Protocol), jehož zprávy jsou založené na XML, další možnost je GraphQL.

3.5 JSON

Termín JSON je zkratka pro JavaScript Object Notation. Je to odlehčený textový formát pro výměnu dat používaný hlavně v aplikacích komunikující přes internet. JSON je jednoduše čitelný i zapsatelný jak strojem, tak i člověkem. Pochází z programovacího jazyka JavaScript, pro použití na jazyce ale nezáleží. Je dobře známý programátorům z rodiny jazyků C (C, C++, C#, Java, JavaScript, Perl, Python a dalších).

JSON může obsahovat dohromady šest datových typů: objekt (object), pole (array), textový řetězec hodnota, číslo, boolean (true / false) a nulová hodnota neboli null. Prázdné znaky (whitespace), mimo textový řetězec nenesou žádnou informaci, a tudíž zastávají jen visuální funkci. Prázdné znaky jsou netisknutelné znaky například mezera, tabulátor nebo nový řádek („JSON" b.r.). Na obrázku 3 je znázorněn postup rozboru JSON objektu.

Obrázek 3: Postup rozboru JSON objektu



Zdroj: („JSON" b.r.)

3.6 Internet of Things

Internet věcí (IoT) popisuje síť fyzických objektů tzv. věcí – které jsou zabudovány do senzorů, softwaru a dalších technologií za účelem připojení a výměny dat s jinými zařízeními a systémy přes internet.

IoT se za posledních několik let stala jednou z nejdůležitějších technologií 21. století. Díky nízkonákladovým výpočtům, cloudu, big data, analytice a mobilním technologiím mohou fyzické věci sdílet a sbírat data s minimálním lidským zásahem. V tomto propojeném světě mohou digitální systémy zaznamenávat, sledovat a upravovat každou interakci mezi propojenými zařízeními (Gubbi et al. 2013).

3.7 WebML

Web Modeling Language (WebML) je notace pro specifikaci webů nebo mobilních aplikací na koncepční úrovni. WebML pochází z italské univerzity Politecnico di Milano (Ceri, Fraternali, a Bongio 2000), kde tuto notaci vytvořila skupina akademiků. WebML umožňuje popis webové stránky na vysoké úrovni v různých dimenzích: její datový obsah (strukturální model), stránky, které jej tvoří (model kompozice), topologie odkazů mezi stránkami (navigační model), rozložení a grafické požadavky na vykreslování stránky (prezentační model) a funkce přizpůsobení pro doručování obsahu one-to-one (model přizpůsobení).

WebML umožňuje návrhářům vyjádřit základní vlastnosti webu či aplikace na vysoké úrovni, aniž by se zavázali k podrobným architektonickým detailům. Koncepty WebML jsou spojeny s intuitivním grafickým znázorněním, které lze snadno podporovat nástroji CASE a usnadnění komunikace s netechnickými členy týmu (grafický designer, content writer) při vývoji aplikace. WebML také podporuje syntaxi XML, které lze místo toho přenést do softwarových generátorů pro automatickou produkci implementace webu. Specifikace webu ve WebML podle italské studie sestává ze čtyř perspektiv.

1) **Strukturální model** vyjadřuje datový obsah webu z hlediska příslušných entit a vztahů. WebML nenavrhuje žádný jiný způsob pro modelování dat, ale je kompatibilní s klasickými notacemi, jako je preferovaný Entity Relationship Diagram (dále jen ERD), objektově orientovaný model ODMG nebo diagram tříd UML.
2) **Hypertextový model** popisuje navigaci uživatele ve webové aplikaci a její složení. Popisy zobrazení stránek sestávají ze dvou dílčích části, které se spojuje do Hypertextového modelu:
    - **Model kompozice** zachycuje, z jakých logických elementů se webová stránka skládá. K vytváření obrazovek lze dle WebML několik předdefinovaných jednotek obsahu tzv. unit například data, index, filter nebo scroller. Jednotka data se používá k publikování informací o jediném objektu, zatímco zbývající typy jednotek představují alternativní způsoby procházení sady objektů (například seznam alb – jednotka multi-data).

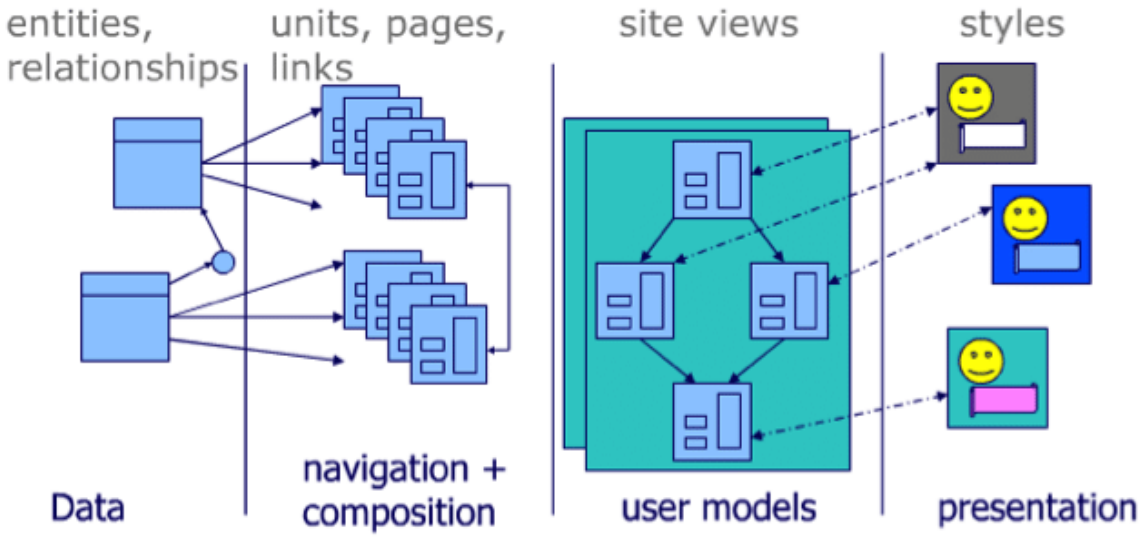
- **Navigační model** vyjadřuje, jak jsou stránky a jednotky obsahu propojeny za účelem vytvoření hypertextu (stránky). Odkazy jsou buď nekontextové, když spojují sémanticky nezávislé stránky (např. stránka umělce na domovskou stránku), nebo kontextové, když obsah cílové jednotky odkazu závisí na obsahu zdrojové jednotky (např. stránka umělce obsahuje jeho alba).

3) **Prezentační model** vyjadřuje rozvržení a grafický vzhled stránek, nezávisle na výstupním zařízení a na jazyce interpretace. Prezentaci specifikuje jak pro obecné vzhledy, tak i pro konkrétní stránky. V prvním případě jsou to vzhledy, které nejsou závislé na konkrétním obsahu stránky a zahrnují odkazy na obecné prvky obsahu (například rozložení atributů obecného objektu). V druhém případě diktují prezentaci konkrétní stránky a zahrnují explicitní odkazy na obsah stránky.

4) **Uživatelský model** explicitně modeluje uživatele a skupiny uživatelů ve schématu struktury ve formě předdefinovaných entit zvaných uživatel a skupina. Funkce těchto entit lze použít k ukládání skupinového nebo individuálního obsahu, jako jsou nákupní návrhy, seznam oblíbených položek a zdroje pro grafické přizpůsobení.

Vývoj webových aplikací je mnohostranná aktivita zahrnující různé hráče s různými dovednostmi a cíli. Proto je oddělení obav klíčovým požadavkem pro jakýkoli jazyk pro modelování webu. WebML řeší tento problém a předpokládá vývojový proces, kde různé druhy specialistů hrají odlišné role: datový expert navrhuje strukturální model, aplikační architekt navrhuje stránky a navigaci mezi nimi, grafik navrhuje styly prezentace stránek, správce webu navrhuje uživatele a možnosti personalizace. Propojení WebML modelů znázorňuje následující obrázek:

Obrázek 4: WebML spojení modelů



Zdroj: (Abrahão et al. 2018)

4 Vlastní práce

Cíl práce je rozdělen na dvě části podle technologické implementace. V první části je popsán znovupoužitelný navigační modul. V druhé části je navigační modul použit pro implementaci Android aplikace, která slouží jako průvodce v zoologické zahradě.

Veškerý kód byl napsán za pomocí IDE Android Studio. Android Studio je založeno na pokročilém IDE Intellij IDEA od společnosti JetBrains (autoři programovacího jazyka Kotlin). Tento samotný základ zkombinovaný se zaměřením na Android funkce dělá z Android Studia bezkonkurenční IDE. K velmi populárním nástrojům patří například memory profiler, náhled XML layoutů, emulace zařízení nebo přímá instalace do fyzického zařízení s debuggerem (Google 2020b). Tyto vlastnosti z něj dělají ideálního pomocníka pro vývoj Kotlin Android aplikací.

4.1 Navigační modul

V této kapitole je popsána analýza, návrh, implementace a testování navigačního modulu neboli knihovny. Cílem tohoto modulu je usnadnit vytváření Android aplikací s charakterem průvodce po vytyčené mapě. Tato knihovna je vyvíjena pod jménem Mapigate.

4.1.1 Specifikace požadavků

Pro správné naplnění cíle této části jsme nejprve nadefinovali požadavky pro navigační modul. Ze zadání diplomové práce plynou čtyři hlavní požadavky – orientace v mapě, zobrazení současné pozice, navigace v mapě a jednoduchá reakce na události vyvolané modulem.

Orientace v mapě je klíčovým požadavkem. Musí poskytovat možnost mapu vykreslit a zaznamenat do ní pevné body i zaznačit cestu. Mapa by měla být maximálně nastavitelná jako například podklad mapy, její rozměry a další elementy vyskytující se v mapě. Implementace mapy musí dovolovat její přiblížení a rotaci uživatelem pomocí intuitivních UX gest. Mapa by neměla být nekonečná, využívat se bude jen pro omezené místo, kde záleží na úrovních přiblížení.

Pro určení současné polohy musí být využita technologie GPS, ke které má každé Android zařízení přístup. V mapě by se měla současná pozice zobrazovat jako bod, který půjde libovolně nastavit. Kromě pozice je třeba zobrazovat i současný směr zařízení. Pokud uživatel nepovolí aplikaci přístup k současné poloze, musí modul fungovat omezeně – bez navigace.

Navigace v mapě musí být dynamická v závislosti na současné poloze. Bez přístupu k současné poloze by měla být navigace nepřístupná. Hlavní priorita tohoto požadavku je vyhledání nejkratší cesty vhodným algoritmem od bodu současné pozice do cílového bodu a tuto cestu do mapy znázornit. Princip nalezení této cesty je ukázat uživatelovi ostatní body při cestě k destinaci.

Poslední požadavek je možnost lehce rozšířit funkcionalitu modulu vývojářem, který tuto knihovnu využívá. Cílem tohoto požadavku je umožnit aplikačním vývojářům naslouchat různým událostem, které automaticky vyvolává navigační modul. Například možnost reakce na uživatelský dotek prvku v mapě.

4.1.2 Návrh

Mapa

Kvůli náročnosti implementace vlastní technologie pro práci s mapou jsme vybrali již dostupnou technologii. Při výběru technologie pro mapu jsme cílili na rychlost, výpočetní nenáročnost, použitelnost a podmínky užití.

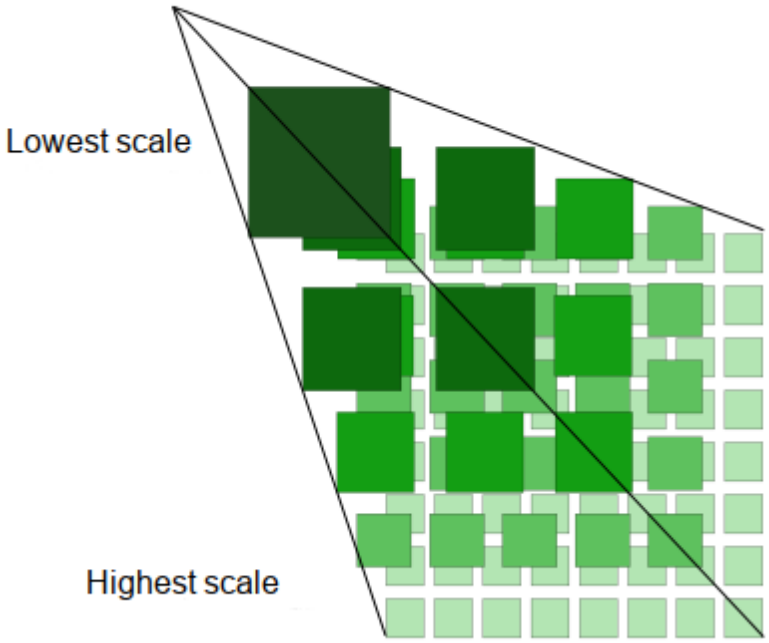
Z analýzy dostupných technologií použitelných pro Android aplikace vyplynuli kandidáti Google Maps API a MapView. Google Maps API je nejpopulárnější zobrazení mapy i navigace v ní pro Android zařízení. Obsahuje několik již zabudovaných funkcionalit, jako je například navigace v mapě se zobrazením nejkratších cest a dopravní vytíženost úseků v reálném čase. Google Maps API nejsou open source, a tudíž je zapotřebí licence podle typu používání (Google b.r.). Ovšem kvůli těmto komplexním zabudovaným funkcím nemůže nabízet pokročilé nastavení mapy, které jsou v tomto projektu vyžadovány. Naopak open source Android knihovna MapView neposkytuje funkce jako Google Maps API. Je velmi konfigurovatelná, což jí děla ideálním základem pro náš navigační modul.

Autor MapView, Pierre Laurence, představuje MapView jako rychlou, paměťově efektivní knihovnu pro Android, která zobrazuje mapu složenou z tzv. kachlí (tiles) s minimálním úsilím. Zobrazuje pouze viditelnou část mapy a podporuje pohyb, přetahování, změnu měřítka a otáčení. Je do ní také možné přidat značky a cesty. Původní projekt TileView byl napsán v programovacím jazyce Java, a později byl přepsán do jazyku Kotlin, čímž autor dosáhnul mnohem lepší výkonnosti (Laurence 2020).

Knihovna MapView řeší problém systému Android, který někdy nedokáže zobrazit příliš velký obrázek a dojde tak k chybě *OutOfMemoryError*. Řešením je rozdělit tento obrázek na několik částí tzv. kachlí a tyto kachle zobrazovat podle pohledu uživatele. Při přiblížení nebo posunutí v mapě uživatelem, se kachle mimo obrazovku recyklují, tudíž neblokují paměť zařízení. Knihovna dále optimalizuje zobrazování obrázků tzv. metodou subsample, která umožňuje šetření paměti při vykreslování většího obrázku do menší velikosti zobrazení (Dunn 2019), například snižuje využití paměti při načítání obrázku v rozlišení 1024x768 px do dispozice 128x96 px.

MapView pracuje s tzv. metodou deep-zoom, která umožňuje detailní přiblížení velké mapy bez vysoké spotřeby paměti a zobrazení více úrovní mapy (Laurence 2020). Za úroveň mapy považujeme hloubku přiblížení uživatelem, tudíž čím víc uživatel mapu přibližuje tím hlouběji se do mapy ponořuje. Touto funkcionalitou lze docílit vysokého detailu podkladu mapy bez přetížení paměti systému zařízení. V reálném užití lze tuto metodu využít k počátečnímu zobrazení jednoduché mapy a při přiblížení uživatelovi zobrazit více prvků v mapě. Funkce deep-zoom je graficky znázorněna v následujícím obrázku v tzv. pyramidě obrázků.

Obrázek 5: Princip metody deep-zoom

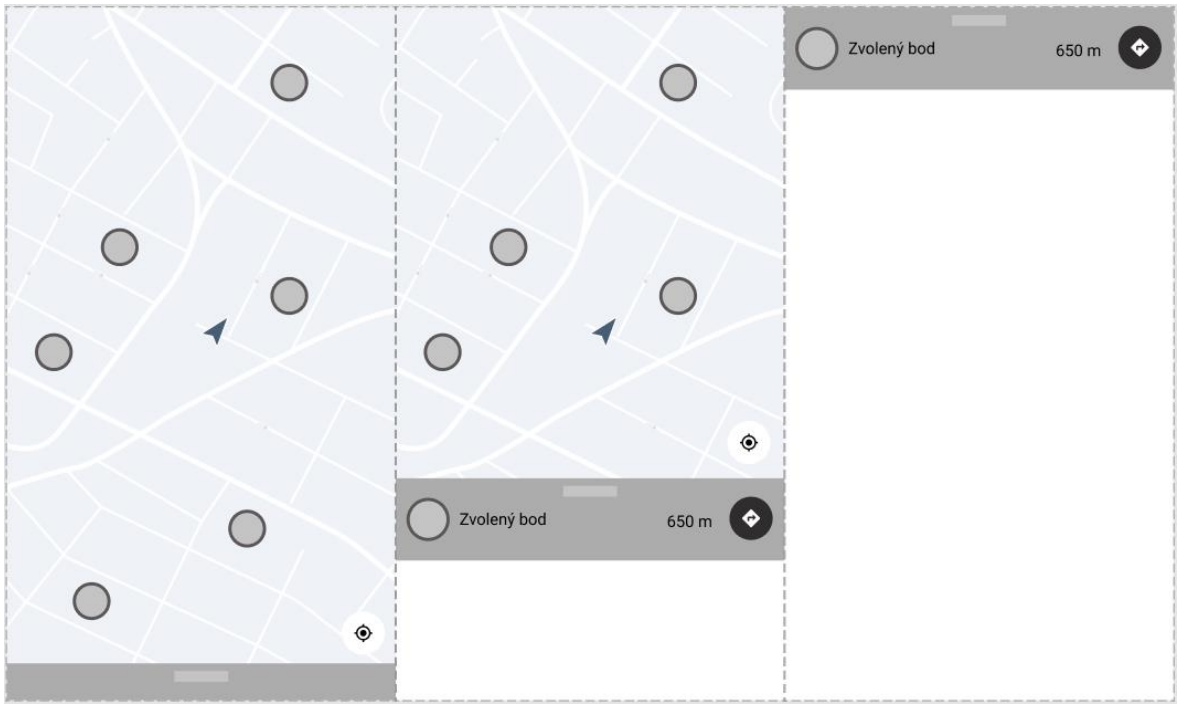


Zdroj: (Laurence 2020)

Body v mapě budou zobrazeny jako malé ikony s doporučenou velikostí 24x24 px. Při uživatelově přiblížení nebo přemístění v mapě musí body zůstat na svém místě a ve své původní velikosti. Při rotaci mapy musí ikony bodů reagovat tak, aby byly vždy čitelné pro uživatele, tj. ikona bodů se nachází ve svislé pozici. Rotace se nevztahuje na bod současné polohy. Po kliknutí na bod musí vývojář být schopen zachytit tuto událost i s informací o daném bodě.

Pro zobrazování detailu prvků v mapě je do modulu zakomponována komponenta s funkcí spodní vertikálně roztažitelné stránky, dále jen „spodní stránka“. Je důležité, aby tento prvek byl uživatelsky přívětivý a lehce modifikovatelný. Do této stránky bude moct vývojář vložit své grafické prvky pro zobrazení více informací, nejen k samotnému prvku.

Obrázek 6: Wireframe roztažené spodní stránky



Zdroj: vlastní zpracování

Záhlaví této spodní stránky bude z větší části neměnitelné. Budou se zde nacházet informace o současném prvku – ikona prvku a jeho obecný název; a tlačítko pro nalezení nejkratší cesty i s výslednou vzdáleností plynoucí z tohoto výpočtu. Při interakci uživatele se záhlavím bude možné změnit stav spodní stránky na jeden ze čtyř stavů – schovaná, složená, roztažená napůl a plně roztažená.

Současná poloha

V případě návrhu určení současné polohy je odpověď jednoznačná. Budeme používat přístup k poloze zprostředkované Android zařízením. Zařízení, na které tento modul cílí na smartphony, popřípadě tablety, by měli technologii určení polohy obsahovat. Přístup aplikacím k poloze Android nabízí již od API úrovně 1. Údaje o poloze obsahují konkrétní GPS souřadnice (šířka a délka), nadmořskou výšku a horizontální úhel směru zařízení – azimut, angl. *bearing*. Určení úhlu azimutu neumí určit každé zařízení (Google 2020a). S těmito údaji jsme schopni zaznačit bod současné polohy do mapy a dát mu i patřičný úhel.

Získání polohy na všech místech má svá úskalí. Současná poloha závisí z velké části na GPS, která je závislá na satelitním signálu. Tento signál funguje velmi dobře za ideálních podmínek, kdy se mezi satelity a zařízením nevyskytuje fyzický objekt. Kde nastává větší odchylka GPS polohy je například v budově, mezi vysokými budovami nebo v lese. Reálná přesnost polohy je v rozsahu 5-15 m. V případě smartphonů pro určení současné polohy napomáhá i WPS (WiFi positioning system) (Merry a Bettinger 2019). Přesnost současné polohy závisí hlavně na hardwaru zařízení a na jeho poloze v reálném světě. V případě potřeby zlepšení signálu mezi budovami lze využít i další IoT zařízení jako je například tzv. Bluetooth beacon. Ovšem přesnost signálu není předmětem tohoto modulu, ale toho, kdo tento modul využívá.

Kvůli nutnosti definice přístupu ke službám současné polohy v aplikačním manifestu, nebude modul získávat tuto polohu automaticky, ale bude jen zprostředkovávat výsledek, který mu vývojář předá. Vývojář bude tímto mít daleko větší kontrolu nad získáváním současné pozice zařízení. Bez přístupu k poloze bude modul fungovat omezeně. Nebude přístupná funkcionalita vyhledání nejkratší cesty a zaměření současné polohy. Takto lze využít modul jen jako zobrazení mapy a bodů v ní.

Hledání nejkratší cesty

Vyhledání nejkratší cesty v reálné mapě za pomocí dostupné technologie a bez vynaložení nadměrného úsilí na konfiguraci mapy nebude nikdy perfektně přesné. Snaha řešitele by měla být tuto chybovost co nejvíce omezit. Proto byl implementován algoritmus, který nevyžaduje detailní konfiguraci. Zvolený algoritmus nalezení nejkratší cesty sestává ze tří kroků.

První krok algoritmu má za cíl najít nejbližší bod k současnému bodu. Nejbližší bod *A* lze zjistit porovnáním výsledků výpočtem Pythagorovy věty na každý bod v rovině a následně výběrem nejmenší hodnoty. Tento krok počítá s dvojrozměrnou rovinou v kartézské sestavě souřadnic.

Druhý krok má za úkol spočítat vzdálenost mezi současným bodem a výsledným bodem z prvního kroku. Jelikož vycházíme z premisy, že se modul věnuje reálným mapám (povrch je zeměkoule), lze zapojit Haversinovu rovnici pro výpočet vzdálenosti mezi dvěma souřadnicemi.

Třetí krok zahrnuje nalezení nejkratší cesty výchozí z bodu *A* (výsledného bodu z prvního kroku) přes ostatní body v mapě do cílového bodu *B*. Tato úloha není problém obchodního cestujícího, protože nás zajímá pouze jedna cesta do cílového bodu. Takto formulované zadání je nejlepší řešit neorientovaným ohodnoceným grafem a v něm hledat nejkratší cestu přes jeho vrcholy.

Nejkratší cesta je důležitým tématem výzkumu v teorii grafů. Existuje několik algoritmů, které tento problém řeší, ovšem každý z nich má své využití. Rozlišují se na algoritmy nejkratší cesty s jedním zdrojem a nejkratší cesty s více zdroji. Nejkratší cesta jednoho zdroje je získání nejkratší cesty z daného vrcholu do jakéhokoli jiného vrcholu. Klasický algoritmus hledání nejkratší cesty s jedním zdrojem je Dijkstrův algoritmus, který vytvořil počítačový vědec E. W. Dijkstra. Další příklad tohoto typu je Bellman-Ford algoritmus, který má významný rozdíl od Dijkstrova algoritmu, lze jej použít na grafy se zápornými hranovými vahami, pokud graf neobsahuje žádný negativní cyklus dosažitelný ze zdrojového vrcholu. Typický algoritmus nejkratší cesty je ve váženém grafu je Floyd-Warshall algoritmus, který počítá vzdálenosti všech hran v grafu mezi sebou (Wang 2018).

Při výběrů byl kladen důraz na reálné využití v mapě a na rychlost výpočtu. Floyd-Warshall algoritmus byl zavržen kvůli nadbytečnému počítání nejkratších cest všech vrcholů v grafu navzájem. Dijkstrův a Bellman-Fordův algoritmy jsou si podobné, proto je vhodné je více prozkoumat. Autor komparativní studie těchto dvou algoritmů, Samah W.G. AbuSalim, uvádí následující tabulku jejich rychlostí:

Tabulka 2: Porovnání rychlosti algoritmů Dijkstra a Bellman-Ford

Počet vrcholů	Dijkstra (milisekundy)	Bellman-Ford (milisekundy)
5	1351	513
10	3724	756
50	2072	9577

Zdroj: (AbuSalim et al. 2020)

Z této tabulky je patrné, že Bellman-Ford je efektivnější pro menší počet vrcholů v grafu. Ovšem Dijkstrův algoritmus je více než 4,5krát efektivnější při 50 vrcholech než Bellman-Ford algoritmus. S těmito informacemi máme jednoznačného vítěze – Dijkstrův algoritmus. Byl vybrán hlavně kvůli své rychlosti výpočtu v grafu s více než 50 vrcholy. Je pravděpodobné, že Dijkstrův algoritmus bude i nadále efektivnější při rostoucím počtu vrcholů.

Celkovým výstupen našeho algoritmu, tj. kombinace Pythagorovy věty, Haversinovy rovnice a Dijkstrova algoritmu, musí být kolekce navštívených bodů, celková vzdálenost i vzdálenosti mezi jednotlivými navštívenými body. S těmito daty bude lehké zaznačit do mapy cestu a v případě potřeby i detailní informace o cestě.

Použitelnost modulu

Použití modulu bude cíleno na implementaci abstraktního fragmentu vývojářem. V tomto fragmentu se bude nacházet veškerá logika a díky tomuto návrhu půjde tuto Android knihovnu velmi lehce použít.

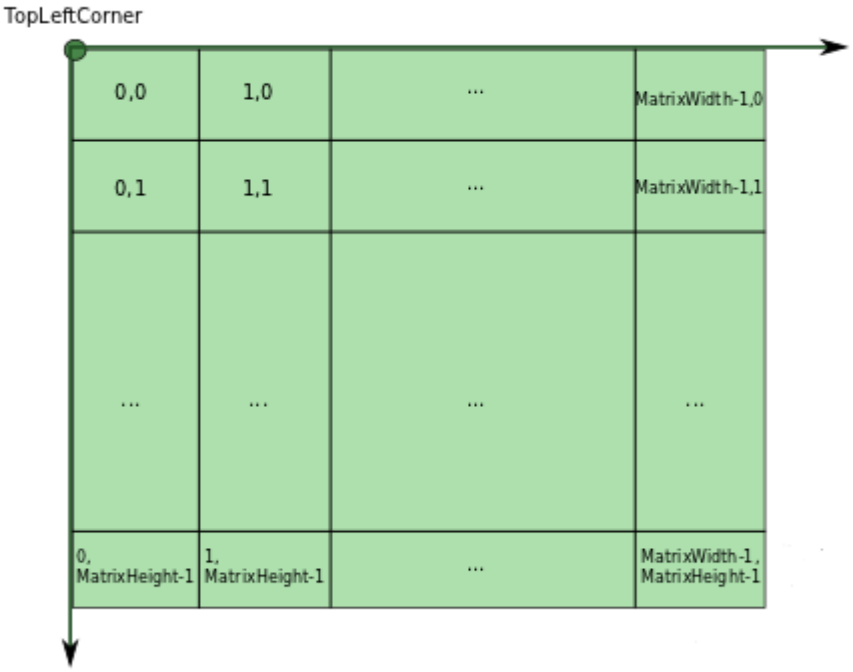
Přístupné operace nad modulem jsme klasifikovali do tří skupin – konfigurace mapy, API modulu a události spuštěné uživatelem. Konfigurace mapy je pro implementaci povinná, bez informací o mapě ji nelze zobrazit. Klíčovou funkcionalitu označujeme jako API modulu. Zde patří funkcionality jako naplnění mapu body, vytvoření grafu pro výpočet Dijkstrova algoritmu, aktualizování současné polohy atd. Na události uživatele v modulu nebude třeba reagovat, modul automaticky funguje s poskytnutými daty. Ovšem modul musí umožňovat rozšíření těchto událostí.

4.1.3 Implementace

Mapa

Jak již bylo zmíněno, MapView obecně řeší problém zbytečné spotřeby paměti při načítání velkých obrázků. V praxi toho docílíme jeho rozčleněním na několik menších částí o značně menší velikosti podle úrovně přiblížení a pozice v matice. Každá úroveň má poté svou matici grafických zdrojů. Knihovna MapView poskytuje tuto funkcionalitu třídou *TileStreamProvider*, která zprostředkovává tyto obrázky na základě pozice v matici. Následující obrázek přibližuje, jak tato funkcionalita vybírá podklad mapy.

Obrázek 7: Rozložení matice obrázků – tiles



Zdroj: (Laurence 2020)

Jako vstup je nutné poskytnout tzv. pyramidu obrázků – rozkrájené obrázky ve složkách jako úroveň přiblížení, číslo řádku jako podsložka a název obrázku jako číslo řádku. Aplikované například na pozici v matici a_{23} při úrovni přiblížení 1 bude *TileStreamProvider* hledat podklad ve složce ‚*1*' (dle úrovně), podsložce ‚*2*' a obraz ‚*3.jpg*'. Co se týče přiblížení tak platí vždy každé nové přiblížení by mělo mít dvakrát takový detail jako předchozí přiblížení, tím docílíme hladkého přiblížení mapě. Implementace této funkcionality v praxi vypadá následovně:

Kód 14 – Kotlin

```kotlin
val tileStreamProvider = TileStreamProvider { row, col, zoomLvl ->
  try {
    val tilePath = if (mapConfig.assetPathToMapTiles.endsWith("/")) {
      mapConfig.assetPathToMapTiles
    } else {
      mapConfig.assetPathToMapTiles.plus("/")
    }
    context.assets.open("$tilePath$zoomLvl/$row/$col.jpg") // img pyramid
  } catch (e: Exception) {
    null
  }
}
```

Zdroj: vlastní zpracování

S tímto mapovým podkladem můžeme přistoupit k zaznačení bodů do mapy. Naše mapa umí zobrazovat body, které jsou zastoupené třídou *MapMarker*. Tato třída nese informace o daném bodě.

Kód 15 – Kotlin

```
class MapMarker(var latitude: Double,
                var longitude: Double,
                val markerId: Any,
                val title: String,
                val entity: Any? = null,
                val markerIconResId: Int? = null,
                val clickable: Boolean = true,
                private val accessPoint: MapCoordinate? = null) {

  val accessLatitude: Double = latitude
    get() = accessPoint?.lat ?: field
  val accessLongitude: Double = longitude
    get() = accessPoint?.lng ?: field

  init {
    if (!(markerId is String || markerId is Number)) {
      throw IllegalArgumentException(
        "MapMarker.markerId must be String or Number,"
          + " provided: ${markerId.javaClass.name}")
    }
  }
}
```

Zdroj: vlastní zpracování

MapMarker udává zeměpisnou šířku a délku – *latitude*, *longitude*; identifikátor bodu (třída *Any* může být jakýkoliv objekt, ale my jej zde limitujeme na text nebo číslo) – *markerId*; identifikátor ikony, který zprostředkovává Resource Manager – *markerIconResId*; titulek bodu – *title*; data o entitě – *entity*; a zdali jde na bod kliknout – *clickable*. Speciální proměnná je *accessPoint* typu *MapCoordinate*, která slouží k určování přístupu k bodu v případě, kdy konec cesty má končit v jiném bodě než definuje jeho poloha v mapě. Třída *MapCoordinate* slouží pro zabalení zeměpisné šířky a délky pro jednodušší práci v mapě. Touto implementací docílíme jednoduchý přístup k datům při vyvolání události na bodu v mapě.

Velkou část logiky modulu jsme zakomponovali do třídy *MapigateFragment*. Ta uskupuje všechny prvky GUI v modulu a navazuje je na funkcionalitu modulu – vývojářské API i výchozí funkce modulu (reakce na uživatelův podnět). Drží v sobě také proměnné, které obsahují informace o stavu samotné mapy, bodech a cestách v ní. Kvůli rozsáhlosti kódu ve fragmentu *MapigateFragment*, jsou v této práci uvedeny jen jeho hlavní části.

Třída *MapigateFragment* rozšiřuje oficiální Android třídu *Fragment* a implementuje rozhraní *MapigateEventListener*. Fragment byl vybrán hlavně protože slouží jako kontejner pro rozšiřitelnou a znovupoužitelnou část kódu. Třída *MapigateFragment* je abstraktní, což nepovoluje přímé použití a nevyžaduje implementaci rozhraní (více popsáno v pozdější kapitole – Použitelnost). Protože rozšiřuje grafickou složku fragment, je nutné vytvořit tzv. XML layout. XML layout dává pokyn systému Android, jaké grafické prvky a v jakém stylu je má vykreslit na obrazovku pro uživatele. Grafické prvky se specifikují pomocí XML tagu a jejich styly jednotlivými atributy, které jsou definovány v XML tzv. namespacem *app* (viz xmlns:app) nebo *android*. Tyto prvky lze vytvořit a modifikovat i programaticky.

Pro implementaci spodní stránky jsme vytvořili XML layout *mapigate_bottom_sheet.xml*. Tento layout nese její grafické rozložení. Jednotlivé XML tagy zastupují třídy Android komponent.

Kód 16 – XML

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:android="http://schemas.android.com/apk/res/android"
  app:layout_behavior=
    "com.google.android.material.bottomsheet.BottomSheetBehavior"
  android:id="@+id/mapigateBSheet"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">
  <View
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white" />
  <LinearLayout
    android:id="@+id/mapigateBSheetHeader"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:paddingTop="5dp">
    <ImageView
      android:id="@+id/mapigateBSheetDragger"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:src="@drawable/map_sheet_dragger_up" />
    <include layout="@layout/map_bottom_sheet_header" />
  </LinearLayout>
  <RelativeLayout
    app:layout_behavior=
      "com.google.android.material.bottomsheet.BottomSheetBehavior"
    android:id="@+id/mapigateBSheetContent"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:layout_gravity="bottom"
    android:orientation="vertical"
    android:layout_below="@id/mapigateBSheetHeader">
  </RelativeLayout>
</RelativeLayout>
```

Zdroj: vlastní zpracování

Velmi důležitým prvkem je *mapigateBSheetContent* (třída *RelativeLayout*), který umožňuje přidat vývojářům svůj vlastní obsah do spodní stránky. Do předchozího XML layoutu je vnořen XML layout *mapigate_bottom_sheet_header.xml*, který konstruuje samotnou hlavičku spodní stránky. Tato hlavička obsahuje údaje o zaměřeném bodu například jeho ikonu nebo název.

Vnoření layoutu docílíme pomocí použití XML tagu <*include layout="..."/*>. Jeho XML layout je následující:

Kód 17 – XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:gravity="center">
  <ImageView
    android:id="@+id/mapigateHeaderIcon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="10dp" />
  <TextView
    android:id="@+id/mapigateHeaderName"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:gravity="left|center"
    android:layout_marginStart="10dp" />
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:gravity="end|center">
    <TextView
      android:id="@+id/mapigateSumDistText"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:paddingStart="5dp"
      android:paddingEnd="10dp" />
   <com.google.android.material.floatingactionbutton.FloatingActionButton
      android:id="@+id/mapigateDirButton"
      android:layout_width="wrap_content"
      android:layout_height="40dp"
      android:src="@drawable/map_btn_ic_directions"
      android:scaleType="center"
      android:layout_marginEnd="5dp"
      app:fabSize="mini" />
  </LinearLayout>
</LinearLayout>
```

Zdroj: vlastní zpracování

Prvky, se kterými potřebujeme pracovat v kódu, jsou označeny identifikátorem jako například *android:id="@+id/mapigateSumDistText"*. Posléze k nim lze jednoduše přistupovat pomocí Android identifikátoru '*R.id.mapigateSumDistText*' v kódu aplikace. Veškeré identifikátory XML prvků tohoto modulu jsou prefixovány textem *mapigate*, aby se předešlo kolizím v aplikacích využívajících tuto knihovnu. Takto sepsaný layout stačí inicializovat ve funkci

onCreateView z Android třídy *Fragment* a poté přiřadit jednotlivé grafické prvky do proměnných.

Současná poloha

Pro zaznačení současné polohy jsme vytvořily dvě metody, které umožňují vývojáři vložit bod aktuální polohy do mapy podle souřadnic a dát mu volitelnou rotaci. Metody nesou stejný název a liší se jen podle vstupních parametrů. Metoda, která přijímá objekt třídy *Location* existuje pro jednodušší použití poslední polohy zařízení od Google API Location service a volá funkci, kde se logika realizuje. Metody pracují s proměnnou *currPosMarker* typu *MapMarker*, která v našem modulu zastupuje bod současné polohy v mapě.

Kód 18 – Kotlin

```
import android.location.Location

fun updateCurrentPosition(location: Location) {
  updateCurrentPosition(location.longitude, location.latitude,
    location.bearing)
}

fun updateCurrentPosition(lng: Double, lat: Double, bearing: Float = 0f) {
  if (currPosMarker == null) {
    currPosMarker = mapView.addCurrPositionMarker(lng, lat)
  }
  mapView.moveMarker(currPosMarker!!, lng, lat)
  currPosMarker!!.mapMarker.latitude = lat
  currPosMarker!!.mapMarker.longitude = lng
  currPosRefOwner.angleDegree = bearing
}
```

Zdroj: vlastní zpracování

Nejkratší cesta

Podle návrhu z předchozí kapitoly je naimplementován algoritmus nejkratší cesty v omezené mapě. Algoritmus se skládá z následujících tří kroků:

1) **Pythagorova věta**

Nalezení nejbližšího bodu provedeme jednoduchým výpočtem pomocí Pythagorovy věty pro všechny body v mapě podle vzorce:

$$a^2 + b^2 = c^2 \dots d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \qquad (1)$$

Vzorec aplikujeme na všechny body v mapě v závislosti na současnou pozici určenou bodem (GPS souřadnice). Nejbližší bod zvolíme podle nejmenší hodnoty Pythagorovy věty.

Celkový výpočet realizujeme v iteraci přes kolekci bodů, kde si zároveň uchováváme bod do proměnné, jehož hodnota výpočtu Pythagorovy věty je nejmenší. Po dokončení iterací stačí tuto proměnnou s bodem vrátit jako výstup z funkce, což značí nejbližší bod k současné poloze.

Kód 19 – Kotlin

```kotlin
import kotlin.math.*

fun findNearestMarker(markers: List<MapMarker>, currLat: Double,
    currLng: Double): MapMarker {
  lateinit var nearestMarker: MapMarker
  var minDistance = Double.POSITIVE_INFINITY
  for (m in markers) {
    val distance = sqrt((currLat - m.accessLatitude).pow(2)
      + (currLng - m.accessLongitude).pow(2))
    if (minDistance > distance) {
      minDistance = distance
      nearestMarker = m
    }
  }
  return nearestMarker
}
```

Zdroj: vlastní zpracování

2) Haversinova rovnice

Druhý algoritmus je použit pro výpočet reálné vzdálenosti v kilometrech mezi bodem současné polohy a nejbližším bodem (výsledek z předchozího kroku). Tato rovnice je určena obecně pro výpočet vzdálenosti mezi dvěma body nacházejícími se na povrchu koule.

$$2r\ arcsin\left(\sqrt{sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right)+cos(\varphi_1)cos(\varphi_2)sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right) \quad (2)$$

Haversinova rovnice vyžaduje poloměr koule *r*. V našem případě použijeme střední poloměr zeměkoule, který dosahuje délky 6371 km. Tento parametr je nastaven pouze jak výchozí, tudíž při potřebě si vývojář může definovat svůj poloměr.

Díky pokročilé syntaxi a matematické knihovně *kotlin.math* programovacího jazyku Kotlin je implementace v kódu přehledná. Vzorec lze napsat i bez nutnosti vytvářet proměnné, ovšem to by zhoršilo čitelnost kódu.

Kód 20 – Kotlin

```kotlin
import kotlin.math.*

const val EARTH_RADIUS = 6371

fun haversineDistanceKm(lat1: Double, lng1: Double, lat2: Double,
    lng2: Double, radius: Int = EARTH_RADIUS): Double {
  val diffLat = Math.toRadians(lat2 - lat1)
  val diffLng = Math.toRadians(lng2 - lng1)
  val a = sin(diffLat / 2).pow(2) + cos(Math.toRadians(lat1))
    * cos(Math.toRadians(lat2)) * sin(diffLng / 2).pow(2)
  return 2 * radius * asin(sqrt(a))
}
```

Zdroj: vlastní zpracování

3) Dijkstrův algoritmus

Nejkomplikovanějším krokem našeho algoritmu je nalezení nejkratší cesty implementací Dijkstrova algoritmus v Kotlin kódu. Výstupem tohoto kroku musí být nejen kolekce navštívených bodů ale i vzdálenosti mezi nimi.

Pro implementaci Dijkstrova grafu byla vytvořena třída *DijkstraGraph*, která nese veškerou logiku ohledně vyhledávání nejkratší cesty mezi dvěma body v neorientovaném grafu. Zastupuje samotný graf, který obsahuje vrcholy a ohodnocené hrany.

Kód 21 – Kotlin

```
data class Edge(val from: MapMarker,
                val to: MapMarker,
                val value: Double)

class DijkstraGraph {
  private val edges = mutableSetOf<Edge>()
  private val vertices = mutableSetOf<MapMarker>()

  fun addLink(pair: Pair<MapMarker, MapMarker>, value: Double) {
    val (from, to) = pair
    vertices += from
    vertices += to

    edges.add(Edge(from, to, value))
    edges.add(Edge(to, from, value))
  }
  ...
}
```

Zdroj: vlastní zpracování

Pro použití grafu stačí vytvořit instanci této třídy a dále přidat jednotlivé spojení grafu pomocí metody *addLink*. Funkce *addLink* přidává uzly i hrany, a protože pracujeme s neorientovaným grafem, tak se uzly automaticky provážou oboustrannou hranou.

Dále přichází komplikovanější funkcionalita – samotné vyhledání nejkratší cesty. Kód níže se stále nachází ve třídě *DijkstraGraph*. Hlavní funkce, která spouští vyhledání se jmenuje *shortestPath* s dvěma parametry – začáteční a cílový bod. Zavoláním této funkce se zahájí výpočet, který začne v počátečním uzlu, postupně navštíví všechny ostatní uzly, přičemž plní nejmenší vzdálenost do uzlu pro každý z nich do programové mapy (klíč k hodnotě) *dists* a programovou mapu cest do určitého bodu *paths* (funkce *calcAdjacent* v kódu níže). Po navštívení všech relevantních uzlů je algoritmus ukončen a funkce navrátí výsledek. Kód 22 níže uvádí funkci *shortestPath* společně i s jejími pomocnými funkcemi.

Kód 22 – Kotlin

```kotlin
data class DijkstraResult(val markers: List<MapMarker>,
                          val distStepMap: Map<Any, Double>,
                          val totalDistance: Double)

class DijkstraGraph {
  private val edges = mutableSetOf<Edge>()
  private val vertices = mutableSetOf<MapMarker>()
  ...

  fun shortestPath(from: MapMarker, target: MapMarker): DijkstraResult {
    val unvisitedSet = vertices.toSet().toMutableSet()
    val dist = vertices.map {
      it.markerId to Double.POSITIVE_INFINITY }.toMap().toMutableMap()
    val paths = mutableMapOf<MapMarker, List<MapMarker>>()
    dists[from.markerId] = 0.0 // 0 to itself
    var curr = from // start from itself

    while (unvisitedSet.isNotEmpty() && unvisitedSet.contains(target)) {
      adjacentVertices(curr).forEach {
        adj -> calcAdjacent(curr, adj, dists, paths)
      }
      unvisitedSet.remove(curr)
      if (curr.markerId == target.markerId
          || unvisitedSet.all { dists[it.markerId]!!.isInfinite() }) {
        break
      }
      if (unvisitedSet.isNotEmpty()) {
        curr = unvisitedSet.minByOrNull { dists[it.markerId]!! }!!
      }
    }

    if (paths.containsKey(target)) {
      return dijkstraResult(target, paths[target]!!, dists)
    }
    return DijkstraResult(emptyList(), emptyMap(), 0.0) // no result
  }

  private fun calcAdjacent(curr: MapMarker, adj: MapMarker,
      dists: MutableMap<Any, Double>,
      paths: MutableMap<MapMarker, List<MapMarker>>) {
    val dist = getDistance(curr, adj)
    if (dists[curr.markerId]!! + dist < dists[adj.markerId]!!) {
      dists[adj.markerId] = dists[curr.markerId]!! + dist
      paths[adj] = paths.getOrDefault(curr, listOf(curr)) + listOf(adj)
    }
  }

  private fun dijkstraResult(target: MapMarker, markers: List<MapMarker>,
      dists: MutableMap<Any, Double>): DijkstraResult {
    val distStepMap = mutableMapOf<Any, Double>()
    var lastDist = 0.0
```

```
    var prevMarkerId: Any? = null
    for ((i, m) in markers.withIndex()) {
      if (i != 0) {
        val distStep = dists[m.markerId]!!
        distStepMap[prevMarkerId!!] = distStep - lastDist
        lastDist = distStep
      }
      prevMarkerId = m.markerId
    }
    return DijkstraResult(markers, distStepMap, dists[target.markerId]!!)
  }

  private fun adjacentVertices(marker: MapMarker): Set<MapMarker> {
    return edges.filter { it.from.markerId == marker.markerId }
      .map { it.to }.toSet()
  }

  private fun getDistance(from: MapMarker, to: MapMarker): Double {
    return edges.filter { it.from.markerId == from.markerId
        && it.to.markerId == to.markerId }.map { it.value }.first()
  }
}
```

Zdroj: vlastní zpracování

Nalezená cesta se konstruuje podle nastavení bodů. Samotný bod v mapě nemusí být zahrnut do vykreslené trasy, nebo může mít několik spojů *PathPoint* pro pokročilejší vykreslení cesty. Trasa vždy začíná v bodu současné polohy a pokračuje do nejbližšího bodu (dvojice parametrů *accessLongitude* a *accessLatitude*), ze kterého se napojí na hrany grafu. Styl vykreslení samotné cesty mapy záleží na implementaci vývojářem, může si zvolit styl cesty pomocí funkce *stylePath* z API knihovny. Níže se nacházejí funkce pro nalezení a následné vykreslení nejkratší cesty, které jsou přístupné v knihovně – vývojář je může využít i odděleně.

Kód 23 – Kotlin

```
fun findDijkstraPath(from: MapMarker, to: MapMarker): DijkstraResult {
  return if (from.markerId == to.markerId) {
    DijkstraResult(listOf(from), emptyMap(), 0.0)
  } else {
    dijkstraGraph.shortestPath(from, to)
  }
}

fun drawPath(markerList: List<MapMarker>,
    startFromCurrentPosition: Boolean = true) {
  val paths = mutableListOf<PathPoint>()
  if (startFromCurrentPosition && currPosMarker != null) {
    paths.add(PathPoint(currPosMarker!!.mapMarker.longitude,
      currPosMarker!!.mapMarker.latitude))
  }
  val lastIndex = markerList.size - 1
  for ((i, n) in markerList.withIndex()) {
    paths.add(PathPoint(n.accessLongitude, n.accessLatitude))
```

```
    if (i != lastIndex && pathMap.containsKey(n.markerId)) {
      pathMap[n.markerId]!![markerList[1 + i].markerId]?.let {
        it.forEach { p -> paths.add(PathPoint(p.lng, p.lat)) }
      }
    }
  }
  pathView.visibility = View.VISIBLE
  val pathStyle = stylePath() ?: defaultPathStyle
  pathView.updatePaths(listOfNotNull(paths.toFloatArray(mapView))
    .map { createDrawablePath(it, pathStyle) })
}
```

Zdroj: vlastní zpracování

V neposlední řadě stačí tyto tři algoritmy sloučit do jednoho výpočtu ve funkci *findShortestPath* a dané grafické prvky napojit na výsledek algoritmu. Konečné zapojení vyhledání nejkratší cesty na grafické tlačítko *directionButton* provedeme zavoláním metody *setOnClickListener* s argumentem naší funkce *findShortestPath*. Kromě nalezení a vykreslení cesty obsahuje funkce *findShortestPath* i další funkcionalitu, jako je srolování komponenty spodní stránky nebo přemístění na bod současné polohy v mapě.

Kód 24 – Kotlin

```
fun findShortestPath(marker: MapMarker) {
  if (dijkstraReady && currPosMarker != null) {
    val currLat = currPosMarker!!.mapMarker.latitude
    val currLng = currPosMarker!!.mapMarker.longitude
    val nearestMarker = findNearestMarker(
      markerViewMap.values.map { it.mapMarker }, currLat, currLng)
    val haversineDist = haversineDistanceKm(
      currLat, currLng, nearestMarker.latitude, nearestMarker.longitude)
    val dijkstraResult = findDijkstraPath(nearestMarker, marker)

    drawPath(dijkstraResult.markers)

    targetMarkerId = marker.markerId
    distTextToTarget = DistanceUtil.prettifyMeters(
      haversineDist * 1000 + dijkstraResult.totalDistance)
    sumDistText.text = distTextToTarget

    onShortestPathFind(dijkstraResult.markers,
      dijkstraResult.distanceStepMap, haversineDist,
      dijkstraResult.totalDistance)

    mapView.moveToMarker(currPosMarker!!, 1f, true)

    halfExpandBottomSheet()
  } else {
    showUnavailableTest()
  }
}
...
directionButton.setOnClickListener {
```

```
  if (clickedMarker != null) {
    findShortestPath(clickedMarker!!)
  }
}
```

Zdroj: vlastní zpracování

Použitelnost modulu / API

Jak již bylo zmíněno, navigační modul lze použít velmi jednoduše – rozšířením základní třídy fragmentu *MapigateFragment*. Toto lehké rozšíření umožňuje vývojářům velmi lehce zapojit veškerou logiku této Android knihovny. Interakce vývojáře s modulem je rozdělena na konfiguraci a volitelné reakce na podněty uživatele.

Hlavní třída *MapigateFragment* je označena klíčovým slovem *abstract*, což nám zajistí, že ji vývojář nebude moct instanciovat napřímo, ale bude muset vytvořit svou implementaci. Touto implementací získá přístup k veškeré funkcionalitě modulu a zde ji také může rozšiřovat.

Kód 25 – Kotlin

```
import android.graphics.*

data class MapConfiguration(
        val topLatitude: Double, val rightLongitude: Double,
        val bottomLatitude: Double, val leftLongitude: Double,
        val fullWidth: Int, val fullHeight: Int, val levelCount: Int,
        val tileSize: Int, val assetPathToMapTiles: String = "tiles/",
        val maxScale: Float, val enableRotation: Boolean = true,
        val currPositionMarkerResId: Int? = null)

data class PathStyle(val paint: Paint? = null, val width: Float? = null)

abstract fun configureMap(): MapConfiguration

open fun stylePath(): PathStyle? = null

open fun showUnavailableText() = Toast.makeText(activity,
  "Unavailable", Toast.LENGTH_SHORT).show()
```

Zdroj: vlastní zpracování

Nakonfigurovat mapu je povinná operace, kterou musí vývojář vykonat. Je realizována vytvořením funkce *configureMap*, která je označena klíčovým slovem *abstract*. Slovo *abstract* dělá z funkce povinnou pro implementaci při rozšíření. Tímto po vývojářovi vynutíme nezbytnou konfiguraci mapy. Pro přehlednost jsou data nastavení mapy zabalena do objektu *MapConfiguration*.

Protože jeden z cílů použitelnosti je jednoduché zapojení modulu, je poskytnuta výchozí implementace funkce *stylePath*, která stylizuje vyznačovanou cestu v mapě. Kotlin automaticky označuje třídy, funkce a proměnné jako uzavřené, což brání v jejich úpravě. Tuto úpravu chceme povolit, proto je třeba označit funkci *stylePath* slovem *open*, tak jazyk Kotlin povolí přepis funkce. Podobný přístup jsme zvolili u funkce *showUnavailableText*. Reakci na podněty vyvolané modulem lze odchytit pomocí předem definované funkce v rozhraní *MapigateEventListener*. Tímto návrhem docílíme nepovinnou implementaci.

Kód 26 – Kotlin

```kotlin
interface MapigateEventListener {
  fun onMapMarkerTap(marker: MapMarker) { }

  fun onCurrentPositionMarkerMarkerTap(lat: Int, lng: Int) { }

  fun initBottomSheetLayout(sheetContent: RelativeLayout { }

  fun onBottomSheetStateChange(state: Int) { }

  fun onShortestPathFind(markerList: List<MapMarker>,
    distanceStepMap: Map<Any, Double>, distanceToFirst: Double,
    dijkstraDistance: Double) { }
}
```

Zdroj: vlastní zpracování

Při potřebě rozšířit událost uživatelova kliknutí na bod v mapě, lze jednoduše implementovat třídu *onMarkerTap*. V takto implementované funkci lze zahrnout vlastní logiku na výstupu události, v tomto případě konkrétní zvolený bod v mapě.

Kód 27 – Kotlin

```kotlin
class MapFragment: MapigateFragment() {
  ...
  override fun onMapMarkerTap(marker: MapMarker) {
    // implementation
  }
}
```

Zdroj: vlastní zpracování

Poslední část použitelnosti je samotné API Mapigate, které umožňuje vyvolat akce nad modulem. Toto API je určeno pro vývojáře, když potřebuje například naplnit mapu, vytvořit graf nebo manuálně přemístit uživatele v mapě. Níže jsou uvedeny vybrané funkce tohoto API s jejich implementací:

- *populateMap* – Základní funkce, která plní body do mapy. Dává možnost vývojářovi odstranit předchozí body v mapě a rovnou je naplnit nanovo (argument *clearOldMarkers*).

Kód 28 – Kotlin

```kotlin
fun populateMap(markers: List<MapMarker>,
    clearOldMarkers: Boolean = false) {
  if (clearOldMarkers) {
    clearMapMarkers()
  }
  for (m in markers) {
    val mView = MapMarkerView(requireContext(), m)
    markerViewMap[m.markerId] = mView
    mapView.addMarker(mView, m.longitude, m.latitude, -0.5f, -0.5f)
  }
}
```

Zdroj: vlastní zpracování

- *initDijkstra* – Povinná funkce pro fungování vyhledání nejkratší cesty. Tato funkce vytvoří graf, spojí body hranami s váhou a připraví body pro vykreslení cesty. Pro lehké zapojení bylo vytvořeno rozhraní *IMapPath*, které usnadňuje plnění mapy. Zastupuje objekty hrany neorientovaného grafu. Obsahuje informace nutné pro výpočet nejkratší cesty i volitelné informace pro speciální vykreslení cesty do mapy (proměnná *pathList*).

Kód 29 – Kotlin

```
fun initDijkstra(dijkstraPaths: List<IMapPath>) {
  dijkstraGraph = DijkstraGraph()
  for (dp in dijkstraPaths) {
    if (markerViewMap.containsKey(dp.fromMarkerId)
        && markerViewMap.containsKey(dp.toMarkerId)) {
      dijkstraGraph.addLink(
        markerViewMap[dp.fromMarkerId]!!.mapMarker to
        markerViewMap[dp.toMarkerId]!!.mapMarker, dp.distanceInMeters)

      if (dp.pathList != null) {
        val fromMap = pathMap.getOrPut(dp.fromMarkerId) { HashMap() }
        fromMap[dp.toMarkerId] = dp.pathList!!
        val toMap = pathMap.getOrPut(dp.toMarkerId) { HashMap() }
        toMap[dp.fromMarkerId] = ArrayList(dp.pathList!!).reversed()
      }
    }
  }
  dijkstraReady = true
}
```

Zdroj: vlastní zpracování

- *moveToMarkerById* – Přemístí uživatele na bod podle zadaného identifikátoru (jestli takový bod v mapě existuje) s cílovým přiblížením *destScale* a možností animace *animate* cesty od současného stavu mapy do cíle.

Kód 30 – Kotlin

```
fun moveToMarkerById(markerId: Any, destScale: Float = 1f,
    animate: Boolean = true): Boolean {
  if (markerViewMap.containsKey(markerId)) {
    mapView.moveToMarker(markerViewMap[markerId]!!, destScale, animate)
    return true
  }
  return false
}
```

Zdroj: vlastní zpracování

Distribuce

Před publikací modulu pro veřejnost je vhodné zvolit licenci kódu, aby jej ostatní mohli používat bez komplikací. Protože je žádoucí, aby modul byl přístupný pro jakékoliv užití, zvolili jsme licenci Apache License 2.0. Tato licence je definována jako: „*Povolující licence, jejíž hlavní podmínky vyžadují zachování autorských práv a licenčních oznámení. Přispěvatelé poskytují výslovné udělení patentových práv. Licencovaná díla, úpravy a větší díla mohou být*

distribuována za různých podmínek a bez zdrojového kódu“. Apache License 2.0 dále povoluje bezplatné osobní i komerční užití (ChooseALicense.com 2020).

Přístup vývojářům k modulu Mapigate poskytneme jeho publikováním do veřejného úložiště balíčků tzv. package repository. Takto publikovanou verzi našeho modulu lze velmi jednoduše použít v jiné aplikaci. Jako package repisotory jsme vybrali JitPack, který nám umožní knihovnu zkompilovat a poté distribuovat. JitPack jsme vybrali kvůli jednoduchému propojení s GitHub repositářem, kde stačí udělat nový tzv. release (git tag) pro publikaci nové verze. Vývojář si posléze může stáhnout tuto publikaci pomocí nástroje pro automatizaci sestavování programu – Gradle (v případě Androidu je volba nástroje Gradle jednoznačná).

Kód 31 – Groovy (Gradle)

```
repositories {
  ...
  maven { url 'https://jitpack.io' }
}

dependencies {
  implementation 'com.github.unbearables:mapigate:v1.0.3'
}
```

Zdroj: vlastní zpracování

V teoretické části bylo zmíněno, že programovací jazyky Java a Kotlin jsou plně interoperabilní. Z toho plyne, že tato Android knihovna může být použita pro aplikace, napsané v obou těchto jazycích.

4.1.4 Testování

Správná funkcionalita bude zajištěna jednotkovými testy. Není vždy potřeba otestovat 100 % kódu, stačí otestovat části, které jsou náchylné chybám. V našem případě je to hlavně algoritmus hledání nejkratší cesty – Pythagorova věta, Haversinova rovnice a Dijkstrův algoritmus.

Kód 32 – Kotlin

```kotlin
@Test
fun shortestPathTest() {
  val dijkstraGraph = DijkstraGraph()
  val m1 = MapMarker(1.0, 1.0, 1, "1")
  val m2 = MapMarker(2.0, 2.0, 2, "2")
  val m3 = MapMarker(5.0, 5.0, 3, "3")
  val m4 = MapMarker(3.0, 3.0, 4, "4")

  dijkstraGraph.addLink(m1 to m2, 1.0)
  dijkstraGraph.addLink(m1 to m3, 5.0)
  dijkstraGraph.addLink(m2 to m3, 1.0)
  dijkstraGraph.addLink(m2 to m4, 2.0)

  val result = dijkstraGraph.shortestPath(m1, m4)
  assertEquals(result.totalDistance, 3.0, 0.0)
  assertArrayEquals(result.markers.toTypedArray(), arrayOf(m1, m2, m4))
}
```

Zdroj: vlastní zpracování

Kvůli obtížnosti instrumented testů (Mapigate knihovna nemá aktivitu), nebyly automatizovány testy grafického prostředí fragmentu.

4.2 Aplikace pro Zoo Praha

Tato část popisuje analýzu, návrh, implementaci a testování Android aplikace pro Zoo Praha. Cíl aplikace je demonstrovat reálné použití navigačního modulu popsaného v předchozí kapitole. Tato aplikace nebude reálně publikovatelná, kvůli obtížnosti realizace vlastního RESTful API serveru.

4.2.1 Analýza

Celkově by aplikace měla cílit na zobrazení podobných informací, které zobrazuje již existující webová aplikace Zoo Praha. Kromě samotných zvířat jsme se rozhodli přidat do aplikace novinky o zoo. Tímto návštěvníkům Zoo Praha nabídneme mapu se zvířaty, zajímavé novinky a statické informace například jak Zoo Praha pomoci.

Hlavní město Prahy od roku 2015 usiluje o otevření dat svých institucí v oborech, kde je to možné. Přístup otevřených dat umožňuje vývojářům tyto data čerpat a posléze s nimi pracovat (Magistrát hlavního města Prahy b.r.). V současné době tzv. datasety města Prahy obsahují například jízdní řády PID, seznam dětských hřišť jednotlivých částí Prahy a čerpání rozpočtu. Mezi instituce hlavního města Prahy patří i Zoo Praha, která otevření svých dat plánuje. Bohužel v době, kdy byla tato aplikace implementována, nebyli data ze zoologické zahrady přístupné pro využití. Ovšem i přes přístup k těmto datům by informace nestačili pro vykreslení námi implementované mapy s vyhledáváním nejkratší cesty. Z těchto důvodů plyne nutnost vytvoření backend serveru, který bude tyto data zprostředkovávat pro naši mobilní aplikaci. Informace o zvířatech je možné čerpat z webových stránek sekce lexikon zvířat, kde nalezneme popis zvířete, vědeckou klasifikaci, obrázek i zajímavosti o zvířeti. Novinky pak lze čerpat ze sekce Aktuálně.

Požadavky na aplikaci míří hlavně na funkci digitalizované mapy, vyhledávání nejkratší cesty ke zvířeti a zobrazování jeho detailu. Vedlejší funkcionalita jsou poté novinky ze Zoo Praha – jejich přehled i detail. V aplikaci nebude chybět také úvod do aplikace a zobrazení krátkého textu o jejím vypracování.

4.2.2 Návrh

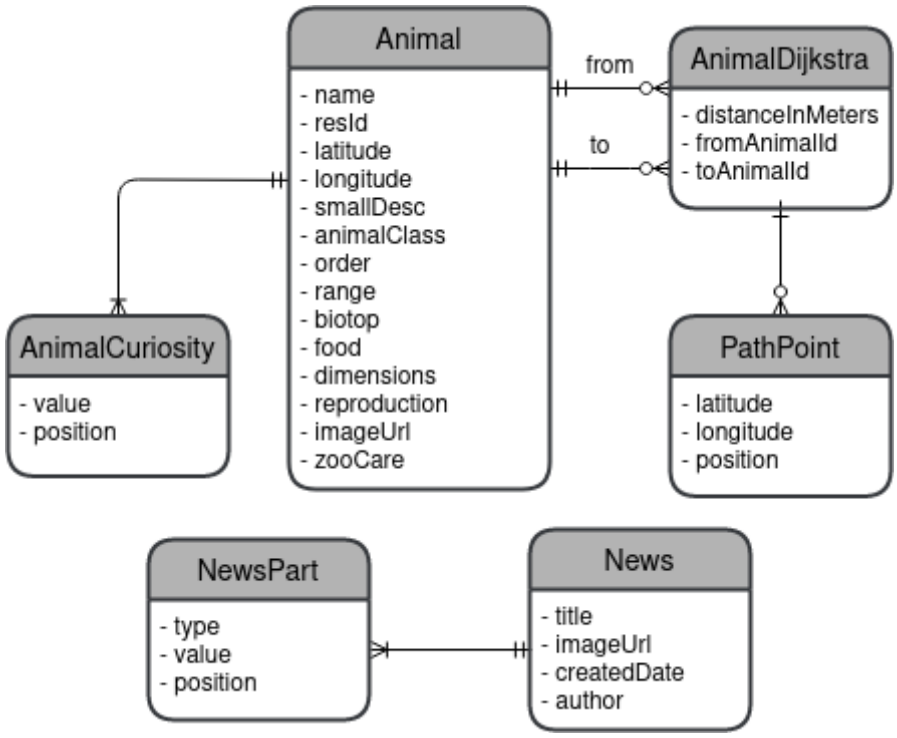
Návrh aplikace jsme zprostředkovali pomocí nástroje WebML. Tento nástroj představuje dohromady čtyři modely, které byly vysvětleny v teoretické části této práce. Protože není potřeba přizpůsobovat vzhled aplikace tak jsme vynechali uživatelský model.

Přínos návrhu za pomocí WebML je přehledné propojení entit definované ER diagramem s grafickou částí aplikace – reprezentovány hypertextovým a prezentačním modelem. Hlavní model notace WebML je hypertextový model, který nám umožní přehledně propojit jednotlivé stránky a lehce naznačit jejich obsah. Prezentační model bude zkonstruován pomocí tzv. wireframů více viz kapitola prezentační model.

Strukturální model

První krok návrhu pomocí notace WebML je vytvořit datovou strukturu. My jsme pro znázornění datové struktury zvolili ERD.

Obrázek 8: ERD



Zdroj: vlastní zpracování

Výše uvedený ERD obsahuje dohromady šest entit. Entity uvedené výše jsou pojmenovány anglicky, jejich význam je následující:

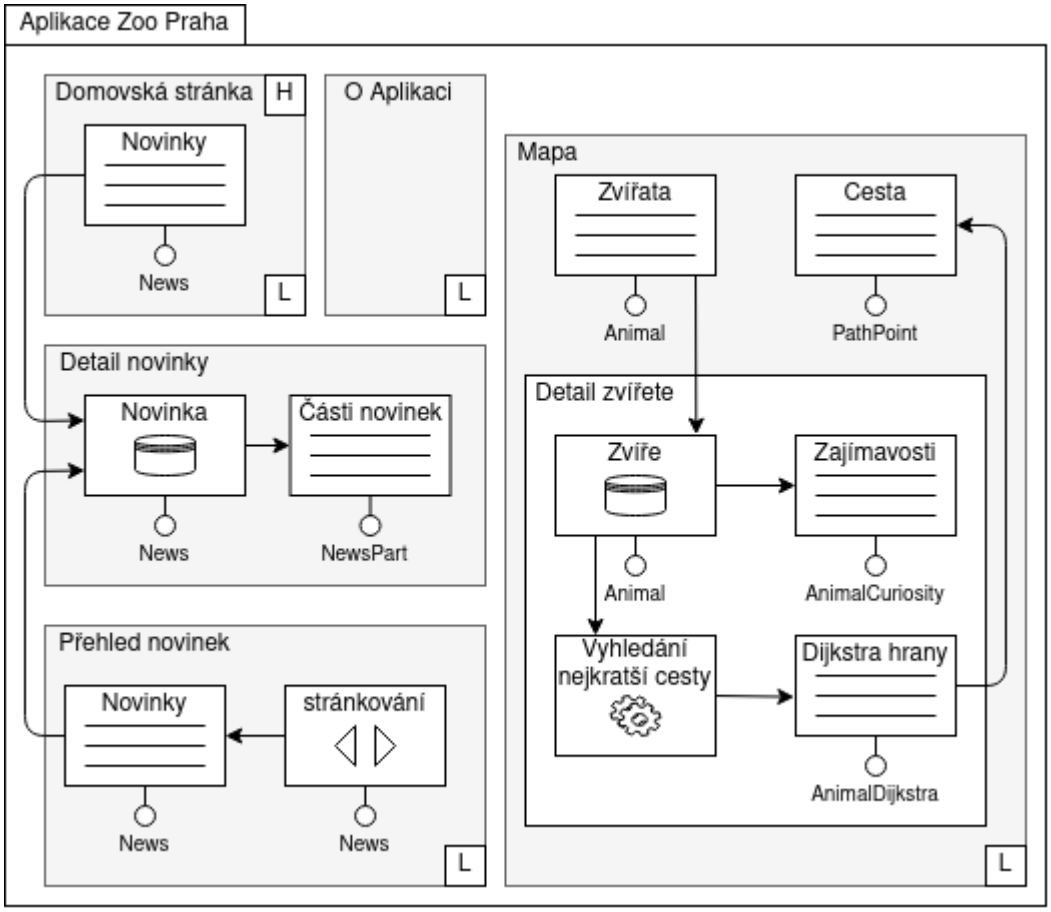
- **Animal** – Zvíře; jedna entita reprezentuje jeden druh zvířete ve výběhu, celý pavilon nebo jinou budovu sdružující mnoho zvířat. Tato entita nese informace o zvířeti.
- **AnimalCuriosity** – Zajímavost o zvířeti; jednoduchá věta se zajímavostí o konkrétním zvířeti. Zvíře má vždy alespoň jednu zajímavost.
- **AnimalDijkstra** – Hrana Dijkstrova grafu; jeden záznam vypovídá o spojení dvou zvířat (uzlů grafu) hranou. Zde se nacházejí nejobtížnější kardinality s entitou *Animal*. Každé zvíře by mělo být provázané do grafu buď přes kardinalitu *from* (*fromAnimalId*) nebo *to* (*toAnimalId*) – tudíž alespoň jednu hranu v grafu. Proto je u obou vazeb uvedena kardinalita *1–0..**; slovně: jedno zvíře může mít 0 až N hran.
- **PathPoint** – Bod cesty; souřadnice, která udává rozšiřitelné volitelné vykreslení cesty v mapě pro hranu v grafu (*AnimalDijkstra*).
- **News** – Novinka; záznam novinky vydané od Zoo Praha. Tato entita nenese samotný článek.

- **NewsPart** – Část novinky; jedna entita reprezentuje úsek novinky například text, obrázek nebo titulek. Novinka (*News*) musí mít alespoň jednu část.

Hypertextový model

Dále jsme vytvořili hypertextový model, který se skládá z modelu kompozice a navigačního modelu. Model kompozice nám umožní abstraktně navrhnout stránku bez nutnosti zabývat se detaily. Navigační model poté definuje navigaci v aplikaci. V modelu níže jsme definovali pět stránek, z čehož stránka Mapa obsahuje vnořenou stránku tzv. substránku Detail zvířete, podle návrhu Mapigate modulu spodní stránky. Čtyři z těchto stránek jsou označeny značkou *L* jako landmark, což je ve WebML označení pro stránku, která je přístupná z jakékoliv části aplikace. Stránka O Aplikaci neobsahuje žádnou kompozici, protože nese jen statické informace. WebML nám umožňuje definovat vlastní kompozice, kde je to nutné (Ceri, Fraternali, a Bongio 2000). Pro přehlednost modelu jsme definovali vlastní kompozici pro operaci vyhledání nejkratší cesty v mapě – označena ozubeným kolem.

Obrázek 9: WebML – hypertextový model

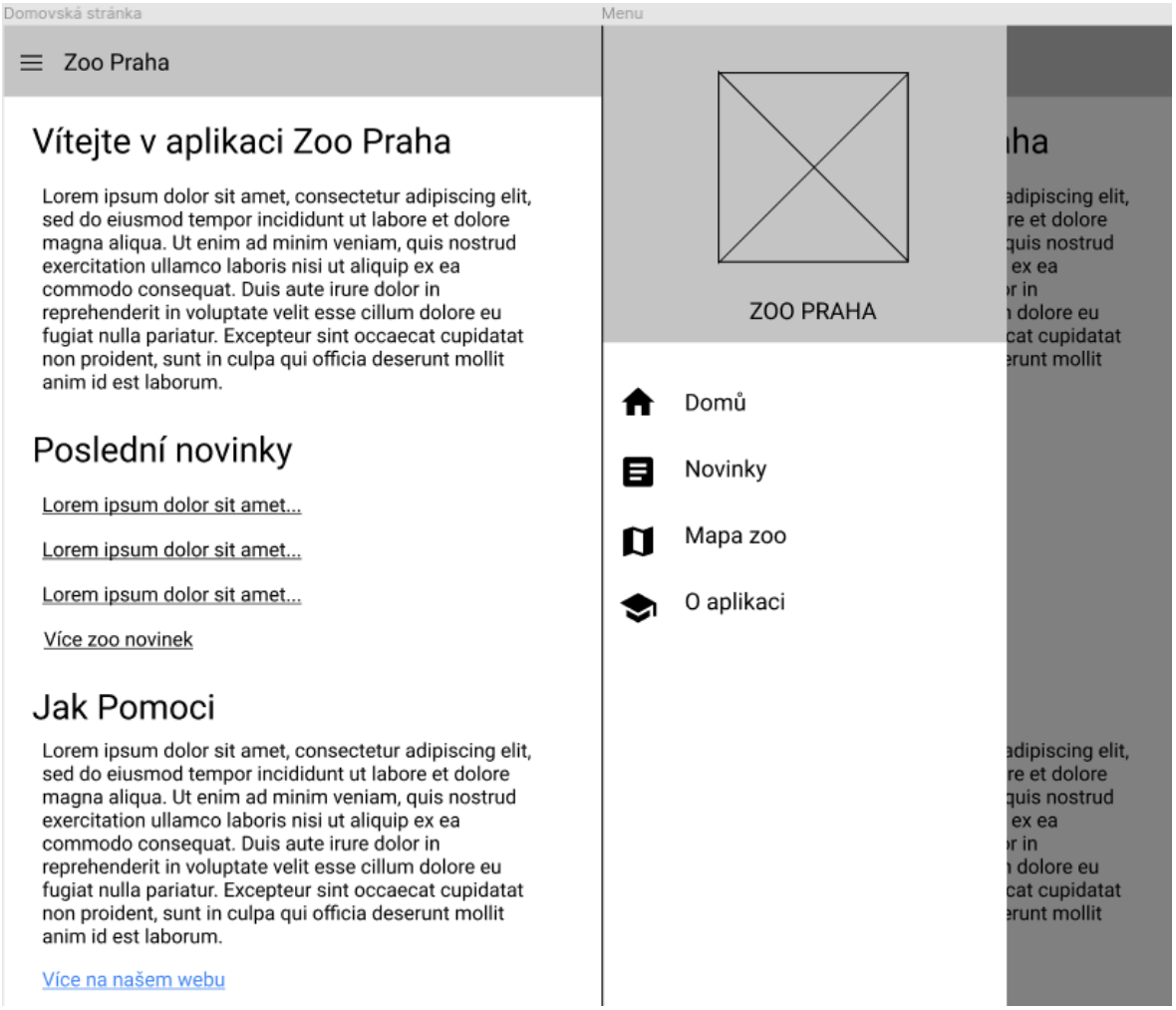


Zdroj: vlastní zpracování

Prezentační model

Předchozí model jsme následně interpretovali do prezentačního modelu. Každé stránce jsme dali svůj vzhled znázorněný wireframem tj. jednoduchá reprezentace stránky a její grafických komponentů. Samotný vzhled mapy zde není uveden, protože z grafického hlediska je to jen obrázek s body. Prezentační model neudává finální vzhled aplikace, pouze návrh na grafický design, proto se konečná implementace může lišit od návrhu.

Obrázek 10: Wireframy Domovská stránka a Menu

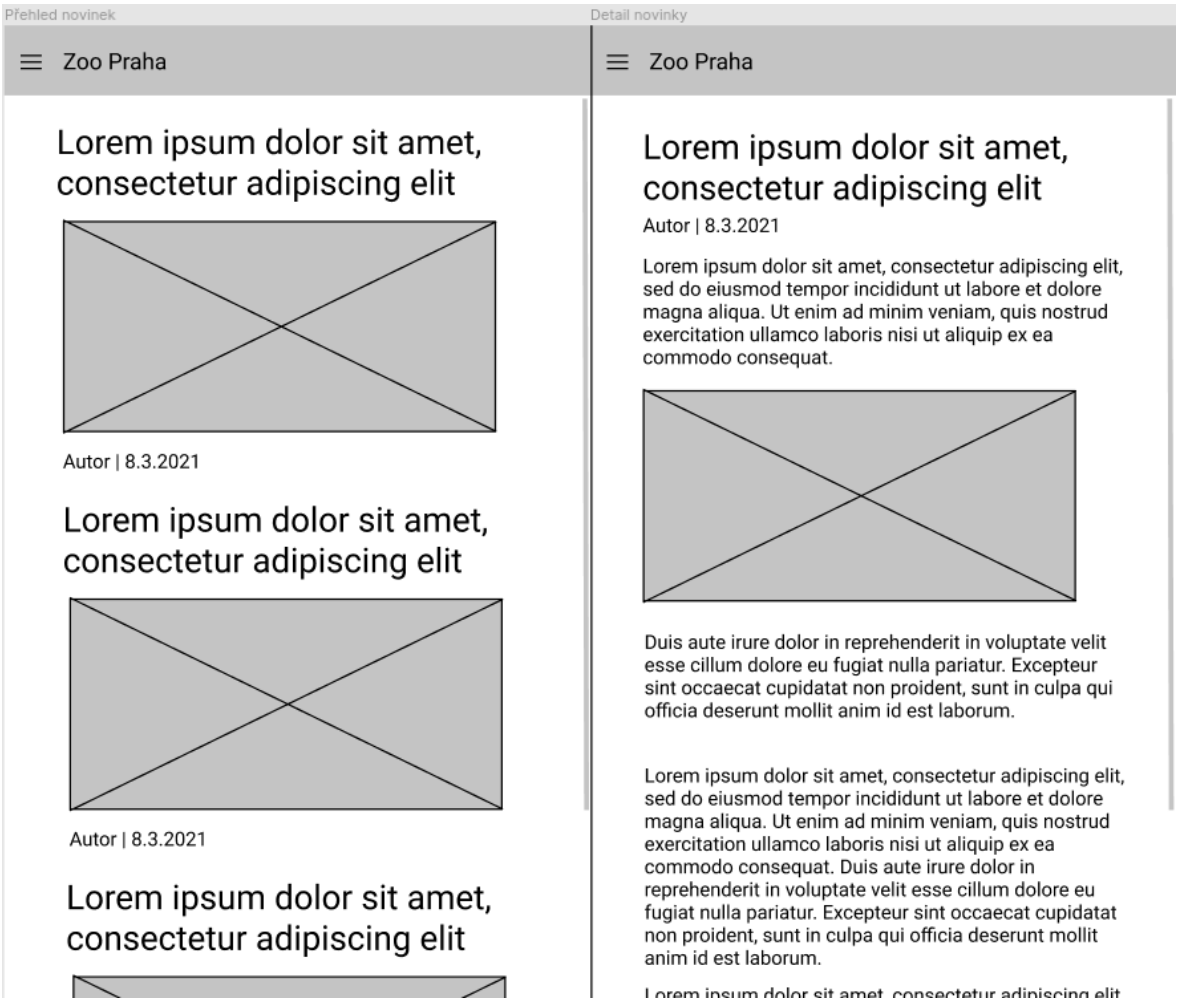


Zdroj: vlastní zpracování

Wireframe Domovská stránka (obrázek 10 vlevo) znázorňuje Domovskou stránku. Vítá uživatele v aplikaci a zobrazuje tři nejnovější novinky s možností prokliku na danou novinku i odkaz na zobrazení stránky s novinkami. Nakonec informuje uživatele, jak může organizaci Zoo Praha pomoci.

Wireframe Menu (obrázek 10 vpravo) propojuje čtyři definované landmarky ve formě všudypřítomného menu. Toto menu je navrženo tak, aby bylo dostupné všude v aplikaci za pomocí kliku na ikonu menu v aplikační liště nebo gestem přejetím prstu zleva na druhou stranu.

Obrázek 11: Wireframy Přehled novinek a Detail novinky

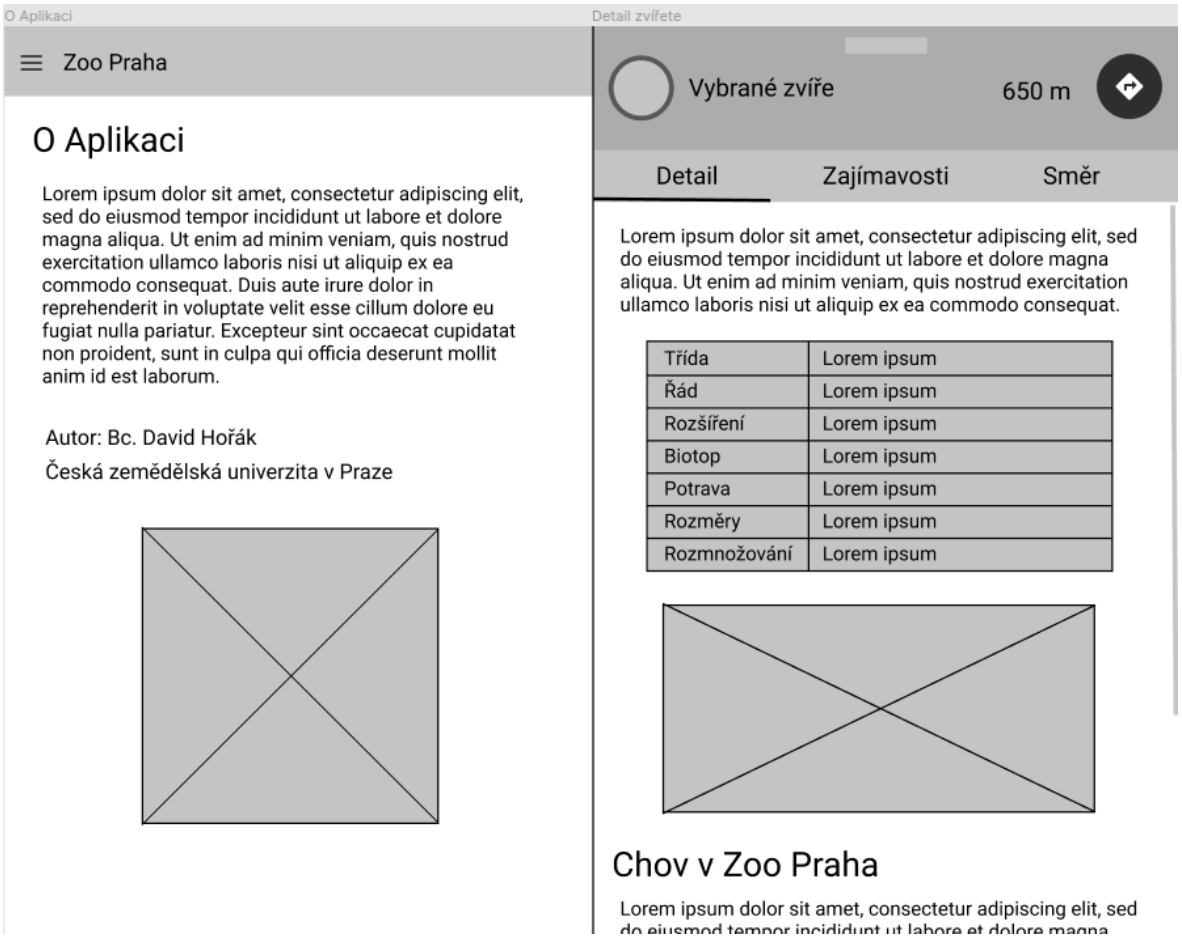


Zdroj: vlastní zpracování

Wireframe Přehled novinek (obrázek 11 vlevo) nabízí náhled na nejnovější novinky, které se načítají podle uživatelovy interakce se scrollem – vždy, když dosáhne konce stránky, načtou se další zprávy tzv. infinite scroll (lazy load).

Wireframe Detail novinky (obrázek 11 vpravo) již podle názvu zobrazuje detail novinky – titulek, autor i čas článku přímo z entity *News* a obsah ve formě dynamických částí novinek *NewsPart*.

Obrázek 12: Wireframy O Aplikaci a Detail zvířete

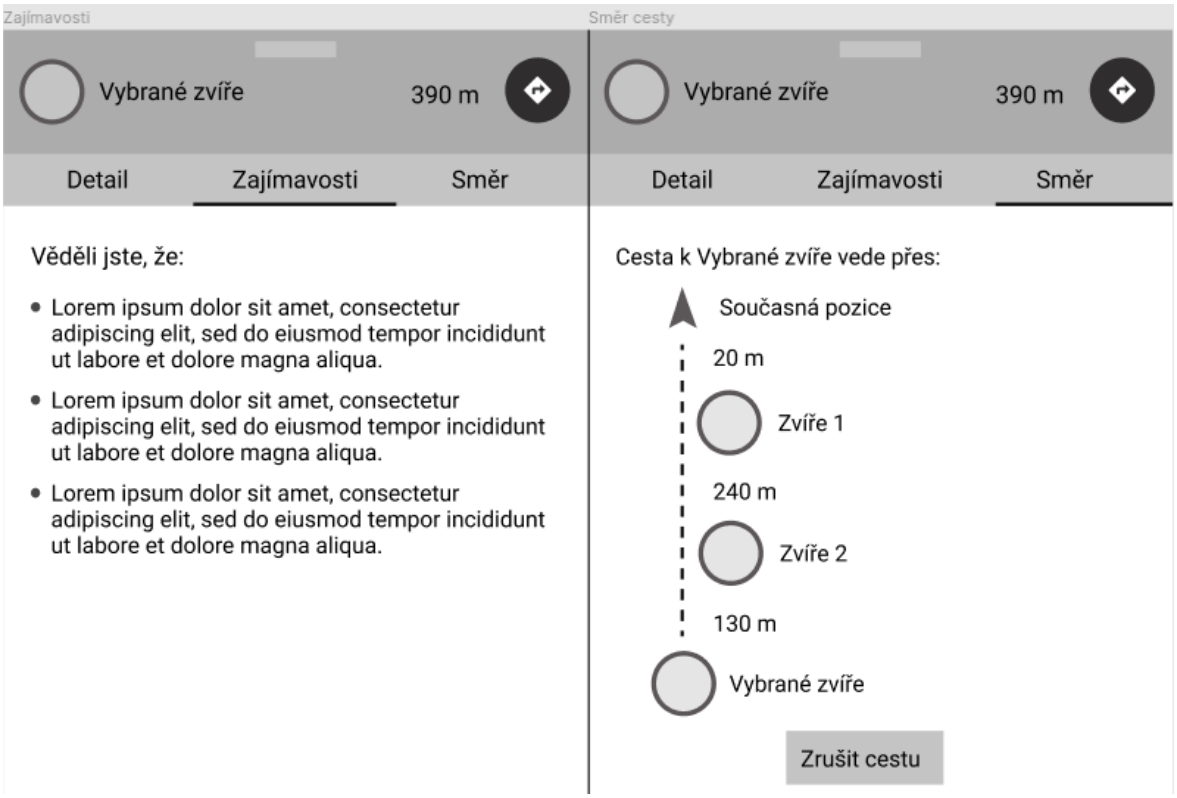


Zdroj: vlastní zpracování

Wireframe O Aplikaci (obrázek 12 vlevo) je pouze statická stránka s informacemi o aplikaci a jejímu účelu.

Wireframe Detail zvířete (obrázek 12 vpravo) je nejdůležitější a pravděpodobně bude i nejnavštěvovanější pohled v aplikaci. Wireframy Detail zvířete, Zajímavosti a Směr cesty (zmíněno níže) jsou navrženy jako záložky ve spodní stránce Mapigate modulu, které lze jednoduše změnit výšku. Takto uživateli dodáváme příjemný UX design – ovládání jednou rukou, intuitivní scroll apod. Pohled samotný nese hlavní informace o zvířeti jako popis, tabulku s vědeckou klasifikací a krátký popis, jak je o zvířeti postaráno v Zoo Praha.

Obrázek 13: Wireframy Zajímavosti a Směr cesty



Zdroj: vlastní zpracování

Wireframe Zajímavosti (obrázek 13 vlevo) jednoduše zobrazuje zajímavosti o zvířeti v odrážkovém seznamu. Podle obsahu pohled obsahuje rolovací lištu, tzv. scroll.

Wireframe Směr cesty (obrázek 13 vpravo) je detailní pohled na vybranou cestu k cíli. Po vyhledání nejkratší cesty uživatelem (stisknutí tlačítka vpravo nahoře) se naplní tato záložka jejím detailem. Detail cesty se skládá z jednotlivých zvířat, kolem kterých uživatel projde a jejich vzdálenost mezi nimi. Nakonec se zde nachází tlačítko pro zrušení cesty, které resetuje obsah této záložky a vyznačenou cestu v mapě schová.

Mapa

Protože implementujeme aplikaci pro Zoo Praha, máme místo mapy jednoznačně dané – areál Zoo Praha v Troji. V hrubém měření v mapě je tento areál dlouhý 800 m a široký 1250 m, celkem 58 ha rozlohy, což vyhovuje podmínkám použití našeho Mapigate modulu. Zvířata v zoologické zahradě budou reprezentována jednotlivými body v mapě a uzly v grafu. Hrany grafu budou cesty pro návštěvníky a jejich vykreslení bude doplněno speciálními body *PathPoint*.

V případě určení nejbližšího bodu v mapě v závislosti k současné poloze se není třeba obávat. Zoo Praha uvádí, že se v zde nachází více než 150 expozic (Zoo Praha 2020), což plně pokrývá plochu jejího areál. Když si představíme, že stojíme kdekoliv v zoo na návštěvnické stezce, tak je velká pravděpodobnost, že stojíme přímo u některého ze zvířecích výběhů. S tímto předpokladem lze s jistotou prohlásit, že výpočet nejkratší cesty Mapigate knihovny bude minimálně chybový.

4.2.3 Implementace

Zapojení Navigačního modulu

Pro přístup k modulu stačí v našem Gradle souboru *build.gradle* registrovat package registry JitPack a přidat závislost *'com.github.unbearables:mapigate:xxx'*, kde xxx je verze knihovny (viz předchozí kapitola Použitelnost).

Kód 33 – Kotlin

```kotlin
@AndroidEntryPoint
class ZooMapFragment: MapigateFragment() {
  @Inject
  lateinit var animalService: AnimalService
  private lateinit var animalAdapter: AnimalPagerAdapter
  private lateinit var animalTabLayout: TabLayout
  ...

  override fun onCreateView(inflater: LayoutInflater,
      container: ViewGroup?, savedInstance: Bundle?): View? {
    return super.onCreateView(inflater, container, savedInstance).also {
      fillMap()
    }
  }

  override fun configureMap(): MapConfiguration = MapConfiguration(
    50.1218811, 14.4130258, 50.1141575, 14.3979947, // coordinates
    3815, 3085, 5, 256, 4f)

  override fun initBottomSheetLayout(sheetContent: RelativeLayout) {
    val content = LayoutInflater.from(context)
      .inflate(R.layout.map_animal_bottom_sheet_content, sheetContent)
    animalTabLayout = content.findViewById(R.id.animalTabLayout)
    val pager: ViewPager2 = content.findViewById(R.id.animalTabPager)
    animalAdapter = AnimalPagerAdapter(requireActivity(), this, 3)
    pager.adapter = animalAdapter
    pager.registerOnPageChangeCallback(object : OnPageChangeCallback() {
      override fun onPageSelected(position: Int) {
        animalTabLayout.getTabAt(position)!!.select()
        animalAdapter.tabChange(position)
      }
    })

    animalTabLayout.addOnTabSelectedListener(
      object : TabLayout.OnTabSelectedListener {
        override fun onTabSelected(tab: TabLayout.Tab) {
          pager.currentItem = tab.position
        }
      })
    }
  }
}
```

Zdroj: vlastní zpracování

V Kódu 33 výše jsme implementovali fragment *MapigateFragment* a navázali jsme funkcionalitu záložek na spodní roztažitelnou stránku pomocí funkce *initBottomSheetLayout*. Dále jsme implementovali povinnou funkci *configureMap*. Touto funkcí řekneme knihovně Mapigate, s jakými parametry má inicializovat mapu. V našem případě jsou to souřadnice obklopující Zoo Praha a parametry podkladové mapy. Jak bylo zmíněno v dokumentaci navigačního modulu, podklad mapy je nutné poskytnout ve formě rozkrájených obrázků ve složkách podle přiblížení tzv. pyramida obrázků. Zhotovení tohoto rozčlenění by manuálně bylo velmi obtížné, proto jsme použili nástroj pro zpracování obrazu – libvips. Libvips nám umožňuje udělat pyramidu z velkého obrazu pomocí funkce *dzsave* s parametry cestou k souboru obrázku, cílovou složkou pro uložení a dalšími volitelnými parametry (Libvips b.r.) viz Kód 34 níže. Libvips spustíme mimo náš projekt v terminálu.

Kód 34 – Bash

```
$ vips dzsave map.png output --layout=google --tile-size=256 --suffix=.jpg
```

Zdroj: (Libvips b.r.)

Výstup z tohoto programu přesuneme do projektové složky *'/assets/tiles/'*, kde k němu bude mít přístup třída knihovna Mapigate.

Podle návrhu jsme do spodní stránky modulu zakomponovali záložky pomocí Android prvku *TabLayout*. Aby záložky fungovaly správně, potřebují tzv. adaptér, který udržuje stav záložek a zprostředkovává přepínání mezi záložkami. Implementace adaptéru musí být vždy vlastní, v našem případě jsme vytvořili třídu *AnimalPagerAdapter*, za jejíž pomoci inicializujeme obsah záložek (fragmenty) a předáváme mezi nimi zprávy.

Naplnění mapy

Abychom simulovali reálné použití mapy, vytvořili jsme lokální server, který zprostředkovává data pro vykreslení bodů a cesty mezi nimi. Protože jsme dosud programovali v programovacího jazyku Kotlin, nebude lokální server výjimkou. Server bude distribuovat REST API pro zoologickou zahradu. Byl vytvořen za pomocí populárního frameworku Spring Boot verze 2.5.0. Tento server není přímý cíl této práce, tudíž je probrán jen okrajově.

Pro splnění nároků na backend API jsme vytvořili dohromady šest endpointů. Všechny z nich jsou přístupné pomocí HTTP metody GET (nepotřebujeme posílat data z aplikace na server) a každý z nich zprostředkovává data ve formě JSON:

- `/api/zoo/animal` – Vrátí pole všech zvířat nacházejících se v zoo. Data jsou pouze v jednoduché podobě bez detailu pro urychlení komunikace viz Kód 35.

Kód 35 – JSON

```
[
  {
    "id": 36,
    "name": "Slon indický",
    "resId": "elephant",
    "latitude": 50.1192456,
    "longitude": 14.4055908,
```

```
    "smallDesc": "Je o něco menší než slon africký...",
    "accessPoint": {
      "lat": 50.1189778,
      "lng": 14.4057744
    }
  }
]
```

Zdroj: vlastní zpracování

- `/api/zoo/animal/{id}` – Najde a vrátí detail zvířete viz Kód 36 podle jeho identifikátoru *id*. Odpověď obsahuje vědeckou klasifikaci, potravu, reprodukci i jak se o zvíře Zoo Praha stará.

Kód 36 – JSON

```
{
  "animalClass": "Savci (Mammalia)",
  "order": "Chobotnatci (Proboscidea)",
  "range": "Asie (jižní a jihovýchodní Asie)",
  "biotop": "travnaté území, tropický les",
  "food": "části rostlin (listy, větve, kůra, plody)",
  "dimensions": "délka těla 5,5–6,5 m, hmotnost 3–5 t",
  "reproduction": "březost 18–22 měsíce, počet mláďat 1",
  "imageUrl": "...",
  "zooCare": "Historie chovu slonů v pražské zoo sahá do roku 1933, ..."
}
```

Zdroj: vlastní zpracování

- `/api/zoo/animal/dijkstra` – Vrátí pole informací nutných pro konstrukci grafu, pomocí kterého se realizuje výpočet Dijkstrova algoritmu. Každý objekt v poli reprezentuje oceněnou hranu spojující dva uzly. Tento objekt navíc nese informaci o vykreslení reálné cesty (*pathList*) viz Kód 37.

Kód 37 – JSON

```
[
  {
    "fromId": 28,
    "toId": 30,
    "distanceInMeters": 40,
    "pathList": [
      {
        "lat": 50.1179336,
        "lng": 14.4095336
      }
    ]
  }
]
```

Zdroj: vlastní zpracování

- `/api/zoo/animal/{id}/curiosity` – Vrátí pole zajímavostí o konkrétním zvířeti podle jeho identifikátoru *id* viz Kód 38.

Kód 38 – JSON

```
[
  {
    "id": 1,
    "value": "Sloni chováni v lidské péči se mohou dožít až 70 let.",
    "position": 5
  }
]
```

Zdroj: vlastní zpracování

- `/api/zoo/news?pNo={pNo}&pS={pSize}` – Vrátí zoo novinky (viz Kód 39) podle informací pro stránkování pomocí parametrů – číslo stránky *pNo* a počet prvků na stránku *pSize*. Takto navržený endpoint nám umožní načítat novinky postuně podle uživatelova zájmu tzv. lazy load.

Kód 39 – JSON

```
[
  {
    "id": 1,
    "title": "Souboj s časem v Údolí slonů",
    "imageUrl": "...",
    "createdDate": "06.02. 2021",
    "author": "..."
  }
]
```

Zdroj: vlastní zpracování

- `/api/zoo/news/{id}` – Vrátí detail zprávy se všemi dynamickými částmi novinky (*parts*) například titulek, text nebo obrázek podle identifikátoru *id*, viz Kód 40.

Kód 40 – JSON

```
{
  "id": 1,
  "title": "Souboj s časem v Údolí slonů",
  "imageUrl": "...",
  "createdDate": "06.02. 2021",
  "author": "...",
  "parts": [
    {
      "id": 1,
      "type": "TEXT",
      "value": "Souboj s časem v Údolí slonů...",
      "position": 1
    }
  ]
}
```

Zdroj: vlastní zpracování

Tento návrh endpointů byl následně interpretován ve Spring frameworku pomocí programovacího jazyku Kotlin. Výsledkem je tzv. RESTful server. Server nevyžaduje žádný autorizační proces. Ukázky JSON struktury výše jsme přetvořili na Kotlin data třídy, níže uvedené jako návratové hodnoty.

Kód 41 – Kotlin (Spring)

```
@GetMapping("/zoo/animal")
fun getAllAnimals(): List<AnimalDto> {
  return animalService.readAllAnimals()
}

@GetMapping("/zoo/animal/{id}")
fun getAnimalDetailById(@PathVariable id: Int): AnimalDetailDto {
  return animalService.readAnimalDetailById(id)
}

@GetMapping("/zoo/animal/dijkstra")
fun getDijkstraGraphInfo(): List<DijkstraInfoDto> {
  return animalService.readAllDijkstraInfo()
}

@GetMapping("/zoo/animal/{id}/curiosity")
fun getCuriositiesByAnimalId(@PathVariable id: Int)
    : List<AnimalCuriosityDto> {
  return animalService.readAllAnimalCuriosityById(id)
}

@GetMapping("/zoo/news")
fun getNewsPageable(@RequestParam pNo: Int, @RequestParam pS: Int)
    : List<NewsDto> {
  return newsService.readNewsPageable(pNo, pS)
}

@GetMapping("/zoo/news/{id}")
fun getNewsDetailById(@PathVariable id: Int): NewsDto {
  return newsService.readNewsDetailById(id)
}
```

Zdroj: vlastní zpracování

V samotné Android aplikaci komunikujeme se serverem pomocí těchto vytvořených endpointů, abychom získali data pro uživatele. Známá Android knihovna pro vykonání HTTP požadavků je Retrofit, kterou v naší aplikaci použijeme s kombinací Android knihovny Hilt.

Hilt je knihovna pro jednoduchou realizaci tzv. dependency injection (DI) pro Android. DI je programovací technika, díky níž je třída nezávislá na jejích závislostech (objektech) – instance třídy si závislosti získá sama. Implementace této techniky nám poskytuje výhody opakované použitelnosti kódu a snadnost refactoringu (Google 2020c). Hilt je nadstavba knihovny Dagger – Dagger kód generuje a Hilt tento generovaný kód optimalizuje pro Android aplikace.

Pro zapojení knihovny Hilt do našeho projektu musíme přidat Gradle závislost, přidat anotaci *@HiltAndroidApp* do naší aplikační třídy a označit aktivity anotací *@AndroidEntryPoint*.

S takto připraveným projektem můžeme vytvářet jednotlivé injektované objekty a Hilt moduly pro jejich injekci (Google 2021). Je v našem zájmu, aby injektovali jen potřebné objekty obvykle používané na více místech, například klient Retrofit knihovny nebo API service.

Kód 42 – Kotlin

```kotlin
interface AnimalApi {
  @GET("zoo/animal")
  suspend fun getAllAnimals(): Response<List<AnimalDto>>

  @GET("zoo/animal/{id}")
  suspend fun getAnimalById(@Path("id") id: Int): Response<AnimalDto>

  @GET("zoo/animal/dijkstra")
  suspend fun getAllDijkstraInfo(): Response<List<AnimalDijkstraDto>>

  @GET("zoo/animal/{id}/curiosity")
  suspend fun getAllCuriosityByAnimalId(@Path("id") id: Int):
    Response<List<AnimalCuriosityDto>>
}
```

Zdroj: vlastní zpracování

Takto přichystané rozhraní zvládne Retrofit knihovna interpretovat pro příjemnější dotazování serveru. Každá funkce je označena klíčovým slovem *suspend*, které jazyku Kotlinu říká, že ji lze volat jen z coroutin bloku nebo další funkce označené tímto slovem. Návratová hodnota je vždy typu *Response*, což je Retrofit třída reprezentující odpověď serveru, tím dokážeme reagovat na různé HTTP stavy a chybové hlášky. Pro serializaci JSON objektů do Kotlin objektů používáme Jackson knihovnu. Rozhraní *AnimalApi* lze následně injektovat do dalších tříd.

Kód 43 – Kotlin

```kotlin
class AnimalService @Inject constructor() {
  @Inject
  lateinit var animalApi: AnimalApi

  suspend fun findAllAnimals(): Response<List<AnimalDto>> {
    return animalApi.getAllAnimals()
  }

  suspend fun findAllDijkstra(): Response<List<AnimalDijkstraDto>> {
    return animalApi.getAllDijkstraInfo()
  }

  suspend fun findAllCuriosityByAnimalId(id: Int)
      : Response<List<AnimalCuriosityDto>> {
    return animalApi.getAllCuriosityByAnimalId(id)
  }
}
```

Zdroj: vlastní zpracování

Díky tomuto návrhu kódu lze jednoduše řetězit požadavky na server a celkově lépe pracovat s asynchronním programováním, v tomto případě s couroutinami, které jsme vysvětlili

v teoretické části této práce. Stejný přístup jsme zvolili při vytváření service API třídy pro zacházení s novinkami ze Zoo Praha (třída *NewsApi* a *NewsService*).

Dále jsme vytvořili modul *ApiModule*, který seskupuje funkcionalitu ohledně Retrofit knihovny. Poskytuje znovupoužitelného Retrofit klienta s adresou námi vytvořeného serveru a dvě rozhraní API podle Retrofit architektury (viz předchozí kód).

Kód 44 – Kotlin

```kotlin
@Module
@InstallIn(SingletonComponent::class)
object ApiModule {
  @Provides
  @Reusable
  internal fun provideAnimalApi(retrofit: Retrofit): AnimalApi {
    return retrofit.create(AnimalApi::class.java)
  }

  @Provides
  @Reusable
  internal fun provideNewsApi(retrofit: Retrofit): NewsApi {
    return retrofit.create(NewsApi::class.java)
  }

  @Provides
  @Reusable
  internal fun provideRetrofitInterface(): Retrofit {
    return Retrofit.Builder()
      .baseUrl(ZOO_API_BASE_URL) // ends with /api/
      .addConverterFactory(
        JacksonConverterFactory.create(jacksonObjectMapper())
      ).build()
  }
}
```

Zdroj: vlastní zpracování

Nakonec lze tento kód využít v námi implementovaném fragmentu *ZooMapFragment*, konkrétně ve funkci *fillMap*. V tělu této funkce jsme vytvořili blok coroutiny v globálním rozsahu. Jak jsme již vysvětlili v teoretické kapitole, coroutiny neblokují vlákna, což z nich dělá optimální nástroj pro asynchronní programování. Funkce našich service tříd jsou označeny slovem *suspend*, tudíž blok coroutiny čeká, než se dokončí jejich provedení. Kód 45 níže volá tyto dvě metody, kde druhá se spustí až po dokončení první bez blokování GUI vlákna.

Kód 45 – Kotlin

```
@Inject
lateinit var animalService: AnimalService

private fun fillMap() {
  GlobalScope.launch(Dispatchers.Main) {
    val animalsResponse = animalService.findAllAnimals()
    if (animalsResponse.isSuccessful) {
      populateMap(animalsResponse.body()!!.map { animalToMarker(it) })

      animalService.findAllDijkstra()
        .takeIf { it.isSuccessful }
        ?.let { initDijkstra(it.body()!!) } ?: showUnavailableToast()
    } else {
      showUnavailableToast()
    }
  }
}
```

Zdroj: vlastní zpracování

Současná poloha

Pro získání současné polohy využijeme Google služeb – Google Play polohové služby. Tato služba nám poskytne přístup k poslední poloze zařízení, tj. zeměpisné šířce, délce a azimutu zařízení. Ve většině případů je poslední poloha ekvivalentní současné poloze. U Android zařízení je tento způsob získání současné polohy doporučený od samotné platformy.

Samotného klienta poskytující poslední polohu je potřeba nastavit pro naše účely. Nejdříve je třeba vyžádat uživatelovo svolení o přístup k těmto službám. Náš návrh nevynucuje přístup k poloze na pozadí, které je dostupné až od Android úrovně API 29. Pro naše účely stačí, aby si aplikace zažádala o úrovně *ACCESS_COARSE_LOCATION* a *ACCESS_FINE_LOCATION* (přesnější určení polohy) – tyto dva přístupy musíme zahrnout do aplikačního manifestu (Google 2020m). Dále bylo nutné definovat standardní a nejrychlejší interval mezi přijetí současné polohy – proměnné *interval* a *fastestInterval*. Poté vytvoření samotného klienta a přiřazení ho k proměnné pro možnost reakce na cyklus fragmentu:

- při pozastavení (funkce *onPause*) čerpání polohy zastavíme,
- při obnovení (funkce *onResume*) zase začneme tyto služby požadovat.

Kód 46 – Kotlin

```
private val locationCallback = object : LocationCallback() {
  override fun onLocationResult(locationResult: LocationResult) {
    updateCurrentPosition(locationResult.lastLocation)
  }
}
...
val locationRequest = LocationRequest()
locationRequest.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
locationRequest.interval = 2000 // 2 seconds
locationRequest.fastestInterval = 1000 // 1 second
```

```
fusedLocationClient =
  LocationServices.getFusedLocationProviderClient(requireActivity())
fusedLocationClient.requestLocationUpdates(
  locationRequest, locationCallback, Looper.getMainLooper())
```

Zdroj: vlastní zpracování

Zapojení současné polohy do Mapigate modulu realizujeme vytvoření tzv. callbacku (proměnná *locationCallback* v kódu výše) viz kapitola Asynchronní programování v teoretické části. Tento callback nám umožní jednoduchou reakci na příjem signálu o poslední poloze – objekt typu *LocationResult*. V tomto případě potřebujeme aktualizovat bod polohy zařízení v mapě, což uděláme námi připravenou funkcí navigačního modulu *updateCurrentPosition*. Tímto docílíme vykreslení bodu současné polohy pro orientaci uživatele v mapě.

Grafická implementace stránek

Podle návrhu jsme dále implementovali grafické náhledy hlavně za pomoci fragmentů ale i jedné aktivity. Dohromady jsme vytvořili sedm grafických stránek aplikace, z toho jedna stránka je jen zobrazení informací o aplikaci (*AboutAppFragment*).

- **HomePageFragment**

Tento fragment je inicializován při každém otevření aplikace uživatelem. Vítá uživatele krátkým textem, nabídne tři nejnovější zprávy s jejich odkazem i odkaz na stránku s ostatními zprávami, a nakonec zobrazí text, jak může uživatel nepříspěvkové organizaci Zoo Praha pomoci. Z hlediska kódu je na této stránce nejdůležitější vytváření Intentů pro navigaci do detailu novinky (funkce *openNewsDetailActivity*) a otevření webové stránky v prohlížeči mimo naši aplikaci:

Kód 47 – Kotlin

```
private fun openNewsDetailActivity(newsId: Int) {
  val intent = Intent(context, NewsDetailActivity::class.java)
  intent.putExtra("zooNews", newsId)
  startActivity(intent)
}
...
howToHelpWebLink.setOnClickListener {
  val intent = Intent(Intent.ACTION_VIEW)
  intent.data = Uri.parse("https://www.zoopraha.cz/jak-pomoci")
  startActivity(intent)
}
```

Zdroj: vlastní zpracování

- **AnimalDetailFragment**

Nejdůležitější stránka pro uživatele je samotný detail zvířete. Zde se uživatel dozví základní informace o zvířeti, jako jeho základní popis, jak zvíře vypadá, vědeckou klasifikaci, potravu, reprodukci, nebo jak se o něj Zoo Praha stará. Detail zvířete je implementován jako fragment, který v sobě nese grafické prvky pro zobrazení těchto informací. Kód níže dotazuje backend server o detailní informace k danému zvířeti a následně jej vyplňuje do prvků. Níže je také použita knihovna Glide, která nám ulehčuje asynchronní načítání obrázků.

Kód 48 – Kotlin

```kotlin
fun loadFullDetail() {
  GlobalScope.launch(Dispatchers.Main) {
    val curiosityResponse = animalService.findAnimalDetailById(animal.id)
    if (curiosityResponse.isSuccessful) {
      loadingLayout.visibility = View.GONE
      animal = curiosityResponse.body()!!
      classText.text = animal.animalClass
      orderText.text = animal.order
      rangeText.text = animal.range
      biotopText.text = animal.biotop
      foodText.text = animal.food
      dimensionsText.text = animal.dimensions
      reproductionText.text = animal.reproduction
      zooCareText.text = animal.zooCare

      if (animal.imageUrl != null) {
        Glide.with(this@AnimalDetailFragment)
          .load(animal.imageUrl)
          .into(animalImage)
        animalImage.visibility = View.VISIBLE
      } else {
        animalImage.visibility = View.GONE // no image
      }

      loadingLayout.visibility = View.GONE
      detailInfoLayout.visibility = View.VISIBLE
      scrollView.scrollable = true
    } else {
      Toast.makeText(activity, UNAVAILABLE_MSG, Toast.LENGTH_LONG).show()
    }
  }
}
```

Zdroj: vlastní zpracování

- **AnimalCuriosityFragment**

Zobrazení zajímavostí o zvířeti je jednoduchá funkcionalita. Zahrnuje pouze seřazení a přetvoření listu objektů typu *AnimalCuriosityDto* na text, který se následně vyplní do hlavního textového prvku *TextView*. Pro konverzi z listu na text jsme vytvořili tzv. extension funkci. Kotlin touto funkcionalitou poskytuje možnost rozšířit třídu o nové funkce, aniž by bylo nutné třídu dědit, nebo používat návrhové vzory, jako je například dekoratér. Tyto funkce jsou k dispozici obvyklým způsobem, jako by šlo o metody původní třídy. Například je možné napsat nové funkce pro třídu z knihovny jiného výrobce, kterou normálně nelze upravit (Ebel 2019, 80). V našem případě extension funkce přidává funkcionalitu předělání listu textu na text v bodech.

V úryvku kódu níže nejprve zacházíme s odpovědí serveru a poté voláme extension funkci definovanou v úryvku Kód 49 níže.

Kód 49 – Kotlin

```
val curiosities = curiosityResponse.body()!!
curiosityText.text = if (curiosities.isEmpty()) {
  NO_DATA
} else {
  curiosities.toMutableList()
    .sortedBy { it.position }
    .map { it.value }
    .toBulletedList("\n\n")
}
...
private fun List<String>.toBulletedList(separator: String = "\n")
    : CharSequence {
  val lastIndex = this.size - 1
  return SpannableString(this.joinToString(separator)).apply {
    this@toBulletedList.foldIndexed(0) { i, acc, span ->
      val curr = if (i != lastIndex) separator.length else 0
      val end = acc + span.length + curr
      this.setSpan(BulletSpan(20), acc, end, 0)
      end
    }
  }
}
```

Zdroj: vlastní zpracování

- **AnimalDirectionFragment**

Pro zobrazení detailní cesty jsme připravili fragment, který interpretuje návratovou hodnotu Mapigate modulu po nalezení nejkratší cesty spuštěné funkcí *onShortestPathFind*. Cíl tohoto fragmentu je uživatelovi strukturovaně zobrazit detail cesty. Detail cesty znamená její celková vzdálenost, zvířata, které uživatel potká po cestě ke svému cíli, a vzdálenosti mezi těmito zastávkami. Funkce níže znázorňuje zaznamenání těchto informací.

Kód 50 – Kotlin

```
fun refreshValues(animal: AnimalDto, markerList: List<MapMarker>,
    distStepMap: Map<Any, Double>, distToFirst: Double,
    dijkstraDist: Double) {
  decorateDistanceHeader(animal, distToFirst)

  val lastIndex = markerList.size - 1
  for ((i, m) in markerList.withIndex()) {
    val animalInfoLayout = LinearLayout(context)
    animalInfoLayout.layoutParams = layoutParams
    val animalIcon = ImageView(context)
    m.markerIconResId?.let { animalIcon.setImageResource(it) }
    animalInfoLayout.addView(animalIcon)
    val animalNameText = TextView(context)
    animalNameText.text = m.title
    animalNameText.layoutParams = textLayoutParams
    animalInfoLayout.addView(animalNameText)
    if (i == lastIndex) {
      pathTargetLayout.addView(animalInfoLayout)
```

```
    } else {
      pathStepLayout.addView(animalInfoLayout)
      val distText = TextView(context)
      distText.layoutParams = textLayoutParams
      distText.text = DistanceUtil.prettifyMeters(
        distStepMap[m.markerId] ?: dijkstraDist)
      pathStepLayout.addView(distText)
    }
  }
}
```

Zdroj: vlastní zpracování

- **NewsOverviewFragment**

Poslední fragment má za úkol zobrazení seznamu zpráv v jednoduché formě. Tyto zprávy jsou čerpány z backend serveru pomocí jednoduchého stránkování neboli lazy load. V praxi jsme lazy load napojili na infinity scroll, kde vždy, když uživatel dosáhne konce stránky, tak mu zobrazíme další novinky v pořadí. Stránkování jsme použili pro limitaci velkého počtu zpráv, aniž by je uživatel viděl. Tímto způsobem zobrazíme jen nejnovější zprávy rychle při limitaci datového přenosu.

Kód 51 – Kotlin

```
newsRecyclerView.addOnScrollListener(
  object : RecyclerView.OnScrollListener() {
    override fun onScrollStateChanged(recyclerView: RecyclerView,
        newState: Int) {
      super.onScrollStateChanged(recyclerView, newState)
      if (hasMoreNews && !recyclerView.canScrollVertically(1)
          && newState == RecyclerView.SCROLL_STATE_IDLE) {
        fetchNews() // lazy load news
      }
    }
  })
...
private fun fetchNews() {
  progressCircle.visibility = View.VISIBLE
  GlobalScope.launch(Dispatchers.Main) {
    val response = zooService.findAllNewsSimpleByPage(
      pageNumber++, pageSize)

    if (response.isSuccessful) {
      val news = response.body()!!
      if (news.size < pageSize) {
        hasMoreNews = false
      }

      newsList.addAll(news)
      newsAdapter.notifyDataSetChanged()
      hasSomeNews = true
    } else {
      Toast.makeText(activity, UNAVAILABLE_MSG, Toast.LENGTH_LONG).show()
    }
```

```
    if (!hasSomeNews) {
      unavailableMsg.text = "Nejsou dostupné žádné novinky"
    }
    progressCircle.visibility = View.INVISIBLE
  }
}
```

Zdroj: vlastní zpracování

- **NewsDetailActivity**

Tento grafický náhled jsme implementovali jako aktivitu, což zní dělá jedinou aktivitu mimo tu hlavní. Nutnost aktivity vzešla z potřeby reakce na uživatelův pokyn jít zpět, typicky vyvolán tlačítkem zpět na zařízení. Když vytvoříme aktivitu z aktivity, nemusíme implementovat extra navigaci. Vytvořením nové aktivity také dává možnost aplikaci otevřít na konkrétní zoo novince. Název této třídy je *NewsDetailActivity* a rozšiřuje Android aktivitu.

Situaci nám komplikuje nutnost vytvoření dynamického obsahu – podle typu části zprávy; enum *NewsPartType*, který může nabývat tří stavů: *TITLE*, *TEXT* a *IMAGE*. Každý z těchto typů nese specifickou hodnotu, například enum *IMAGE* nese URL adresu obrázku. Podle těchto částí článku musíme programaticky vytvořit grafické prvky. Tato funkcionalita je znázorněna níže ve funkci *renderNewsParts*, která se volá z coroutiny v reakci na odpověď serveru.

Kód 52 – Kotlin

```
private fun renderNews(news: NewsDto) {
  newsHeaderText.text = news.title
  newsAuthorAndDateText.text = "${news.author} | ${news.createdDate}"

  for (np in news.parts!!) {
    when (np.type) {
      NewsPartType.TITLE, NewsPartType.TEXT -> {
        val textView = TextView(applicationContext)
        textView.text = np.value
        if (np.type == NewsPartType.TITLE) {
          textView.layoutParams = titleLayoutParams
          textView.setTextAppearance(R.style.H2)
        } else {
          textView.layoutParams = textLayoutParams
        }
        newsContent.addView(textView)
      }
      NewsPartType.IMAGE -> {
        val imageView = ImageView(applicationContext)
        Glide.with(this)
          .load(np.value)
          .into(imageView)
        newsContent.addView(imageView)
      }
    }
  }
}
```

Zdroj: vlastní zpracování

4.2.4 Testování

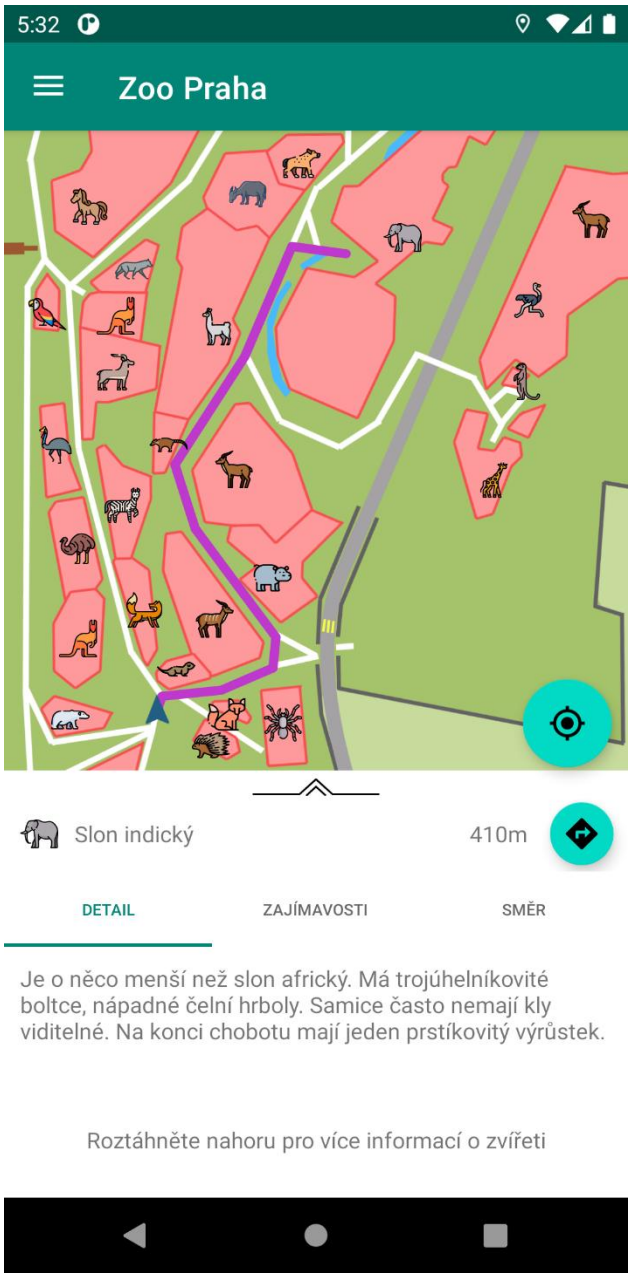
Aplikaci jsme otestovali manuálně. Testy proběhly ve formě white box testů při vývoji a posléze tzv. monkey testingem. Jako zařízení jsme použili emulovaný přístroj Google Pixel 3a (Android 11) a fyzické zařízení Redmi Note 4 (Android 7). Emulace Android zařízení, tzv. AVD (Android Virtual Device) je zabudovaná funkcionalita námi používaného IDE Android Studio. Tato zvolená zařízení nám poskytují dobrou variabilitu zařízení – Android 11 s API úrovní 30 a Android 7 s API úrovní 24-25, což je limit naší zoo aplikace.

Aplikace fungovala správně (bez problémů) na zařízeních zmíněných výše. Použitá metoda deep-zoom mapy se projevila jako velmi efektivní a poskytla hladkou orientaci v mapě. Velmi příjemná funkce je rotace mapy, kterou jsme využili hlavně pro orientaci v prostoru.

4.2.5 Výsledný vzhled aplikace

Níže jsou uvedeny dva nejdůležitější screenshoty – vzhled mapy a záložka s detailem o cestě. Zbylé screenshoty částí aplikace jsou uvedeny v příloze B. Screenshoty byly pořízeny na emulovaném zařízení Google Pixel 3a (Android 11) o úhlopříčce 5,6 palce.

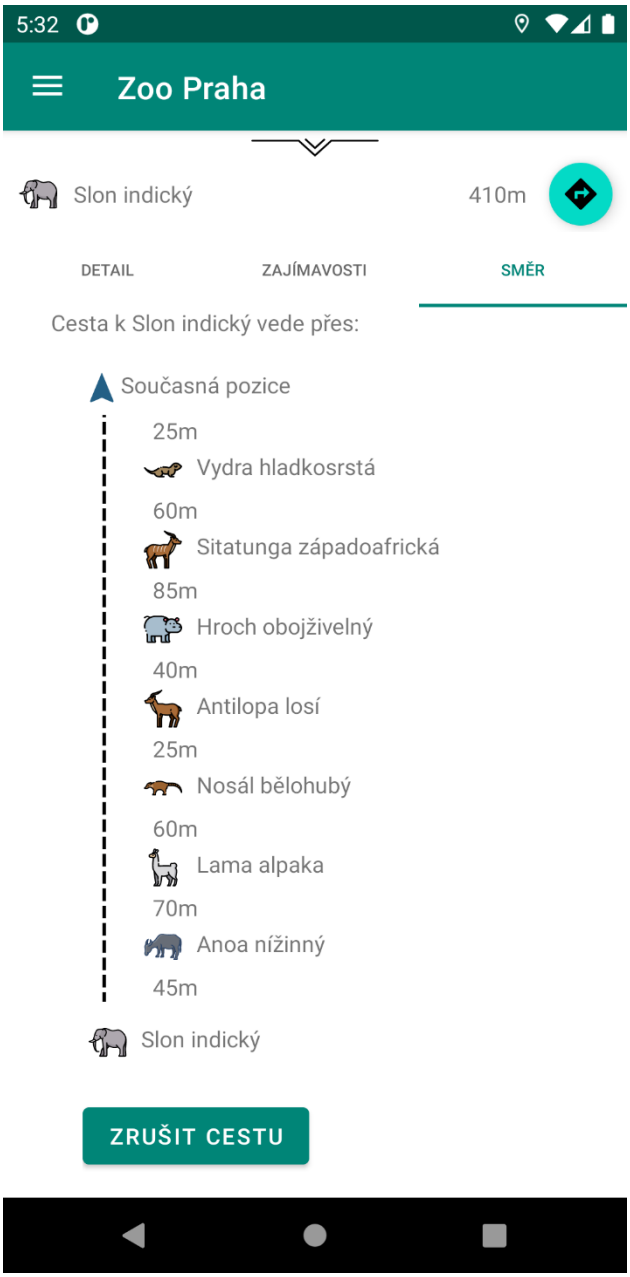
Obrázek 14: Výsledný vzhled aplikace – mapa a detail zvířete



Zdroj: vlastní zpracování

Předchozí obrázek zobrazuje aplikaci ve stavu, kde uživatel právě vyhledal nejkratší cestu ke zvířeti Slon indický. Celková vzdálenost mezi současnou polohou a cílovým zvířetem je 410 m. Cesta (fialová trasa) začíná v bodě současné polohy (modrá šipka) a pokračuje kolem ostatních zvířat až do cíle. Ve spodní stránce je zobrazen krátký popis zvířete. Při roztažení této stránky se o něm uživatel dozví další detaily.

Obrázek 15: Výsledný vzhled aplikace – záložka s detailem nejkratší cesty



Zdroj: vlastní zpracování

V obrázku výše je zobrazen detail cesty v plně rozvinuté spodní stránce. Odpovídá vyhledání z předchozího obrázku *Výsledný vzhled aplikace – mapa a detail zvířete*. Uživatel se zde dozví podrobné části trasy, a hlavně jaká zvířata po cestě k cíli navštíví. Ve spodní části obrázku se nachází tlačítko pro zrušení trasy v mapě i v této záložce.

5 Výsledky a diskuse

V této diplomové práci se nám povedlo úspěšně dokončit dva projekty – navigační modul (Android knihovna Mapigate) a mobilní aplikaci pro zoologickou zahradu v Praze se svou RESTful backend aplikací. Dohromady jsme v rámci této diplomové práce napsali 2300 řádků Kotlin kódu (bez prázdných řádků) a 940 řádků XML Android layoutu v grafických aplikací.

5.1 Navigační modul

Výsledkem je plně funkční Android knihovna, která umí zobrazovat mapu, a umí do ní zapisovat body i cestu mezi nimi. Knihovna je veřejně přístupná přes package repository JitPack. Její zapojení do projektu je velmi jednoduché – stačí zapojit hlavní fragment MapigateFragment a vývojář má přístup k výhodám knihovny. Dále lze jen naslouchat vytvořeným funkcím nebo pracovat s API knihovny.

Tento projekt je open-source, zdrojový kód je volně přístupný všem v GitHub repositáři viz příloha A. V takto vytvořeném repositáři mohou vývojáři lehce nahlašovat chyby, navrhovat zlepšení nebo použít kód jako základ ke svému projektu, tzv. fork.

Funkcionality modulu jsou v základní podobě a odpovídají první verzi nové knihovny. Určitě existuje vylepšení současného kódu, a hlavně nové funkcionality. Například by knihovna mohla nabídnout možnost výběru algoritmu nejkratší cesty vývojářem, nebo optimalizovat algoritmus výpočtu nejkratší cesty podle počtu uzlů v grafu. Nápadů je několik, ovšem je otázkou, které funkcionality patří do této knihovny. Tyto nápady mohou být implementovány v případě zájmu GitHub komunity.

Možnosti využití této knihovny je široké. I když jsme v praktické části doporučovali používat Mapigate na omezené, rozumně velké areály, není vyloučeno, že knihovnu nelze použít na celou zeměkouli – v tomto případě bude hrát hlavní roli velikost přiblížení. Doporučujeme Mapigate knihovnu používat na detailní mapy objektů všech velikostí, ovšem vždy je třeba dát pozor na potřebu přiblížení (světová mapa s největším detailem přiblížení nemusí mít správnou rychlost zobrazení). Využití Mapigate může být například pro zobrazení zajímavostí ve městě, turistických tras v horách nebo i jednoduchá mapa v interiéru (muzeum, výstavy umění).

Celkový čas strávený na tomto projektu je cca půl roku, kde velkou část zabral návrh knihovny – hlavně návrh mapy a použitelnost knihovny. V následující tabulce jsou uvedeny vybrané metriky kódu této knihovny.

Tabulka 3: Metriky kódu knihovny Mapigate

Název metriky	Hodnota
Počet řádků v jazyce Kotlin	cca 630 (80 %)
Počet řádků v jazyce XML	cca 150 (20 %)
Počet Kotlin souborů (.kt)	15
Počet XML souborů (.xml)	10

Zdroj: vlastní zpracování

5.2 Aplikace pro Zoo Praha

Účel této aplikace byl demonstrovat použití navigačního modulu (knihovna Mapigate), což se nám podařilo splnit. Aplikace umí zobrazit novinky ze zoo, a hlavně interaktivní digitalizovanou mapu se zvířaty. Mezi další možnosti funkcionality patří upozornění uživatele na současné krmení zvířat a podobné upozornění na akce.

Původně měla tato aplikace reálně sloužit jako digitální mapa pro Zoo Praha, ale kvůli komplikacím s otevřenými daty Zoo Praha (viz kapitola 4.2.1 Analýza) jsme museli vytvořit vlastní API server. I kdyby byli tyto otevřená data přístupná pro naše použití v aplikaci, stejně bychom museli implementovat RESTful API server pro zaznačení bodů do mapy a vyhledávání nejkratší cesty v mapě. Kromě datové komplikace jsme nenarazili na jiné nepříjemnosti při vývoji této aplikace.

Vývoj Android aplikace a RESTful API serveru pro Zoo Praha trvalo zhruba pět měsíců, což ale nepředstavuje čistý čas strávený nad vývojem (čistý čas se špatně hodnotí v retrospektivě). V tabulkách 4 a 5 jsou uvedeny vybrané metriky kódu těchto projektů. Metrika počet řádků nezahrnuje prázdné řádky v kódu.

Tabulka 4: Metriky kódu Android Aplikace pro Zoo Praha

Název metriky	Hodnota
Počet řádků v jazyce Kotlin	cca 1220 (60 %)
Počet řádků kódu v jazyce XML	cca 790 (40 %)
Počet Kotlin souborů (.kt)	27
Počet XML souborů (.xml)	20

Zdroj: vlastní zpracování

Tabulka 5: Metriky kódu API server pro Zoo Praha

Název metriky	Hodnota
Počet řádků v jazyce Kotlin	cca 450
Počet Kotlin souborů (.kt)	21

Zdroj: vlastní zpracování

6 Závěr

Cílem práce bylo vytvořit modul pro mobilní aplikaci zaměřený na platformu Android OS. Pro vývoj byl použit programovací jazyk Kotlin, který byl popsán a porovnán s jazykem Java v teoretické části této práce. Modul je navrhnut pro použití jako základ mobilních aplikací, které slouží jako průvodce v prostoru pomocí polohového zaměření v reálném čase. Vytvořený modul byl následně implementován do navigační aplikace pro zoologickou zahradu.

V teoretické části jsme čtenářovi představili operační systém Android, možnosti vývoje a architekturu této platformy. Následně jsme popsali programovací jazyk Kotlin, jeho využití i technologie, které se s tímto jazykem pojí. Poté jsme jej porovnali s programovacím jazykem Java za pomocí ukázek kódu obou jazyků. Rešerši jsme zakončili stručným popsáním webových technologií API, REST, JSON, pojmu IoT a přiblížením nástroje WebML.

V praktické části jsme se zabývali samotnými částmi vývoje obou softwarů. První část je věnována navigačnímu modulu neboli Android knihovně nazvané Mapigate. Zde jsme důkladně probrali technický návrh mapy i algoritmus vyhledání nejkratší cesty a následně jsme tento návrh implementovali pomocí jazyku Kotlin. Důraz jsme kladli na použitelnost, kterou jsme podrobně vysvětlili. Ve druhé části jsme se zaměřili na aplikování Mapigate knihovny v Android aplikaci pro zoologickou zahradu Zoo Praha. Aplikaci jsme funkčně i graficky navrhnuli podle WebML modelů a následně zdokumentovali programovou implementaci Android komponent. Pro implementaci jsme vytvořili lokální RESTful server, který zprostředkovával data pro mobilní aplikaci. Poté jsme vysvětlili funkcionalitu zobrazení současné polohy v mapě. Praktickou část jsme zakončili obrázky výsledného vzhledu aplikace pro zoo.

Nakonec jsme uvedli výsledky a vedli krátkou diskusi na toto téma. Zhodnotili jsme navigační modul (knihovnu Mapigate), jak se může knihovna v budoucnosti vyvíjet a uvedli jsme možnosti nových funkcionalit. Poté jsme uvedli jeho využití, poskytli jsme doporučení pro implementaci a uvedli vybrané metriky kódu. Dále jsme uvedli výsledky mobilní aplikace pro příspěvkovou organizaci Zoo Praha, a i zde jsme uvedli vybrané metriky kódu aplikace.

7 Seznam použitých zdrojů

Abrahão, Silvia, Lucia De Marco, Filomena Ferrucci, Jaime Gomez, Carmine Gravino, a Federica Sarro. 2018. „Definition and Evaluation of a COSMIC Measurement Procedure for Sizing Web Applications in a Model-Driven Development Environment". *Information and Software Technology* 104 (prosinec): 144–61. https://doi.org/10.1016/j.infsof.2018.07.012.

AbuSalim, Samah W.G., Rosziati Ibrahim, Mohd Zainuri Saringat, Sapiee Jamel, a Jahari Abdul Wahab. 2020. „Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization". *IOP Conference Series: Materials Science and Engineering* 917 (září): 012077. https://doi.org/10.1088/1757-899X/917/1/012077.

Agarwal, Sandeep. 2019. „Pros and Cons of Kotlin for Android App Development". Medium. 22. říjen 2019. https://medium.com/quick-code/pros-and-cons-of-kotlin-for-android-app-development-c4b0f95c1324.

AltexSoft. 2019. „What Is API: Definition, Types, Specifications, Documentation". *AltexSoft* (blog). 18. červen 2019. https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/.

Ardito, Luca, Riccardo Coppola, Giovanni Malnati, a Marco Torchiano. 2020. „Effectiveness of Kotlin vs. Java in Android App Development Tasks". *Information and Software Technology* 127 (listopad): 106374. https://doi.org/10.1016/j.infsof.2020.106374.

Bose, Subham, Madhurima Banerjee, Madhuleena Mukherjee, a Aditi Kundu. 2018. „A COMPARATIVE STUDY: JAVA VS KOTLIN PROGRAMMING IN ANDROID APPLICATION DEVELOPMENT". *International Journal of Advanced Research in Computer Science* 9 (3): 41–45. https://doi.org/10.26483/ijarcs.v9i3.5978.

Ceri, Stefano, Piero Fraternali, a Aldo Bongio. 2000. „Web Modeling Language (WebML): A Modeling Language for Designing Web Sites". *Computer Networks* 33 (1–6): 137–57. https://doi.org/10.1016/S1389-1286(00)00040-2.

Dunn, Mike. 2019. *moagrius/TileView*. Java. https://github.com/moagrius/TileView.

Ebel, Nate. 2019. *Mastering Kotlin: Learn Advanced Kotlin Programming Techniques to Build Apps for Android, IOS, and the Web*. Birmingham: Packt Publishing, Limited. https://www.overdrive.com/search?q=E3FE049D-CDD4-49AB-AE79-37FEF1187AD5.

Facebook. b.r. „React Native · A Framework for Building Native Apps Using React · React Native". Viděno 23. leden 2021. https://reactnative.dev/.

Fielding, Roy Thomas. 2000. „Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation, Irvine: University of California. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Google. 2020a. „Location". Android Developers. 19. duben 2020. https://developer.android.com/reference/android/location/Location.

Google. 2020b. „Meet Android Studio". Android Developers. 16. květen 2020. https://developer.android.com/studio/intro.

Google. 2020c. „Dependency Injection in Android". Android Developers. 10. červen 2020. https://developer.android.com/training/dependency-injection.

Google. 2020d. „Android's Kotlin-First Approach". Android Developers. 20. červenec 2020. https://developer.android.com/kotlin/first.

Google. 2020e. „Fragments". Android Developers. 27. červenec 2020. https://developer.android.com/guide/fragments.

Google. 2020f. „<uses-Sdk>". Android Developers. 30. červenec 2020. https://developer.android.com/guide/topics/manifest/uses-sdk-element.

Google. 2020g. „Activity | Android Developers". 23. srpen 2020. https://developer.android.com/reference/android/app/Activity.

Google. 2020h. „Platform Architecture". Android Developers. 2. září 2020. https://developer.android.com/guide/platform.

Google. 2020i. „Device Compatibility Overview". Android Developers. 4. září 2020. https://developer.android.com/guide/practices/compatibility.

Google. 2020j. „Android 11". Android. 8. září 2020. https://www.android.com/android-11/.

Google. 2020k. „Codenames, Tags, and Build Numbers". Android Open Source Project. 23. listopad 2020. https://source.android.com/setup/start/build-numbers.

Google. 2020l. „Kotlin Coroutines on Android". Android Developers. 24. listopad 2020. https://developer.android.com/kotlin/coroutines.

Google. 2020m. „Request Location Permissions". Android Developers. 25. listopad 2020. https://developer.android.com/training/location/permissions.

Google. 2020n. *flutter/flutter*. Dart. Flutter. https://github.com/flutter/flutter.

Google. 2021. „Dependency injection with Hilt | Android Developers". 19. leden 2021. https://developer.android.com/training/dependency-injection/hilt-android.

Google. b.r. „Custom Maps | Google Maps Platform". Google Cloud. Viděno 6. duben 2020. https://cloud.google.com/maps-platform/maps.

Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic, a Marimuthu Palaniswami. 2013. „Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions". *Future Generation Computer Systems* 29 (7): 1645–60. https://doi.org/10.1016/j.future.2013.01.010.

Haverbeke, Marijn. 2019. *Eloquent JavaScript: a modern introduction to programming*. Third edition. San Francisco: No Starch Press.

Hunter, Scott. 2020. „Introducing .NET Multi-platform App UI | .NET Blog". 19. květen 2020. https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/.

ChooseALicense.com. 2020. „Apache License 2.0". Choose a License. 30. říjen 2020. https://choosealicense.com/licenses/apache-2.0/.

JetBrains. 2020a. „Kotlin Docs | Kotlin". Kotlin Help. 16. srpen 2020. https://kotlinlang.org/docs/reference.

JetBrains. 2020b. „Kotlin Native | Kotlin". Kotlin Help. 16. srpen 2020. https://kotlinlang.org/docs/reference/native-overview.html.

JetBrains. 2020c. „Data Classes | Kotlin". Kotlin Help. 25. srpen 2020. https://kotlinlang.org/docs/data-classes.html.

JetBrains. 2020d. „Asynchronous Programming Techniques | Kotlin". Kotlin Help. 7. prosinec 2020. https://kotlinlang.org/docs/async-programming.html.

„JSON". b.r. Viděno 17. leden 2021. https://www.json.org/json-cz.html.

Laurence, Pierre. 2020. *GitHub - peterLaurence/MapView: A Fast, memory efficient Android library to display tiled maps, with support for markers, paths, and rotation.* https://github.com/peterLaurence/MapView.

Libvips. b.r. „libvips". Viděno 2. květen 2020. https://libvips.github.io/libvips/API/current/Making-image-pyramids.md.html.

Looper, Christian de, a Martin Daniel. 2021. „A Brief History of Google's Android, 12 Years Since Its Inception". Digital Trends. 3. únor 2021. https://www.digitaltrends.com/mobile/android-version-history/.

Magistrát hlavního města Prahy. b.r. „O nás - Opendata Praha". Viděno 17. říjen 2020. https://opendata.praha.eu/about.

Meier, Reto, a Ian Lake. 2018. *Professional Android*. Fourth edition. Indianapolis, Indiana: John Wiley & Sons.

Merry, Krista, a Pete Bettinger. 2019. „Smartphone GPS Accuracy Study in an Urban Environment". Editoval Filip Biljecki. *PLOS ONE* 14 (7): e0219890. https://doi.org/10.1371/journal.pone.0219890.

Microsoft. b.r. „What Is Xamarin? | .NET". Microsoft. Viděno 23. leden 2021. https://dotnet.microsoft.com/learn/xamarin/what-is-xamarin.

Mozilla Contributors. 2020. „API - MDN Web Docs Glossary: Definitions of Web-related terms | MDN". 15. prosinec 2020. https://developer.mozilla.org/en-US/docs/Glossary/API.

Oracle. 2019a. „Java Language Specification, Chapter 1. Introduction". 17. září 2019. https://docs.oracle.com/javase/specs/jls/se14/html/jls-1.html.

Oracle. 2019b. „Java Virtual Machine Specification, Chapter 1. Introduction". 17. září 2019. https://docs.oracle.com/javase/specs/jvms/se14/html/jvms-1.html.

Sims, Gary. 2019. „I Want to Develop Android Apps — What Languages Should I Learn?" Android Authority. 10. srpen 2019. https://www.androidauthority.com/develop-android-apps-languages-learn-391008/.

StatCounter. 2021. „Mobile Android Version Market Share Worldwide". StatCounter Global Stats. 4. leden 2021. https://gs.statcounter.com/android-version-market-share/mobile/worldwide/2020.

Wang, Xiao Zhu. 2018. „The Comparison of Three Algorithms in Shortest Path Issue". *Journal of Physics: Conference Series* 1087 (září): 022011. https://doi.org/10.1088/1742-6596/1087/2/022011.

Zoo Praha. 2020. „Zoo v číslech". Zoo Praha. 31. prosinec 2020. https://www.zoopraha.cz/zvirata-a-expozice/zvirata-v-cislech.

8 Přílohy

8.1 Příloha A: Odkaz na online repositář Android knihovny Mapigate

Online repositář vedený v nejrozšířenější webové službě podporující vývoj softwaru za pomoci verzovacího nástroje Git – GitHub. Tento repositář obsahuje veškerý kód Android knihovny Mapigate a soubor README.md, který přibližuje význam repositáře a aplikace. Odkaz na repositář je trvalý.

Odkaz: https://github.com/unbearables/mapigate

8.2 Příloha B: CD se zdrojovým kódem aplikací pro Zoo Praha

CD obsahuje zdrojový kód Android mobilní aplikace pro Zoo Praha, API backend aplikace a další screenshoty konečného vzhledu Android aplikace pro Zoo Praha.