FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Mobilní lexikon zvířat ZOO Praha
Student:	Bc. Ondřej Košut
Vedoucí:	Ing. Josef Gattermayer, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Mobilní lexikon zvířat ZOO Praha
Pražská ZOO uveřejnila v rámci portálu opendata.praha.eu množství zajímavých informací o zde žijící zvěři –http://opendata.praha.eu/dataset/zoo-lexikon-zvirat. Cílem práce je vytvořit mobilní aplikaci pro telefony a tablety, která tato data atraktivní formou přiblíží návštěvníkům a to včetně serveru, který bude data pravidelně aktualizovat.

1. Navrhněte vhodnou funkcionalitu pro mobilní aplikaci na základě dostupných dat.

2. Konzultujte s vedoucím práce grafické pojetí aplikace.

3. Navrhněte a implementujte server v Node.js, který bude data stahovat z portálu opendata.praha.eu a přes REST API nabízet mobilní aplikaci.

4. Navrhněte, implementujte a otestujte mobilní aplikaci pro Android.

5. Aplikaci publikujte veřejně na Google Play.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 4. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Mobilní lexikon zvířat ZOO Praha

Bc. Ondřej Košut

katedra softwarového inženýrství
Vedoucí práce: Ing. Josef Gattermayer, Ph.D.

8. ledna 2020

// Poděkování

Děkuji vedoucímu práce za vedení a výborné návrhy, děkuji mé přítelkyni a mé rodině za pomoc a za jejich trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 8. ledna 2020 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Ondřej Košut. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Košut, Ondřej. *Mobilní lexikon zvířat ZOO Praha.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Práce popisuje tvorbu doprovodné aplikace ZOO Praha pro mobilní zařízení se systémem Android, za využití volně dostupných dat z portálu opendata.praha.eu. Práce se zabývá celým procesem vývoje aplikace. Rozebírá dostupná data a navrhuje na jejich základě požadavky na aplikaci. Navrhuje, jak mají být data uložena v databázi a popisuje nasazení databáze do cloudového systému. Popisuje tvorbu, nasazení a testování serveru, který data aktualizuje a poskytuje koncové aplikaci. Rozebírá návrh uživatelského rozhraní, implementaci aplikace i její testování – jak uživatelské, tak automatické. Popisuje také proces publikace aplikace do obchodu Google Play. Práce takto dává dohromady ucelený popis procesu vzniku aplikace od jejího návrhu až po její zveřejnění.

Klíčová slova ZOO Praha, open data, databáze, server, Android, Google Play, MongoDB, Kotlin, NodeJS, lexikon

Abstract

This thesis describes the process of creating an application for the Prague ZOO for Android devices, with the usage of freely available data from the website opendata.praha.eu. The thesis goes through the whole process of creating the application. Thesis analyses the available data and describes requirements for

the application based on this analysis. Thesis describes how the data needs to be stored in database and describes the process of deploying this database to a cloud. Then, the thesis describes the process of creating, deploying, and testing of a server, that updates the database and provides this data to the application. User interface design of the Android application is created, implementation is described, and testing is described – both user testing and automatic testing. Thesis describes the process of publishing the application to the Google Play store. In this way, this thesis puts together a full description of the process of creating an application from the initial analysis all the way through to the publishing.

Keywords Prague ZOO, open data, database, server, Android, Google Play, MongoDB, Kotlin, NodeJS, lexicon

Obsah

Úvod **1**

1 Návrh funkcionalit mobilní aplikace **3**
1.1 Dostupná data 3
1.2 Existující aplikace pro Android 5
1.3 Navržené funkcionality 6

2 Implementace serveru **9**
2.1 Databáze 9
2.2 Node.js server 10
2.3 Nasazení 15

3 Implementace mobilní aplikace **21**
3.1 Návrh 21
3.2 Implementace 31
3.3 Testování 40
3.4 Publikace 51

Závěr **59**

Literatura **61**

A Seznam použitých zkratek **65**

B Obsah přiloženého CD **67**

C Snímky obrazovky aplikace **69**

Seznam obrázků

2.1 Dokumentaci API na swaggerhub.com 19
2.2 Výsledky testování v aplikaci *Postman* 20

3.1 Návrh obrazovky seznamu zvířat (vlevo) a seznamu pavilonů (vpravo) 23
3.2 Návrh obrazovky seznamu akcí (vlevo) a navigačního menu (vpravo) 25
3.3 Návrh obrazovky detailu zvířete. Obecné informace (vlevo), rozšíření (veprostřed) a zajímavosti (vpravo) 26
3.4 Graf přechodů obrazovek 27
3.5 Architektura a nasazení 30
3.6 Vrstvy v aplikaci 35
3.7 Vyhledávání opravené dle výsledků uživatelského testování 52
3.8 Stránka v obchodu Google Play. Ikona od uživatele Freepik z www.flaticon.com 58

C.1 Seznam zvířat (vlevo) a Obecné informace o zvířeti (vpravo) . . . 69
C.2 Rozšíření zvířete (vlevo) a Zajímavosti o zvířeti (vpravo) 70
C.3 Seznam pavilonů (vlevo) a Dialog filtrování zvířat (vpravo) 70
C.4 Vyhledávání se zapnutými filtry (vlevo) a Nadcházející akce (vpravo) 71

Úvod

Zoologickou zahradu hl. m. Prahy v Troji (dále jen “ZOO Praha”) ročně navštíví přibližně 1,5 milionu návštěvníků. [1] V roce 2015 byla zařazena serverem TripAdvisor na čtvrté místo v seznamu nejlepších zoologických zahrad na světě. [2] V zahradě je přes 150 expozic a téměř 700 druhů zvířat. [3] Může být tedy až podivuhodné, že zoologická zahrada nemá v dnešní době oficiální doprovodnou aplikaci pro mobilní zařízení, přitom chytré telefony nyní vlastní již většina české populace. [4]

Naštěstí však ZOO Praha (na základě programového záměru hl. m. Prahy) začala poskytovat svá data k veřejnému užití. Ta jsou však pro normálního uživatele nepřehledná a obtížně přístupná, tudíž je nelze využít k dohledání informací přímo při návštěvě ZOO. Nabízí se tedy možnost vytvořit vlastní doprovodnou aplikaci pro mobilní zařízení, která by poskytovala uživatelům možnost těchto dat využít v přehledné a atraktivní formě. A právě vytvoření takové aplikace je hlavním cílem této práce.

V první kapitole se práce zabývá návrhem funkcionalit mobilní aplikace. Za tímto účelem jsou nejdřív analyzována dostupná data, s cílem zjistit jaká data jsou nabízena a k čemu je lze v aplikaci využít. Jaké mají data formáty, jaká data jsou aktuální a podobně. Kromě analýzy dostupných dat je provedena i rešerše aktuálně existujících aplikací souvisejících se ZOO Praha a dostupných pro platformu Android. V závěru této kapitoly jsou definovány navržené funkční a nefunkční požadavky.

V další kapitole je popsána implementace serveru. Tento server stahuje data, ukládá je do databáze a provádí nad daty různé úpravy. Data poté nabízí přes internet mobilní aplikaci. Kapitola je rozdělena do tří částí. V první je popsán výběr databáze, kam mají být data uložena. Ve druhé části je popsána tvorba serveru. Jaký jazyk je využit, jaké knihovny, popis implementace s příklady zdrojového kódu. Ve třetí části je popsáno nasazení. Jedná se o nasazení databáze a serveru na cloudovou platformu (včetně výběru konkrétní platformy), o dokumentaci rozhraní a o popis, jak bylo provedeno testování

serveru.

Třetí a zároveň poslední kapitola se zabývá přímo aplikací pro Android. Nejprve popisuje provedení návrhu uživatelského rozhraní, na základě navržených funkčních požadavků z první kapitoly. To zahrnuje detailní popis jednotlivých obrazovek a popis grafického pojetí aplikace. Druhou částí návrhu je návrh architektury aplikace a volba programovacího jazyka. Dále kapitola popisuje implementaci aplikace, postupně od řešení napojení aplikace na server (popsaný ve druhé kapitole), až po popis implementace uživatelského rozhraní, včetně ukázek vybraných částí zdrojového kódu. Třetí část kapitoly se zabývá testováním, tedy různými formami automatického testování i uživatelským testováním (spolu s popisem úprav, které byly provedeny na základě uživatelských testů). V poslední části se tato kapitola zabývá publikací aplikace v obchodě Google Play, aby byla využitelná pro veřejnost. Je zde tedy popsáno nastavení monitoringu, vytvoření balíčku i celý proces publikace.

Cíle práce

Tato magisterská práce měla za cíl vytvořit digitální mobilní aplikaci pro Zoologickou zahradu hl. m. Praha v pražské Troji (dále jen "ZOO Praha") na platformě operačního systému Android (dále jen "Android"). Za tím účelem byly vytyčeny dílčí úkoly a cíle (tak jak jsou v zadání práce), tj.:

- navrhnout vhodnou funkcionalitu pro mobilní aplikaci na základě dostupných dat,
- konzultovat s vedoucím práce grafické pojetí aplikace,
- navrhnout a implementovat server v jazyce Node.js, který bude data pro provoz aplikace stahovat z portálu opendata.praha.eu a přes REST API nabízet mobilní aplikaci,
- navrhnout, implementovat a otestovat mobilní aplikaci pro Android,
- publikovat aplikaci na online službě pro distribuci aplikací Google Play.

Tyto dílčí úkoly a cíle sledují dříve schválené zadání, a související práce na nich je zde popsána v jednotlivých kapitolách. Ty zobrazují celý vývoj mobilní aplikace pro Android včetně k ní přidruženého serveru, a to od návrhu funkcionalit, až publikaci aplikace.

Kapitola 1

Návrh funkcionalit mobilní aplikace

Funkcionality aplikace (souhrn poskytovaných funkcí) se odvíjí především od dostupných dat. Dostupná data jsou v tomto případě data zveřejněná na portálu opendata.praha.eu. Tento portál byl vytvořen na základě programového záměru schváleného komisí ICT Rady hl. města Prahy v roce 2015 [5]. Vyplývá z politiky Evropské Unie zaměřené na zvěřejňování datových sad veřejných subjektů ve strojově čitelné formě, dostupné přes programové rozhraní (či jako jednorázové stažení tam, kde je to vhodnější) [6]. Tato data jsou volně využitelná pro komerční i nekomerční účely. Není tedy třeba zabývat se licencí a data lze využít pro aplikaci v rámci této práce.

Kromě rozboru dostupných dat je provedený i průzkum aktuálně fungujících podpůrných aplikaci pro ZOO Praha na platformě Android.

1.1 Dostupná data

Dostupná data využitá v aplikaci jsou zveřejněná Pražskou ZOO na portálu opendata.praha.eu.

1.1.1 Datové sady

Zveřejněná data jsou rozdělená do čtyř datových sad, kterými jsou *Akce v zoo*, *Návštěvnost*, *Lexikon zvířat* a *Adopce zvířat*.

Akce v zoo

Tuto datovou sadu v aplikaci lze využít. Obsahuje propagační akce jako jsou například vernisáže, či komentovaná krmení zvířat. Zjevnou možností pro využití těchto dat je vytvoření seznamu nadcházejících událostí. Dále je možné implementovat zasílání upozornění na nadcházející či právě probíhající akce.

Tato datová sada je publikovaná ve třech formátech – *CSV*, *RSS* a *JSON*. Data ve formátu *CSV*, která jsou také přístupná v datovém API, jsou zde ale velmi zastaralá. Obsahují pouze 24 akcí z roku 2017. Data v *RSS* formátu jsou aktuální, ale obsahují pouze akce, které již nastaly. To by bylo možné využít pro upozornění uživatele, ovšem je nutné zvážit, zda uživatele zajímají již uplynulé akce. Data ve formátu *JSON* jsou aktuální a obsahují nadcházející akce. Díky tomu lze data v tomto formátu jednoduše využít pro seznam nadcházejících akcí. Nutno podotknout, že pokud by všechny formáty obsahovaly stejná data, tak by volba formátu byla pouze věc osobních preferencí.

Návštěvnost

Datová sada obsahuje pouze návštěvnost ZOO Praha dělenou dle let (od roku 2004 do roku 2014) a pro každý rok datum, kdy ZOO navštívil miliontý návštěvník. Možnosti pro využití této sady zde nejsou. Běžný uživatel tato data nevyužije a i v případě jejich vystavení v aplikaci jako zajímavost je zde problém jejich značné zastaralosti, kdy poslední údaj je starý již několik let.

Seznam návštěvností lze stáhnout ve formátu *CSV*. Ta samá data jsou dostupná přes datové API.

Lexikon zvířat

Tato datová sada je primárním zdrojem pro aplikaci. Obsahuje seznam všech zvířat, která se nacházejí v zoo. Tento seznam obsahuje i všechny podrobné informace o zvířatech. Těmi jsou:

- Název
- Latinský název
- Podrobnosti
- Odkaz na fotografii
- Řád
- Třída
- Chov v zoo
- Rozmnožování
- Potrava
- Biotop
- Rozšíření
- Kontinent
- Projekty v zoo
- Proporce

- Zajímavosti
- Lokalita v zoo

Kromě tohoto nejdůležitějšího seznamu obsahuje i seznamy těchto jednotlivých informací o zvířatech. Tedy tyto: Seznam biotopů, seznam tříd, seznam potrav, seznam kontinentů a seznam lokalit v zoo (pavilony a části zoo). Tyto seznamy lze využít pro filtrování lexikonu. Například zobrazení všech zvířat v konkrétním pavilonu nebo všech ptáků.

Seznam zvířat lze stáhnout ve formátech *XLSX* a *CSV*. V obou případech se jedná o ta samá data která jsou dostupná přes datové API portálu opendata.praha.eu. Třetí možností je formát *JSON*, který lze získat z API portálu zoopraha.cz. Při jejich porovnání bylo zjištěno, že data z API zoopraha.cz jsou odlišná od dat na portálu opendata.praha.eu. Data z portálu zoopraha.cz jsou ovšem aktálnější, a proto jsou pro využítí v aplikaci vhodnější. Ostatní seznamy nejsou dostupné přes portál zoopraha.cz, ale pouze přes datové API portálu opendata.praha.eu.

Adopce zvířat

Datová sada obsahuje seznam zvířat, která je možné adoptovat. Obsahuje jejich český a anglický název, třídu a cenu adopce v Kč. Tento seznam je možné využít buď jako zobrazení seznamu zvířat k adopci v aplikaci nebo v něm vyhledat, zda je zobrazené zvíře možno adoptovat, a to přidat k informacím. Lze samozřejmě využít obě možnosti.

Data lze stáhnout ve formátu *CSV*. Jsou také dostupná přes datové API.

1.2 Existující aplikace pro Android

ZOO Praha nemá vlastní oficiální mobilní aplikaci pro Android. Při hledání jiných aplikací jsou brány v úvahu pouze aplikace dostupné přes obchod *Google Play* [7]. Tento obchod obsahuje k červnu 2019 přibližně 2,700,000 aplikací [8]. Běžný uživatel systému Android vyhledává aplikace právě zde a jiné obchody tedy nejsou prozkoumány. Součástí zadání této práce je zveřejnění aplikace práve v tomto obchodě a je tedy vhodné se na něj zaměřit.

Vyhledání různých kombinací slov “lexikon”, “zoo”, “praha” a “zvířata”, byly nalezeny celkem tři aplikace mající informace o ZOO Praha, či o jejích zvířatech.

Údolí Slonů

Tato aplikace je vyvinutá firmou Aitom. Aplikace se zameřuje na jednu konkrétní část ZOO Praha – Údolí slonů. Nabízí informace nejen o chovu v ZOO Praha, ale také obecně o slonech, jejich úloze v náboženství a obsahuje naučný kvíz. Protože je aplikace zaměřena pouze na jednu část ZOO Praha, nenabízí

komplexní přehled o všech zvířatech v ZOO a je tedy zaměřena spíše na uživatele nadšené pro slony.

Aplikace byla naposledy aktualizována v březnu 2017, a jelikož se v současnosti zabývá firma Aitom tvorbou webů, dá se předpokládat, že aplikace nebude dále rozvíjena. Dle Google Play má aplikace přes tisíc stažení. [9]

Pavilon Želv

Tato aplikace je vyvinutá firmou Aitom, tedy stejnou firmou jako v případě aplikace Údolí Slonů. Je jí také velmi podobná, ovšem zaměřuje se na jinou část ZOO Praha – Pavilon Želv. Nabízí informace o želvách v ZOO Praha i o želvách obecně a k tomu lze opět nalézt kvíz. Vyskytuje se tu stejný problém jako u pžedchozí aplikace, což je velmi úzké zaměření, které nemíří na všechny návštěvníky ZOO Praha, ale spíše na konkrétní nadšence.

Poslední aktualizace aplikace proběhla v září 2015 a má pouze přes sto stažení, což je nejméně z nalezených aplikací. [10]

Obě tyto aplikace vyvinuté firmou Aitom jsou si velice podobné a jsou zaměřené na konkrétní pavilony. Pravděpodobným problémem u nich je, že běžný uživatel by si pro návštěvu ZOO Praha nechtěl stahovat aplikací více. Je možné, že budou další aplikace přibývat pro další pavilony, ovšem od poslední aktualizace uběhly již téměř tři roky.

Lexikon zvířat Zoo Praha (předběžný přístup)

Tato aplikace je zaměřená nejen na konkrétní pavilon, ale na celou ZOO Praha. Obsahuje seznam všech zvířat v ZOO Praha, jejich detaily, kalendář akcí a mapu. Dá se předpokládat, že vychází ze stejných dat, protože informace v popiscích zvířat jsou shodné s datovým zdrojem.

Aplikace byla naposledy aktualizována v lednu 2018. Je tedy ze všech nalezených nejaktuálnější a má dle Google Play přes sto stažení. Aplikace se ovšem nachází v předběžném přístupu. Není tedy dokončená a na většině zařízení není kompatibilní. Aplikaci nebylo možné nainstalovat na Androidu 4.4, 7.0, 8.1 ani 9.0. Google Play ji v žádném z těchto případů nedovolil ani stáhnout do telefonu. Nebylo tedy možné ověřit, zda aplikace funguje s pravidelným aktualizováním obsahu nebo zda byla pouze stažena původní data z opendata.praha.eu, která nejsou aktualizována. [11]

1.3 Navržené funkcionality

Dle dostupných dat jsou navrženy následující funkcionality v aplikaci:

- **Zobrazení seznamu všech zvířat v ZOO** – Tento seznam musí být seřazený dle názvu zvířete. Seznam se musí dát filtrovat dle několika kritérií, kterými jsou: *Potrava*, *Biotop*, *Kontinent* a *Lokalita v ZOO*.

Bylo zváženo i filtrování dle třídy, ale seznam tříd obsahuje několik set záznamů, což by pro uživatele bylo spíše zmatečné. V seznamu musí být umožněno textové vyhledávání dle názvu zvířete.

- **Zobrazení detailu zvířete** – Detail zvířete se zobrazí po kliknutí na zvíře v seznamu. Detail musí obsahovat všechny dostupné informace o zvířeti a jeho fotku. Mezi zobrazené informace patří: *Název*, *Latinský název*, *Podrobnosti*, *Řád*, *Třída*, *Chov v zoo*, *Rozmnožování*, *Potrava*, *Biotop*, *Rozšíření*, *Projekty v zoo*, *Proporce*, *Zajímavosti* a *Lokalita v zoo*. Z důvodu velkého množství informací by nebylo vhodné vše zobrazit na jedné obrazovce, a proto musí být informace rozděleny do tří kategorií:
  - **Obecné** – Obsahuje obecné informace o zvířeti, tedy *Podrobnosti* a další krátké informace jako je třeba *Třída*, které mohou být přehledně zobrazeny pod sebou jako výčet.
  - **Rozšíření** – Zde musí být text *Rozšíření* a obrazové znázornění kontinentu. Konkrétnější obrazové informace nejsou v datech k dispozici.
  - **Zajímavosti** – Kategorie obsahuje *Zajímavosti*, *Projekty v zoo* a *Chov v zoo*. Tyto informace jsou totiž delšího textového charakteru.
- **Zobrazení seznamu lokalit v zoo** – Lokalitami se myslí pavilony a další oblasti ZOO Praha, neboli data ze seznamu lokalit. V seznamu musí být zobrazeno jméno lokality a tlačítko odkazující na webovou stránku dané lokality (odkaz je součástí datové sady). Kliknutí na lokalitu v seznamu otevře již zmíněný seznam zvířat, ovšem filtrovaný dle této lokality. Seznam musí být seřazený podle abecedy.
- **Zobrazení seznamu nadcházejících akcí** – Tento seznam musí být seřazený dle data začátku akce (nejbližší akce nahoře) a musí obsahovat začátek akce, konec akce, název akce a popis. Jelikož akce mají jen krátký popis, stačí zobrazit celou akci rovnou v seznamu a není třeba vytvářet speciální obrazovku pro její zobrazení.

1.3.1 Funkční a nefunkční požadavky

Aby bylo možné detailněji monitorovat postup vytváření aplikace a naplnění navržených funkcionalit, je vhodné formálně specifikovat jednotlivé požadavky. Seznam požadavků na aplikaci lze rozdělit na funkční a nefunkční. Funkční požadavky jsou takové, které stanovují, co má aplikace umožňovat, neboli jaké mají být dostupné funkce v aplikaci. Díky vytvoření funkčních požadavků lze později kontrolovat, kde konkrétně v aplikaci jsou tyto jednotlivé požadavky naplněny.

Nefunkční požadavky je takový typ požadavků, který stanovuje omezení na provedení aplikace. To může být například design aplikace, požadavky na

udržitelnost a rozšiřitelnost, podporované systémy apod. Nejedná se tedy o funkcionality v aplikaci, ale o její formu.

Specifikace a označení těchto typů požadavků umožní později v této práci odkazovat na jednotlivé požadavky pomocí jejich identifikátoru.

Funkční požadavky

Tento seznam funkčních požadavků vychází z výše navrhovaných funkcionalit. Proto není popsán do podrobna, ale je popsán bodově.

- **F1.** Aplikace zobrazuje zvířata nacházející se v ZOO Praha
- **F2.** Aplikace umožňuje vyhledání zvířete podle jeho názvu
- **F3.** Aplikace umožňuje filtrovat zvířata dle pavilonu
- **F4.** Aplikace umožňuje filtrovat zvířata dle potravy
- **F5.** Aplikace umožňuje filtrovat zvířata dle kontinentu
- **F6.** Aplikace umožňuje filtrovat zvířata dle biotopu
- **F7.** Aplikace zobrazuje detailní informace o zvířeti
- **F8.** Aplikace umožňuje otevření webových stránek o konkrétním pavilonu
- **F9.** Aplikace zobrazuje nadcházející akce ZOO Praha

Nefunkční požadavky

- **NF1.** Design aplikace je formou standardního *Material Designu* (viz 3.1.2).
- **NF2.** Aplikace podporuje systém Android verze 5.0 a vyšší
- **NF3.** Data v aplikaci odpovídají aktuálním datům poskytovaným od ZOO Praha. Maximálně s jednodenním zpožděním.

Kapitola **2**

Implementace serveru

Součástí aplikace je vytvoření a nasazení serveru v Node.js, který pravidelně aktualizuje data ze zdroje a vystavuje je přes REST API mobilní aplikaci. Vytvoření serveru tedy lze rozdělit na několik částí. První je zvolení vhodné databáze. Dále vytvoření Node.js serveru, který aktualizuje data v databázi a vystavuje je přes REST API. Nakonec nasazení obou částí do veřejné cloud služby.

Výše zmíněné technologie jsou blíže popsány v souvisejících sekcích této kapitoly.

2.1 Databáze

Pro uložení dat je vybrána databáze *MongoDB*. Tato databáze je z rodiny tzv. *NoSQL* databází, které nevyužívají tradiční tabulková schémata relační databáze. Jedná se o dokumentově orientovanou databázi, která se vyznačuje jednoduchostí užití, přičemž je stále velmi rychlá. Dokumenty jsou uloženy ve formátu JSON. Díky orientaci na dokumenty není třeba seznamy rozdělovat do množství tabulek a každý seznam lze jednoduše uložit jako jeden dokument. [12]

Protože databáze nemá klasické relační schéma, neobsahuje některé komplexní funkce jako je například *join* (to je záměrné rozhodnutí, protože takové funkce je obtížné poskytnout při zachování rychlosti). Tyto funkce ale nejsou pro práci potřeba, protože data nemají tak komplexní formu. Stále jsou ale dostupné funkce jako je filtrování, či vyhledávání v seznamu, které jsou potřeba pro aplikaci.

Databáze musí obsahovat dokumenty, které částečně kopírují formu dat rozebraných v sekci *Dostupná data*. Jedná se tedy o následující dokumenty:

- Animals – seznam všech zvířat a informací o nich. Informace o zvířatech jsou stejné jako jsou v původních datech s jedním rozdílem. Původní

data mají podrobnosti o zvířeti ve formátu *HTML*, ve kterém je i odkaz na fotografii zvířete. V databázi je toto potřeba uložit odděleně.

- Localities – seznam lokalit v ZOO Praha.
- Biotopes – seznam biotopů.
- Classes – seznam tříd zvířat.
- Foods – seznam druhů potravy.
- Continents – seznam kontinentů.
- Actions – seznam nadcházejících akcí ZOO Praha.

Dokumenty jsou zde pojmenovány v Anglickém jazyce, protože se jedná o jejich přesný název v databázi a všechen kód je psaný v anglickém jazyce.

2.2 Node.js server

Server zde má dvě hlavní funkce. Tou první je udržování aktuálních dat v databázi, tedy stažení dat z datového zdroje, transformace dat pro potřeby databáze a uložení do databáze. Druhou funkcí je vystavení dat ve formě REST API. Server musí být dle zadání napsaný v *Node.js*.

Node.js

Node.js je softwarový systém navržený pro psaní webových serverů. Systém je určený pro tvorbu vysoce škálovatelných aplikací díky jeho zaměření na asynchronní zpracování požadavků a událostí. [13] Funkce Node.js je založena kolem smyčky událostí, která je v tomto systému před uživatelem skrytá (není třeba ji nijak startovat nebo ukončovat). Po spuštění programu Node.js je na hlavním vlákně spuštěna tato smyčka, která monitoruje příchozí události. Node.js zpracovává I/O (Input/Output – například získání dat z databáze) operace inherentně asynchronně tak, že hlavní vlákno je přidá do fronty požadavků. Tyto požadavky jsou jinými vlákny zpracovány (mezitím může hlavní vlákno obsloužit další příchozí událost) a po jejich vyřízení je zpětně upozorněno hlavní vlákno, které poté vyřídí zbytek požadavku. Díky tomuto přístupu je Node.js velmi vhodný pro programy, které jsou intenzivně zaměřené na I/O operace – což je přesně případ tohoto serveru, který vykonává především operace s databází.

Aplikace v Node.js jsou psané v jazyce JavaScript, který je nativně podporován. Je ovšem možné psát aplikace i v jiných jazycích, které podporují kompilaci do jazyku JavaScript. [14]

REST

REST, neboli Representational State Transfer je architektura rozhraní, poprvé popsaná v dizertační práci Roye Fieldinga s názvem *Architectural Styles and the Design of Network-based Software Architectures* z roku 2000. [15]

REST obsahuje několik architektonických omezení:

- **Client-Server** – REST funguje formou Client-Server, kde server je aplikace nabízející služby. Klient tyto služby může využít zasláním požadavku na server. Server tento požadavek buď odmítne nebo vykoná a zašle odpověď klientovi.
- **Statelessness** – Statelessness, neboli bezstavovost znamená, že server ani klient neuchovávají informace popisující stav spojení. Každý požadavek klienta tedy obsahuje všechny informace nutné k jeho vykonání.
- **Cache** – *"Cache funguje jako mediátor mezi klientem a serverem, kde odpovědi na předchozí požadavky mohou být, pokud jsou považovány za kešovatelné, znovu použity v odpovědích na pozdější požadavky, které jsou ekvivalentní."* [16] (překl. autor) Tento přístup umožňuje snížit počet interakcí mezi klientem a serverem, což zvyšuje rychlost.
- **Uniformní rozhraní** – Uniformita rozhraní je důležitým prostředkem pro zjednodušení architektury. Umožňuje také oddělení implementace od rozhraní, což poskytuje možnost samostatného vývoje obou částí. Z tohoto vychází omezení pro rozhraní:
  - Identifikace dat v požadavku – server může zaslat data v jiném formátu, než je jejich interní reprezentace na serveru.
  - Manipulace dat pomocí jejich reprezentací – pokud má klient data konkrétní instance, má i dost informací aby mohl provést odstranění nebo modifikaci na serveru.
  - Popisné zprávy – každá zpráva má sama o sobě dostatečné informace pro její přečtení.
  - Hypermedia jako nositel aplikačního stavu – server zároveň s daty poskytuje odkazy k získání souvisejících dat.
- **Layered system** – Musí umožňovat rozdělení do vrstev tím způsobem, že klient neví, zda komunikuje přímo se serverem, či s přidaným mezistupněm. Díky tomu lze přidat vrstvy pro cache na straně serveru, vyvažování zátěže a podobně.
- **Code-On-Demand** – Server umožňuje klientovi stažení spustitelného kódu za účelem zjednodušení klientů. Kvůli snížení přehlednosti je toto pouze volitelné.

Webové aplikace naplňující celý REST standard poskytují CRUD operace, tedy Create, Read, Update, Delete (Vytvoření, Čtení, Aktualizace, Odstranění), pomocí klasických HTTP metod. GET pro čtení, PUT pro aktualizace, POST pro vytvoření a DELETE pro odstranění. Protože server poskytuje aplikaci pro Android data přes internet, nejjednodušší je využití právě tohoto HTTP standardu.

Implementace

Protože pro navržené funkcionality není potřeba data zapisovat ani měnit, ale pouze je poskytovat aplikaci, není třeba ani naplnit celý standard REST. Z CRUD operací je třeba podporovat pouze čtení. Ostatní operace jsou nežádoucí i z toho důvodu, že data server získává z datového zdroje na `opendata.praha.eu`. Jakékoliv operace upravující tato data server nemůže podporovat, protože je potřebuje pouze vystavit pro čtení.

Server tedy musí obsluhovat příchozí GET požadavky. Data lze vystavit na jednotlivých URI (Uniform Resource Identifier – text popisující zdroj informací) způsobem, který je hojně využívaný ve webových REST službách. Tedy `<adresa-serveru>/api/<resource>`, kde `<resource>` je název kolekce. Tedy například zaslání požadavku GET na adresu `<adresa-serveru>/api/animals` vrátí v odpovědi *JSON* obsahující seznam všech zvířat. Pro získání pouze jedné konkrétní instance z kolekce je poté třeba použít adresu `<adresa-serveru>/api/<resource>/<id>`, kde `<id>` je identifikátor konkrétní dotazované instance. Pokud tedy uživatel ví, že identifikátor Lva Indického je 12, pak zaslání požadavku GET na adresu `<adresa-serveru>/api/animals/12` vrátí v odpovědi všechny informace o Lvu Indickém. [17]

Je třeba vystavit kolekce, které jsou uložené v databázi. Ty budou dostupné na následujících adresách:

- `/api/animals` – Vystavuje datábazovou kolekci Animals
- `/api/localities` – Vystavuje datábazovou kolekci Localities
- `/api/biotopes` – Vystavuje datábazovou kolekci Biotopes
- `/api/classes` – Vystavuje datábazovou kolekci Classes
- `/api/foods` – Vystavuje datábazovou kolekci Foods
- `/api/continents` – Vystavuje datábazovou kolekci Continents
- `/api/actions` – Vystavuje datábazovou kolekci Actions

U adresy `/api/animals` lze využít výše zmíněné `<id>` zvířete – to je potřeba pro funkcionalitu zobrazení informací o zvířeti. Ostatní kolekce jsou v aplikaci vždy potřeba v celku, pročež není nutně potřeba, aby server nabízel tuto službu i u ostatních adres.

K obsloužení jednotlivých URI lze v Node.js využít balíček `express`. Tento balíček poskytuje třídu `router`, která umožňuje vytvoření funkcí, jež jsou provedeny v případě příchozího požadavku na konkrétní adresu. Vytvoření funkce odpovídající na příchozí požadavek GET pro kolekci Animals vypadá takto:

```
//pro URI '/animals' a požadavek GET
router.route('/animals').get(function(req, res) {
    //vyhledání v databázi zvířat
    Animal.find(function(err, animals) {
        //odeslání příslušné odpovědi v případě chyby
        if (err)
            res.send(err);
        //odeslání zvířat ve formátu JSON
        res.json(animals);
    });
});
```

Obdobně bude vypadat přidání funkce pro vyřízení požadavku obsahujícího identifikátor zvířete:

```
//pro URI '/animals', kde následuje id a požadavek GET
router.route('/animals/:ani_id').get(function(req, res) {
    //vyhledání v databázi podel identifikátoru
    Animal.findById(req.params.ani_id, function(err, ani) {
        //odeslání příslušné odpovědi v případě chyby
        if (err)
            res.send(err);
        //odeslání zvířete ve formátu JSON
        res.json(ani);
    });
});
```

Vytvoření těchto funkcí je potřeba také pro všechny ostatní kolekce. Aby router vyřizoval tyto požadavky, je už jen třeba přidat ho do aplikace před spuštěním serveru následovně.

```
//využití instance 'router' pro všechny
//příchozí požadavky na URI '/api'
app.use('/api', router);
```

Těmito kroky je vyřízeno vytvoření webového rozhraní. K routeru lze, kromě funkcí vyřizujících odpovědi na požadavky, přidat i middleware funkce, které jsou vykonávány při každém požadavku. Toho lze využít například pro ukládání informací o probíhajících připojeních na server.

Dále je nutno vytvořit funkce, které budou pravidelně stahovat data z datových zdrojů a ukládat je do databáze. K tomuto účelu je potřeba stáhnout data, provést jejich transformaci do schématu databáze a poté je uložit. V Node.js na práci s databází *MongoDB* lze použít balíček `mongoose`. S tímto balíčkem je potřeba nadefinovat databázové schéma a zadat adresu databáze. Schéma lze nadefinovat zadáním jednotlivých datových položek a jejich typů. Mongoose pak nabízí řadu funkcí pro práci s databází.

Po vytvoření schématu `Animal` lze tedy vytvořit funkci `updateAnimals` pro stažení a uložení dat následovně:

```
//adresa datového zdroje (zde zkráceno)
var baseURI = 'https://www.zoopraha.cz/_api/...';
//odeslání požadavku na datový zdroj
request(baseURI, (err, res2, body) => {
    //v případě chyby neprovádět nic
    if(err) {return}
    //zpracování JSON odpovědi
    output = JSON.parse(body.toString().slice(1));
    //upravení objektů dle databázového schématu
    for (var i = 0, len = output.length; i < len; i++) {
        getPropertiesInAnimalObject(output[i]);
    }
    //'Animal' je schéma dokumentu v databázi
    //odstranění starých dat z databáze
    Animal.remove({}).exec();
    //vložení nových dat do databáze
    Animal.insertMany(output, function(error, docs)
        { if(error) {console.log(error);}});
});
```

Funkce `getPropertiesInAnimalObject` se zde stará o zpracování a přejmenování dat objektu tak, aby odpovídal databázovému schématu. S využitím balíčku mongoose lze pak data jednoduše uložit do databáze použitím funkce `insertMany`.

Obdobné funkce a schémata je nutné vytvořit i pro ostatní seznamy. Některé funkce se liší více, protože datový zdroj je jiný a tím i reprezentace dat. Zpracování dat tak, aby odpovídala schématu databáze, se pak také liší.

Všechny funkce pro obnovení dat v jednotlivých seznamech jsou volány periodicky jednou denně. Data se nemění natolik často, aby bylo potřeba je obnovovat v kratších intervalech. Ovšem delší interval je nežádoucí především kvůli seznamu nadcházejících akcí. Ten by nebylo vhodné obnovovat méně často, protože by docházelo v aplikaci k zobrazování již uplynulých akcí. Jelikož aplikace získává data prostřednictvím tohoto serveru, je touto pravidelnou obnovou naplněn nefunkční požadavek **NF3.** ze sekce 1.3.1.

2.3 Nasazení

Aby byl server dostupný aplikaci, je potřeba zajistit, aby byl přístupný přes internet. Toho lze docílit buď nasazením na vlastním serveru nebo využitím cloudových služeb. Pro tuto aplikaci je vhodnější cloudové řešení. To hlavně z důvodu nižších vstupních nákladů, jelikož není nutná investice do vlastních výpočetních kapacit. V případě nutnosti umožňuje i jednoduché škálování, které by bylo u vlastního serveru finančně náročné.

2.3.1 Nasazení databáze

Pro cloudové nasazení MongoDB nabízí řešení přímo tvůrci MongoDB. Toto řešení se jmenuje Atlas a dle popisu v dokumentaci je to "*plně řízená cloudová databáze vyvinutá stejnými lidmi, kteří vyvíjejí MongoDB. Atlas řeší komplexity nasazení, správy a obnovy nasazení u poskytovatelů cloudových služeb AWS [Amazon Web Services], GCP [Google Cloud Platform] a Azure*". [18] Narozdíl od jiných poskytovatelů také nabízí bezplatnou verzi, která je pro potřeby této databáze dostačující.

Pro nasazení databáze je potřeba vytvořit tzv. Cluster. Cluster může obsahovat různé databáze, které jsou společně spravovány. Cluster M0, tedy ta úřoveň, která je bezplatná, poskytuje možnost spravovat až 100 databází nebo 500 kolekcí o celkové velikosti až 512 MB. [19] To je pro potřeby vývoje a prvotního nasazení dostatečné. Jak bylo výše zmíněno, výhodou cloudového nasazení je škálovatelnost. Pokud by tato úroveň v budoucnu přestala být dostatečná, lze ji zvýšit dle potřeby.

Atlas nabízí různé regiony po světě, kam lze cluster nasadit. Pro tuto databázi je vhodný region Belgie. Je to evropský region, což ho staví geograficky nejblíže. Další regiony dostupné pro tuto úroveň clusteru jsou Iowa (USA), Sao Paulo (Brazílie), Taiwan a Singapur. Tato úroveň clusteru navíc poskytuje replikaci do tří "replica sets". Replikace znamená, že databáze má několik kopií na různých serverech. Tím se zvyšuje odolnost vůči ztrátě jednoho z databázových serverů a tedy dostupnost dat. [20]

Tímto je databáze nasazena a lze se pomocí adresy k ní připojit. Po nasazení je tato adresa dostupná v informacích o clusteru a musí být zkopírována do zdrojového kódu Node.js serveru. Proto je nejdříve nutné vytvoření a nasazení databáze a až poté nasazení serveru.

2.3.2 Nasazení serveru

Google Cloud Platform (dále jen GCP) je veřejná cloudová platforma od společnosti Google. [21] Řadí se mezi největší cloudové platformy na trhu s ročním obratem přibižně 8 miliard dolarů [22] a nabízí se jako vhodná platforma pro nasazení serveru z několika důvodů.

AppEngine od GCP je PaaS (Platform as a service – platforma jako služba), zaměřená na vytvoření a nasazení webových aplikací. [23] Protože se jedná a Paas a ne o IaaS (Infrastructure as a service – infrastruktura jako služba), lze v AppEngine nasazovat pouze aplikace v těch jazycích, které jsou zde podporované. Nabízí díky tomu ale jednoduché a přehledné nasazení i správu a AppEngine se sám stará o poskytnutí serverů a o škálování aplikací. Kromě jiných jazyků nabízí i podporu pro Node.js systém a lze zde tedy nasadit server.

Dále, pokud porovnáme GCP s Amazon Web Services (dále jen AWS), který je největším veřejným poskytovatelem cloudových služeb, zjistíme mimo jiné dvě hlavní výhody GCP. [21]

- AWS nabízí jako své PaaS Beanstalk. Ten je ovšem pouze správcovskou vrstvou nad AWS EC2 (výpočetní cloudová služba) a potřebuje tedy neustále běžící instanci, čímž se zvyšují náklady na provoz oproti AppEngine, který toto – jakožto čistá PaaS – nevyžaduje.

- GCP nevyžaduje žádný poplatek předem a účtuje zpětně za využité služby. AWS naopak nabízí zarezervování a zaplacení služby předem. To může být levnější, ale pouze v tom případě, že poskytnutá výpočetní kapacita je využívána nonstop po celou dobu. Tento přístup je nevýhodný obzvlášť v počátku nasazení aplikace, kdy má pouze omezený počet uživatelů, či je teprve vyvíjena.

AppEngine nabízí dvě prostředí zvaná Standard a Flexible. [24] Prostředí Flexible podporuje více jazyků a nabízí více funkcí. Běží nad instancí Google Compute Engine, což je IaaS, a umožňuje díky tomu přizpůsobení infrastruktury, přístup k operačnímu systému virtuálních strojů apod. S tím ale přichází vyšší cena, a protože Node.js server nepotřebuje přístup k operačnímu systému a Node.js je podporovaný i v prostředí Standard, je toto prostředí vhodnější.

Region, kde má být instance nasazena je potřeba zvolit v Evropě. Předpokládá se, že uživatelé aplikaci využijí převážně v Evropě při návštěvě ZOO Praha, a proto musí být server geograficky co nejblíže kvůli snížení doby odpovědi. Dle toho byl zvolen i region pro Databázi výše.

Nasazení a spuštění v Cloudu

GCP obsahuje unixovou konzoli, kde je mimo jiné dostupný verzovací systém `git`. Pro nahrání zdrojového kódu serveru na GCP stačí tedy naklonovat repozitář. Protože AppEngine je platforma, je spuštění instance jedoduché. Do zdrojového kódu serveru je nejdřív nutno přidat soubor s názvem `app.yaml`, který obsahuje informace o nastavení projektu pro AppEngine. V tomto případě bude tento soubor obsahovat pouze jeden řádek `runtime: nodejs10`. Lze v tomto souboru nastavit i výše zmíněné prostředí, ale prostředí Standard

je výchozí. Jelikož bylo v souboru řečeno o jaký jazyk/systém se jedná, AppEngine provede potřebná nastavení a do konzole nyní stačí napsat příkaz pro spuštění projektu `gcloud app deploy`. Poté je server dostupný na adrese přidělené tomuto projektu.

Upozornění na nedostupnost serveru

GCP poskytuje Marketplace, což je místo, kde lze vybrat různá cloudová řešení a přidat je ke svému projektu. Jedním z těchto řešení je Stackdriver. Stackdriver nabízí možnosti pro monitoring běžícího serveru pomocí tzv. Uptime Checks.

Uptime Checks je opakované automatické zasílání kontrolních dotazů serveru. Pokud server přestane odpovídat nebo odpoví špatně a tato nesprávná odpověď se opakuje po určitou dobu, je server označen jako nefunkční. V tom případě se dají spustit různá opatření. Zaslání informace o nedostupnosti vývojáři, automatický restart serveru a podobně.

Na Stackdriveru lze nastavit, jak má Uptime Check vypadat (jaká má být forma dotazu a jaká má být odpověď na dotaz) a délka periody (od jednoho dotazu za 15 minut až po jeden dotaz každou minutu). K tomu lze nastavit různé Policies. Ty popisují v jakém případě jsou vyvolány (jak dlouho je server nedostupný) a jakým komunikačním kanálem se má vývojáři odeslat upozornění o nedostupnosti.

Díky nastavení Stackdriveru tak dojde ke zrychlení reakce na případnou nedostupnost serveru, kdy vývojář nemusí ručně kontrolovat dostupnost, ale je o ní případně informován.

2.3.3 Dokumentace API

Pro dokumentaci API je využitý nástroj *Swagger*. *Swagger* je open-source systém poprvé uvedený v roce 2011, zaměřený na design a dokumentaci webových služeb formátu REST. Je podporovaný rozhraním na internetové stránce `swagger.io`, kde lze jednoduše vytvořit dokumentaci API. Po přihlášení je zde poskytována možnost zadání požadavků na API, které uživatel chce zdokumentovat. Po jejich zadání *Swagger* vygeneruje definici API ve formátu OAS3.

OAS3, neboli OpenAPI Specification je formát navržený pro dokumentaci rozhrání formátu REST tak, aby byl dokument čitelný strojově a zároveň přehledný pro lidi. [25]

Po vygenerování definice byly provedeny další úpravy, jako jméno API, popis obsahu jednotlivých kolekcí a jejich schémat. Část této definice lze vidět v následujícím kódu:

```
/api/biotopes:
get:
  description: the list of biotopes where animals live
```

```
responses:
  '200':
    description: OK
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
              type: integer
              description: index
            name:
              type: string
              description: name of the biotope
```

Úryvek začíná cestou v API, která vede na dokumentovanou kolekci. Následuje její popis, kde je zmíněno, že se jedná o seznam bitopů, kde žijí zvířata. Dále je kód odpovědi, kterou má server vrátit a schéma obsahu. Zde je ve schématu identifikátor a název biotopu. Lze zadat i příklad a další informace, ale ty jsou z úryvku vynechány.

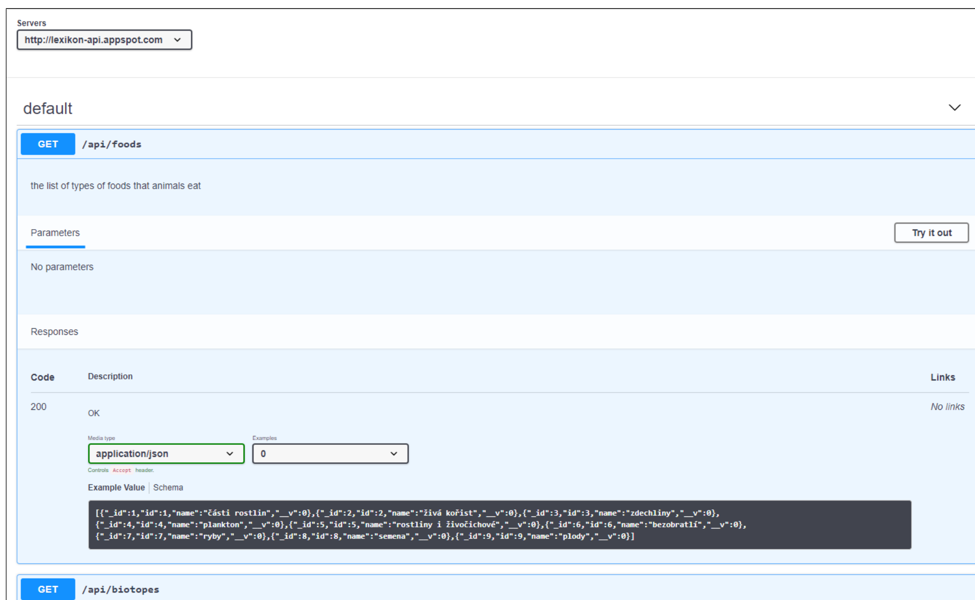
Na základě této definice pak nabízí *Swagger* možnost vytvoření dokumentace ve formě webových stránek. Vygenerovaná dokumentace je součástí příloh této práce. Online verzi dokumentace lze najít také na adrese `https://app.swaggerhub.com/apis-docs/ATDumbass/lexikon-api/1.0.0-oas3`. Část dokumentace vystavené na této stránce lze vidět na obrázku 2.1.

2.3.4 Testování API

Kromě výše zmíněné kontroly nedostupnosti serveru, jsou k API vytvořeny i testy, které kontrolují správnost odpovědí pro jednotlivé požadavky zaslané na server. K tomuto účelu byla využita aplikace *Postman*, což je aplikace určená ke zjednodušení vývoje a testování webových služeb.

V *Postman* aplikaci lze vytvořit kolekce požadavků, které je poté možné hromadně zasílat na API. K jednotlivým požadavkům v kolekci lze vytvářet testy v jazyce Javascript. Pro každou kolekci v API byly vytvořeny dva požadavky. Jeden s dotazem GET na celou kolekci a druhý s dotazem na konkrétní objekt v kolekci. Požadavek na konkrétní objekt je navržen tak, aby na serveru vyvolal chybu a server tak vrátil odpověd s chybovým stavem. Tím se testuje jak to, zda server vrátí správně chybovou hlášku tak to, zda server správně vrátí kolekci.

Například požadavek, zda server vrátí kolekci kontinentů má nastavenou cílovou adresu `http://lexikon-api.appspot.com/api/continents`. Jelikož by se v odpovědi měla vrátit kolekce, jsou k tomuto požadavku přiřazeny tyto testy:



Obrázek 2.1: Dokumentaci API na swaggerhub.com

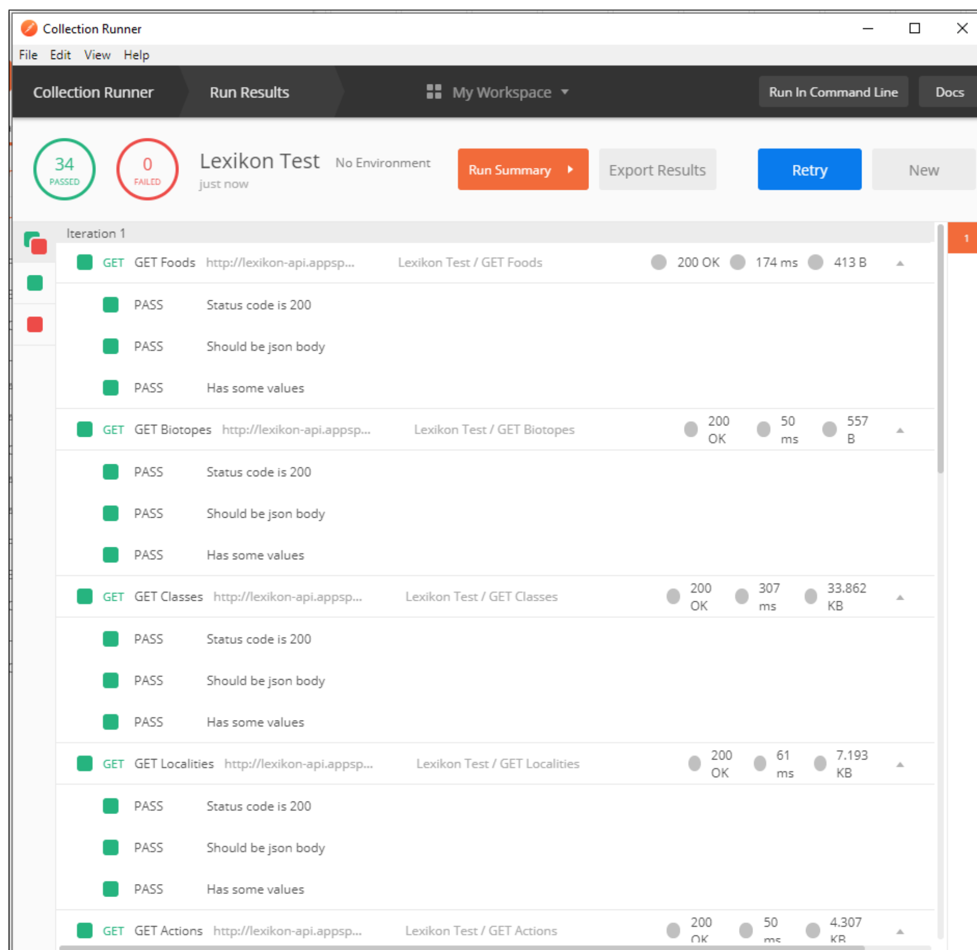
```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
})

pm.test("Should be json body", function () {
    pm.response.to.have.jsonBody();
});

pm.test("Has some values", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an('array').that.is.not.empty;
});
```

První test ověřuje, jestli je status v odpovědi *200*, což znamená, že dotaz byl OK. Druhý test ověřuje, jestli je tělo odpovědi ve formátu *JSON*. Jelikož by testy měly být nezávislé na konkrétních datech – v případě změny obsahu totiž API stále funguje a přitom by testy hlásily chybu – nejsou zde testovány jednotlivé kontinenty. Je ale možné alespoň otestovat, zda obsah navrácených dat je ve formě pole a zda toto pole není prázdné. To je provedeno ve třetím testu.

Obdobné testy jsou vytvořeny pro všechny ostatní požadavky v kolekci. Celkově se jedná o 34 testů. Výsledky testů jsou exportovány do formátu *JSON* a lze je nalézt v přílohách této práce. Zároveň lze na obrázku 2.2 vidět výsledek testování kolekce v *Postman* aplikaci.



Obrázek 2.2: Výsledky testování v aplikaci *Postman*

Kapitola 3

Implementace mobilní aplikace

Tato kapitola se zabývá vývojem mobilní aplikace pro Android. Je zde popsán celý postup vývoje od návrhu uživatelského rozhraní, který vychází z navržených funkcionalit (viz kapitola 1.3) na základě dostupných dat a volba jazyka a celkové architektury aplikace. Následuje sekce zaměřená na implementaci, poté testování – jak automatické unit a integrační testy, tak uživatelské testy. Kapitola je zakončena sekcí popisující proces publikace aplikace v obchodě Google Play.

3.1 Návrh

Před započetím samotného programování aplikace bylo nutné se zamyslet nad jejími funkcemi, rozmyslet, co bude uživateli nabídnuto a v jaké formě a naplánovat, jak bude aplikace vytvořena. Pokud programátor neví, k jakému výsledku má dojít, jen těžko bude hledat správnou cestu. Je tedy důležité začít návrhem uživatelského rozhraní. Tento návrh samozřejmě vychází z navrhovaných funkcionalit z kapitoly 1.3, kde jsou rozebrána dostupná data a jak je lze využít.

Dále je potřeba postoupit k volbě jazyka a k návrhu celkové architektury aplikace. Tím se vytvoří vysokoúrovňový nákres, po čemž lze přistoupit k samotné implementaci.

3.1.1 Návrh uživatelského rozhraní

V kapitole 1.3 jsou na základě dat dostupných v datovém zdroji navrženy tyto funkcionality:

- Zobrazení seznamu všech zvířat v ZOO
- Zobrazení detailu zvířete
  - Obecné

  - Rozšíření
  - Zajímavosti

- Zobrazení seznamu lokalit v zoo
- Zobrazení seznamu nadcházejících akcí

Dle těchto funkcionalit lze navrhnout čtyři hlavní obrazovky. Z toho tři obrazovky budou na nejvyšší úrovni – seznam zvířat, seznam lokalit a seznam akcí. Tyto obrazovky nevyžadují žádné vstupní informace od uživatele, narozdíl od detailu zvířete. Pro zobrazení detailu zvířete je potřeba zjistit, jaké zvíře má být zobrazeno. Toho lze docílit tím, že uživatel zvíře zvolí v seznamu zvířat. Ostatní obrazovky toto nepotřebují, a proto mohou být na nejvyšší úrovni s tím, že se mezi nimi přepíná pomocí navigačního menu.

První zobrazenou obrazovkou po zapnutí aplikace by mohla být domovská stránka s některými vstupními informacemi a rozcestníkem mezi obrazovky na nejvyšší úrovni. Ale za předpokladu, že většina uživatelů otevírá aplikaci za účelem vyhledání informací o zvířeti, by domovská stránka s rozcestníkem zbytečně přidávala jeden krok. Proto je lepší zobrazit rovnou seznam zvířat a jako rozcestník mít již zmíněné navigační menu.

Tímto návrhem se sníží počet kliknutí pro zobrazení seznamu zvířat z 1 na 0. Počet kliknutí potřebných pro zobrazení seznamu lokalit nebo seznamu akcí se ale zvýší z 1 na 2 (je třeba 1. rozkliknout navigační menu 2. zvolit příslušný seznam). Předpoklad, že většina uživatelů jako první seznam potřebuje seznam zvířat bohužel nelze v současné chvíli statisticky potvrdit. To bude možné až po určité době používání aplikace, pokud aplikace bude zaznamenávat, kam uživatelé nejčastěji směřují, či pokud se to bude opakovaně objevovat ve zpětné vazbě. V tuto chvíli bude dále práce vycházet z tohoto předpokladu.

Seznam zvířat

Jak bylo navrženo v kapitole 1.3, tato obrazovka musí splňovat několik náležitostí. Jednou z nich je seřazení podle abecedy kvůli jednoduchosti vyhledávání. To je sice nutno zmínit, ale na návrh uživatelského rozhraní (dále jen *UI*, podle anglického User Interface) nemá tento požadavek další dopad. Tato obrazovka naplňuje funkční požadavek **F1.** z 1.3.1.

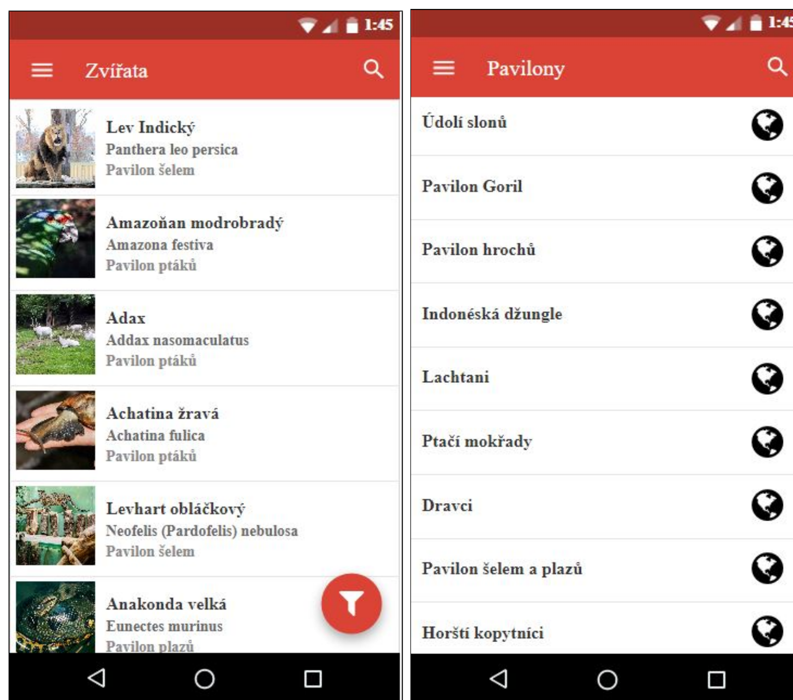
Obrazovka musí obsahovat možnost textového vyhledání zvířete podle názvu. Tím bude na této obrazovce naplněn funkční požadavek **F2.** z 1.3.1. Vyhledávání by mělo probíhat dynamicky – tedy již při psaní, aby uživatel mohl přestat psát ihned ve chvíli, kdy hledané zvíře uvidí na obrazovce. Zároveň, aby to bylo možné, musí vyhledávání být zobrazeno přímo na obrazovce a ne například formou dialogu či jiné obrazovky. Vyhledávání je tedy vhodné umístit na vrch obrazovky a skrýt ho. Vyhledávání by se pak mělo zobrazit po kliknutí na lupu vpravo nahoře. Skrytí vyhledávání do doby, než je

potřeba, zlepší přehlednost. Použítí lupy jako ikony pro vyhledávání je standardní napříč všemi programy.

Používání stadardních postupů je jedním z klíčových prvků návrhu UI. To se může zdát jako postup omezující inovaci a individuality aplikací, ale silně to zvyšuje efektivitu práce s UI pro uživatele, protože se nemusí na každé aplikaci učit nové postupy, ale běžné aktivity dělá vždy stejně.

Seznam dále potřebuje mít podporu filtrování podle různých kritérií. Kritéria pro filtrování jsou 4 a jsou to funkční požadavky **F3.**, **F4.**, **F5.** a **F6.** z 1.3.1. To je již vhodnější vložit do dialogu, kde uživatel zvolí jednotlivá kritéria. Pro vyvolání dialogu lze využít floating action button (kulaté tlačítko v pravém dolním rohu obrazovky) s ikonou filtrování. Tímto dialogem jsou tedy naplněny výše zmíněné funkční požadavky.

Nyní zbývá už jen navigační menu mezi jednotlivými seznamy. To se standardně vyvolává v levém horním rohu obrazovky a proto je vhodné umístit ho tam kvůli dodržení standardů. Výsledný návrh obrazovky vypadá jako na obrázku 3.1.



Obrázek 3.1: Návrh obrazovky seznamu zvířat (vlevo) a seznamu pavilonů (vpravo)

Seznam lokalit

Seznam zvířat je možné filtrovat podle lokality. Protože je ale pravděpodobným případem užití to, že si uživatel chce vyhledat, jaká zvířata se nacházejí v tom pavilonu, ve kterém je právě teď on, je vhodné uživateli nabídnout jednodušší procházení lokalit. Z toho důvodu je navržena obrazovka seznamu lokalit.

Lokalit v zoo není mnoho a lze tedy udělat pouze seznam zobrazující lokality seřazené dle abecedy podle názvu. V případě kliknutí na položku v seznamu se zobrazí seznam zvířat nacházejících se v dané lokalitě. Toho lze docílit otevřením obrazovky *Seznam zvířat*, s přednastaveným filtrem dle lokality. Tím se využije princip znovupoužitelnosti kódu.

Druhou možností by bylo zobrazení speciální obrazovky pro detail lokality. Problémem je to, že v datové sadě mají lokality kromě názvu pouze jednu další informaci, což je odkaz na webové stránky lokality. V detailu lokality tedy nejsou kromě seznamu zvířat další data ke zobrazení. Odkaz na webové stránky lze přidat jako tlačítko přímo do seznamu lokalit, čímž se zároveň naplní funkční požadavek **F8.** z 1.3.1.

Lokalitou se zde myslí pavilony a další části zoo. Protože člověk nemá běžně slovo lokalita spojené se zoologickou zahradou, je v uživatelském rozhraní užíváno slovo *pavilon*. Výsledný návrh obrazovky vypadá jako na obrázku 3.1.

Seznam nadcházejících akcí

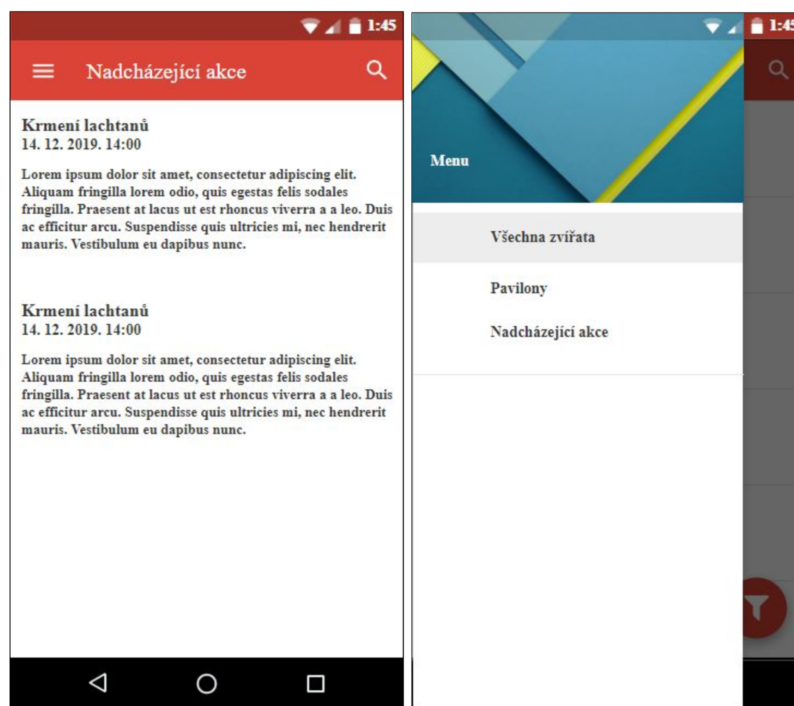
Další navrhovanou obrazovkou je dle kapitoly 1.3 obrazovka se seznamem nadcházejících akcí ZOO Praha, která naplní funkční požadavek **F9.** z 1.3.1. Narozdíl od seznamu lokalit a seznamu zvířat zde není žádoucí řazení dle abecedy. Je vhodnější seřadit akce pro uživatele dle data a času. Uživatel tak uvidí nejdříve ty akce, které jsou nejblíže.

Jak již bylo zmíněno v kapitole 1.3, kromě názvu a času začátku a konce akce, obsahují data pouze krátký popis. Ten se jednoduše vejde přímo do seznamu a vytváření speciální obrazovky pro zobrazení detailu akce je zbytečné.

Výsledná obrazovka je tedy jednoduchá a její návrh je na obrázku 3.2.

Detail zvířete

Na této obrazovce je potřeba zobrazit detailní informace o zvířeti. Zvíře, které má být zobrazeno, je zvoleno v seznamu zvířat – kliknutím na položku v seznamu se otevře tato obrazovka s daným zvířetem. Informace o zvířeti se nevejdou na jednu obrazovku, protože (kromě základních údajů a popisu) jsou k dispozici například zajímavosti či informace o chovu v ZOO Praha, které mohou obsahovat delší text. Informace lze rozdělit na tři segmenty, zmíněné v kapitole 1.3, které jsou: **Obecné**, **Rozšíření** a **Zajímavosti**.



Obrázek 3.2: Návrh obrazovky seznamu akcí (vlevo) a navigačního menu (vpravo)

Pro zobrazení jednotlivých segmentů není potřeba dělat celé různé obrazovky, ale lze využít tzv. ViewPager pro zobrazení záložek. Tento objekt je standardně využívaný v aplikacích pro Android, který zobrazuje určitý počet záložek, mezi nimiž se dá přesouvat pomocí posunutí do stran nebo kliknutím na název záložky v horní části obrazovky.

Fotka zvířete a název (včetně latinského) mohou být pro přehlednost zobrazeny neustále. Toho lze docílit tak, že tyto informace budou zobrazeny v horní části obrazovky a zmíněný ViewPager se bude nacházet pod nimi.

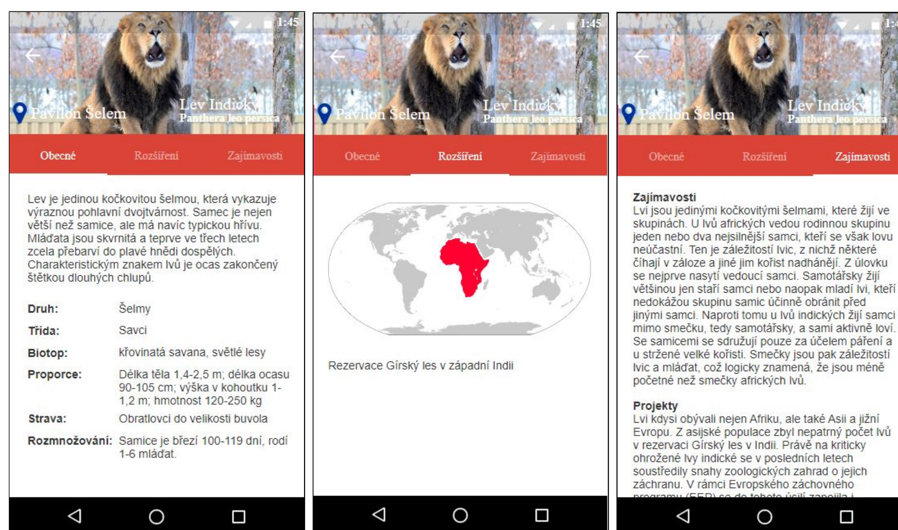
Všechny informace (kromě obrázku) jsou textové, ale například kontinent výskytu zvířete lze převést do obrazové formy. V datech nejsou obrázky rozšíření od ZOO Praha a nelze tedy oblast výskytu zobrazit podrobněji než zabarvením kontinentu.

Vytvořením této obrazovky je naplněn funkční požadavek **F7.** z 1.3.1.

Výsledný návrh obrazovky je na obrázku 3.3.

Naplněné funkční požadavky

Zde je pro přehlednost uvedeno, jaká obrazovka naplňuje jaké funkční požadavky.



Obrázek 3.3: Návrh obrazovky detailu zvířete. Obecné informace (vlevo), rozšíření (veprostřed) a zajímavosti (vpravo)

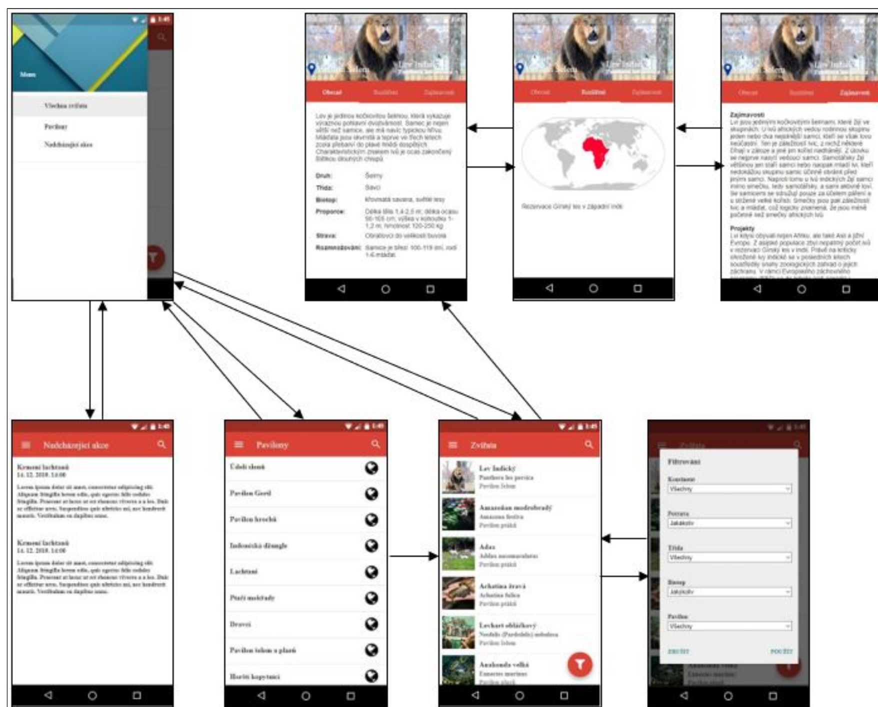
- Obrazovka seznamu zvířat – **F1.**, **F2.**, **F3.**, **F4.**, **F5.**, **F6.**
- Obrazovka detailu zvířete – **F7.**
- Obrazovka seznamu lokalit – **F8.**
- Obrazovka nadcházejících akcí – **F9.**

Ze seznamu je patrné, že jsou návrhem pokryty všechny funkční požadavky stanovené v sekci 1.3.1.

Graf přechodů

Pro přehlednost lze vytvořit graf možných přechodů mezi jednotlivými obrazovkami. Tento graf lze vidět na obrázku 3.4. Na grafu je znázorněno navigační menu jako samostatná obrazovka kvůli zdůraznění rozdílu mezi přímým přechodem mezi obrazovkami a přechodem pomocí navigačního menu.

Ze všech seznamů vede přechod k navigačnímu menu a z něj zase přechod na kterýkoliv seznam. Přímý přechod vede ze seznamu lokalit do seznamu zvířat (zvolením lokality se aplikuje filtr a zobrazí zvířata v dané lokalitě), ze seznamu zvířat k detailu zvířete vybráním zvířete ze seznamu. Dále pak vedou přímé přechody mezi jednotlivými záložkami v detailu zvířete a mezi seznamem zvířat a dialogem pro filtrování.



Obrázek 3.4: Graf přechodů obrazovek

3.1.2 Grafické pojetí aplikace

Pro splnění zadání bylo nutné konzultovat grafické pojetí aplikace s vedoucím práce. Vedoucí práce schválil použití standardního vzhledu používaného v Android aplikacích, tzv. *Material design*. To je zachyceno také formou nefunkčního požadavku **NF1.** v 1.3.1.

Material design

V dokumentaci Material Designu je následující shrnutí: *"Material is an adaptable system of guidelines, components, and tools that support the best practices of user interface design."*. [26] Jedná se tedy o systém směrnic, komponent a nástrojů podporujících správné postupy návrhu uživatelského rozhraní. Jak název napovídá, tak tento design vychází z pojmu "materiál". Designér Googlu, Matías Duarte v souvislosti s tím říká *"Narozdíl od opravdového papíru, náš digitální materiál může inteligentně měnit formu. Materiál má fyzické povrchy a hrany. Švy a stíny poskytují informaci o tom, čeho se lze dotknout"* [27] (překl. autor práce).

Material design byl vyvinut v roce 2014 společností Google a od té doby byla do podoby Material designu převedena většina aplikací Googlu.

Material design má umožňovat konzistentní práci s uživatelským rozhraním, ať je aplikace spuštěná na chytrém telefonu, tabletu nebo počítači. [27]

3.1.3 Jazyk a architektura

Při tvorbě aplikací pro Android má vývojář na výběr ze dvou hlavních jazyků – Java a Kotlin. Java je starší jazyk a má stále větší komunitu než Kotlin. Nevýhodou jazyku Java je to, že pro stejnou funkci často potřebuje mnohem více řádek než Kotlin. Kotlin byl totiž vyvinut roku 2011 společností JetBrains za účelem zjednodušení vývoje aplikací pro Android. [28] Díky tomuto zaměření je vývoj v jazyce Kotlin rychlejší než v jazyce Java.

Oba jazyky mají jisté klady a zápory. Kotlin nabízí řadu funkcí, které v Javě nejsou dostupné a které jsou zaměřeny specificky pro zjednodušení kódu (podpora lambda funkcí, datové třídy apod.) a pro zvýšení bezpečnosti kódu (např. Null-safety). [28]

Protože firma JetBrains nechtěla předělávat všechny své zdrojové kódy z jazyka Java do jazyka Kotlin, byl Kotlin vytvořen tak, aby byl plně interoperabilní s Javou. Je tedy možné dokonce mít část projektu v jazyce Java a část v jazyce Kotlin. Pro jednoduchost Kotlinu byl tento jazyk vybrán jako primární jazyk vývoje této aplikace.

Návrh architektury

Program je nutné rozdělit do několika komponent. Tyto komponenty se musí starat o konkrétní oblasti programu a mezi sebou komunikovat v rámci předem daných rozhraní. Tento přístup je nutný jak pro dlouhodobou udržitelnost programu tak pro zvýšeni přehlednosti kódu. K udržitelnosti přispívají právě předem stanovená rozhraní, která oddělují užití rozhraní od samotné implementace, díky čemuž lze měnit implementaci rohraní aniž by bylo nutné měnit tu část programu, která rozhraní využívá. Mimo dlouhodobé udržitelnosti je využití rozhraní důležité i pro vytvoření automatických unit testů, protože je tím umožněn tzv. *mocking* (více v kapitole 3.3). Rozdělení do komponent snižuje celkovou provázanost a tím, že mají komponenty konkrétní oblasti zájmu, je kód přehlednější.

Jednou z těchto komponent je to, čemu lze říkat "Datová vrstva". Odpovědností této vrstvy je poskytování dat zbytku programu. Standardně to může znamenat, že se vrstva stará o zápis do databáze a čtení z databáze. V tomto případě aplikace nemá offline databázi přímo na zařízení kde běží, ale data jsou získávána ze serveru. Datová vrstva se zde tedy musí starat o připojení k API, které je poskytované serverem vytvořeným a nasazeným v kapitole 2 a o získávání dat z tohoto API. Kvůli inherentní nespolehlivosti internetu (připojení může být pomalé a odezva tedy trvat dlouho, může dojít k chybě v odpovědi apod.), je vhodné, aby datová vrstva fungovala asynchronním způsobem. To znamená, že v případě kdy jiná komponenta vyvolá

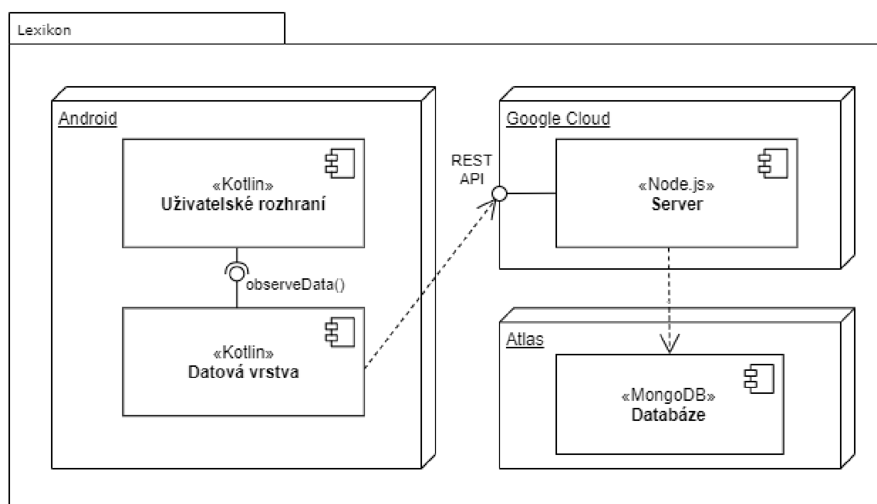
dotaz na datovou vrstvu, není tato komponenta blokována, ale je později informována o výsledku dotazu. Tím je umožněno například zachování responzivity uživatelského rozhraní i ve chvíli, kdy jsou načítána data a nedojde tedy k "zaseknutí" aplikace.

Druhou komponentou je uživatelské rozhraní. Tato komponenta se stará o přímý kontakt s uživatelem – tedy o zobrazování dat, vyhodnocování vstupů (kliknutí na tlačítko, zadání textu apod.). V systému Android je uživatelské rozhraní tvořeno pomocí tzv. Aktivit a Fragmentů. Aplikaci stačí, když bude obsahovat dvě Aktivity. Jedna hlavní, ve které se zobrazují tři různé seznamy zmíněné v sekci 3.1.1. Druhá obsahující detail zvířete. Seznamy lze udělat jako fragmenty, které budou v hlavní Aktivitě přepínány v závislosti na tom, jaká "obrazovka" (míněno ve smyslu ze sekce 3.1.1) je zobrazena. Aktivita detailu zvířete musí obsahovat tři fragmenty – jeden pro každý segment informací o zvířeti (obecné, rožíření, zajímavosti). O aktivitách a fragmentech obecně více v sekci 3.1.4.

Pokud bychom chtěli vytvořit stadardní třívrstvou architekturu, aplikace by musela mít ještě jednu komponentu – tzv. byznys vrstvu. Tato vrstva obvykle obsahuje hlavní funkcionality programů, které třívrstvou architekturu dodržují. Může se jednat o různé výpočty, algoritmy apod. V této aplikaci by ale prostřední byznys vrstva byla velmi tenká, protože z podstaty aplikace a navržených funkcí nevyplývá potřeba větších funkcionalit, které by musely být spravovány přímo v této vrstvě. Funkce jako filtrování nebo vyhledávání mohou být vyřešeny přímo v příslušných Aktivitách a zobrazovaná data není potřeba nějak rozsáhleji transformovat oproti jejich reprezentaci z datové vrstvy.

Z toho vychází architektura o dvou hlavních komponentách – datové vrstvě a vrstvě uživatelského rozhraní. Rozdělení do jednotlivých tříd již z části podléhá použitým technologiím, a proto jsou blíže popsány v kapitole o implementaci (3.2). Z pohledu architektury je sice aplikace rozdělena na dvě vrstvy, ale ve skutečnosti data projdou přes více vrstev. Získání dat ze serverového API je získáváno třídou blokujícím způsobem. Převedení získávání dat na asynchronní způsob je potřeba provést v jiné třídě. Je v podstatě možné tuto třídu označit jako onu prostřední vrstvu, která se stará o komunikaci mezi datovou vrstvou a uživatelským rozhraním a transformuje reprezentaci dat. Pokud ovšem zahrneme asynchronnost získávání dat z datové vrstvy do jejích požadavků, pak tato třída musí spadat do komponenty datové vrstvy.

Celkový náhled architektury zahrnující databázi, server i android aplikaci lze vidět na obrázku 3.5. Funkce `observeData()` na rozhraní mezi komponentami značí celou sadu funkcí (funkce vynechány pro zachování přehlednosti diagramu).



Obrázek 3.5: Architektura a nasazení

3.1.4 Aktivity a Fragmenty

V této části jsou stručně popsány pojmy Aktivita a Fragment v rámci technologie Android.

Aktivita

Aktivita je jednotlivá obrazovka zobrazená na displeji zařízení. Vzhled uživatelského rozhraní je zpravidla definován pomocí jazyka XML. V této aplikaci konkrétně je ovšem uživatelské rozhraní často definováno s pomocí knihovny *Anko*. Ta umožňuje definovat uživatelské rozhraní ve zdrojovém kódu (tím je například umožněno jednoduché přiřazování jednotlivých komponent do třídních proměnných, čímž se eliminuje nutnost vyhledávání těchto komponent pomocí identifikátorů). Při načtení aktivity je toto uživatelské rozhraní zobrazeno.

Aktivita má navíc určitý daný životní cyklus, kdy může procházet různými stavy. S tím jsou spojené funkce jako `onCreate()` (volána při vytvoření), `onPause()` (volána při pozastavení aktivity) nebo `onDestroy()` (volána před zničením aktivity). Tyto funkce lze přetížit za účelem nastavení chování Aktivity v rámci životního cyklu (například uložení dat před zničením apod.).

Fragment

Fragment je samostatná část uživatelského rozhraní nebo chování aktivity. Fragment funguje vždy v rámci nějaké aktivity. Nelze ho tedy zobrazit samostatně bez ní a také s touto aktivitou sdílí svůj životní cyklus (ten je tedy stejný jako u aktivit). Aktivita může obsahovat fragmentů několik nebo i žádný. Fragmenty eliminují duplicitní kód, protože třída definující fragment

může být použita v různý aktivitách. Prvky uživatelského rozhraní mohou být tímto způsobem použity na různých místech v aplikaci, aniž by byly znovu definovány v každé aktivitě, která je využívá.

3.2 Implementace

V této sekci se práce soustřeďuje na detailní popis implementace aplikace pro Android. Implementace samozřejmě v prvé řadě vychází z návrhu architektury, který je popsán v sekci 3.1.3. Podle toho se postupuje i v této sekci. Jelikož byl v předešlých kapitolách popsán server, databáze a data, i tato sekce začíná od implementace datové vrstvy a končí implementací uživatelského rozhraní.

3.2.1 Napojení na server

Jak bylo popsáno v kapitole 2, nasazené API má formu REST. Díky tomu, že je REST tak hojně rozšířený formát, existují knihovny, které výrazně zjednodušují napojení na API v tomto formátu a není tak potřeba implementovat rozsáhlé zasílání požadavku a zpracovávání odpovědí. Jednou z těchto knihoven je *Retrofit*.

Retrofit

Jak je napsáno v dokumentaci knihovny *Retrofit*: *"Retrofit změní vaše HTTP API na rozhraní v Javě".* [29] Knihovna tedy umožňuje vytvoření Java rozhraní, se kterým pak lze jednoduše v kódu pracovat. Toho je docíleno s pomocí atributů, kterými je popsáno rozhraní.

Nejdříve je potřeba použít knihovnu pomocí řádku:

```
import retrofit2.http.*
```

Poté lze vytvořit rozhraní, které je nazváno `ApiDescription` (název může být libovolný). Funkce v tomto rozhraní budou vracet data, která je potřeba přijímat z API do aplikace, a musí být popsána atributem, který knihovně *Retrofit* dá informaci o tom, kde v API jsou potřebná data. Jedna z funkcí rozhraní tedy vypadá následovně:

```
@GET("animals/{id}")
fun getAnimalById(@Path("id") id : Int): Single<Animal>
```

Atribut `@GET` udává informaci, že data o zvířeti lze najít v API na adrese `animals/id`, kde `id` je identifikátor zvířete. Tím, že je identifikátor ve složených závorkách, *Retrofit* dostává informaci, že tato část bude nahrazena některým z parametrů funkce. Parametr, který je využitý k poskytnutí informace je označen atributem `@Path`. Tuto funkci tedy lze v kódu zavolat k získání

zvířete podle jeho identifikátoru. O zaslání požadavku a o jeho zpracování a převedení do třídy `Animal` se postará *Retrofit*.

Obdobné funkce je potřeba nastavit i pro ostatní informace, které aplikace potřebuje získat z API.

Když je rozhraní `ApiDescription` vytvořeno, lze vytvořit instanci pomocí následujícího kódu:

```
Retrofit.Builder()
    .baseUrl(ApiConfig.BASE_URL)
    .build()
    .create(ApiDescription::class.java)
```

`ApiConfig.BASE_URL` je konstantní textová proměnná obsahující webovou adresu serveru. Tento kód vrátí instanci třídy, která implementuje výše zmíněné rozhraní, a lze ji tedy používat k získávání informací. Tato instance je uložena do *Koin* modulu, který ji zpřístupňuje kdekoliv v aplikaci.

Koin

Koin je lehký systém pro vkládání závislostí v jazyce Kotlin. To znamená, že jeho využitím lze spravovat závislosti tříd tak, že není potřeba, aby třídy měly referenci na své závislosti ve chvíli, kdy je program kompilován. [30]

V aplikaci je tento framework využitý převážně pro fungování jednotkových (singleton) tříd. Ty lze vytvořit pomocí Koin modulu tímto způsobem:

```
val remoteModule = module {
    single<ApiInteractor> {
        ApiInteractorImpl(apiDescription = get())
    }

    single {
        Retrofit.Builder()
            .baseUrl(ApiConfig.BASE_URL)
            .build()
            .create(ApiDescription::class.java)
    }
}
```

Slovo `single` zde značí jednotkovou třídu. Jako druhá jednotková třída tam je `ApiDescription` vytvořený pomocí retrofitu (jedná se o kód již zmíněný výše u popisu vytvoření napojení na API). První jednotková třída v tomto úryvku pak rovnou ukazuje využití Koin, kdy stačí zavolat funkci `get()`. Koin zde dosadí tu jednotkovou třídu (ze svých modulů), která odpovídá rozhraní `ApiDescription`.

S Koin lze pak v kódu dynamicky měnit obsah modulů, čímž se mění i závislosti. To lze využít při vytváření jednotkových a instrumentovaných testů (více v kapitole 3.3) pro změnu závislosti z reálného API na testovací API.

ApiInteractor

Další "vrstva" nad napojením na API je rozhraní `ApiInteractor` a jeho implementace. Tato třída již byla zmíněna výše v popisu Koin modulů. V aplikaci se aktuálně jedná o velice jednoduchou třídu a může se zdát být až zbytečnou. Její rozhraní je stejné jako rozhraní třídy `ApiDescription` a její funkce pouze dál předávají volání funkcí. Příklad jedné z funkcí:

```
override fun getLocalities(): Single<List<Locality>>
    = apiDescription.getLocalities()
```

Ostatní funkce mají stejnou formu. Třída je ale důležitá pro vytvoření abstrakce mezi využitým napojením na API a zbytkem aplikace. Například pokud by došlo ke změně API na serveru, tato změna by se nepropagovala přes `ApiDescription` dále do aplikace, protože by se měnila pouze implementace této třídy. Obdobně v případě, že by například napojení na serverové API bylo zaměněno za úplně jiný zdroj dat (například lokální databáze), musela by být pouze vyměněna implementace rozhraní `ApiInteractor`.

Repository třídy

Jedná se o další vrstvu tříd. Tyto třídy se starají o komunikaci s rozhraním `ApiInteractor`, o přesunutí práce na nové vlákno (zajišťují asynchronnnost dotazů, jak bylo zmíněno v kapitole 3.1.3) a mění formu dat na `Observable`. To v doslovném překladu znamená "Sledovatelný", a umožňuje to třídám ve vyšších vrstvách aplikace, aby sledovaly status dat. Díky této formě sledování lze měnit uživatelské rozhraní dynamicky podle toho jak se mění data nebo jak jsou získávána.

Zatímco mají data status "loading", je v aplikaci zobrazeno čekací kolečko. Jakmile se dokončí stahování dat, je sledovatel informován o změně statusu na "loaded" a aplikace může zobrazit data.

Třídy fungujím tímto způsobem:

```
override fun fetchActionData() {
    stateObserver.loading()
    fetchDisposable = apiInteractor.getActions()
        .subscribeOnNewThread()
        .mapException()
        .subscribe({ sampleData ->
            stateObserver.loaded(sampleData)
        }, { e ->
```

```
            stateObserver.error(e)
            Timber.e(e)
        })
}

override fun observeActionData():
    Observable<State<List<Action>>> {
    if (fetchDisposable == null) {
        fetchActionData()
    }
    return stateObserver.observeState()
}
```

Tento úryvek je ze třídy `ActionRepository`, která spravuje data nadcházejících akcí. `stateObserver` a `fetchDisposable` jsou proměnnými v této třídě. Veřejná funkce `observeActionData` funguje tak, že pokud ještě nebyla načtena data, zavolá funkci `fetchActionData`, která nejdříve nastaví status výše zmíněného `Observable` na "loading". Tím značí, že data nejsou načtená, ale že se ještě načítají. Poté se napojí na novém vlákně (`subscribeOnNewThread`) na `ApiInteractor` tím způsobem, že pokud dostane data, změní status `Observable` na "loaded" s těmito daty (čímž značí, že data jsou načtená), ale pokud dostane zprávu o chybě, nastaví status `Observable` na "error" (čímž pošle informaci o chybě dále). Původní funkce pak vrátí odkaz na nastavený `Observable`.

Podobné třídy jsou vytvořeny i pro ostatní druhy dat, jako jsou zvířata, pavilony apod. Všechny tyto třídy jsou opět přidány do Koin modulu.

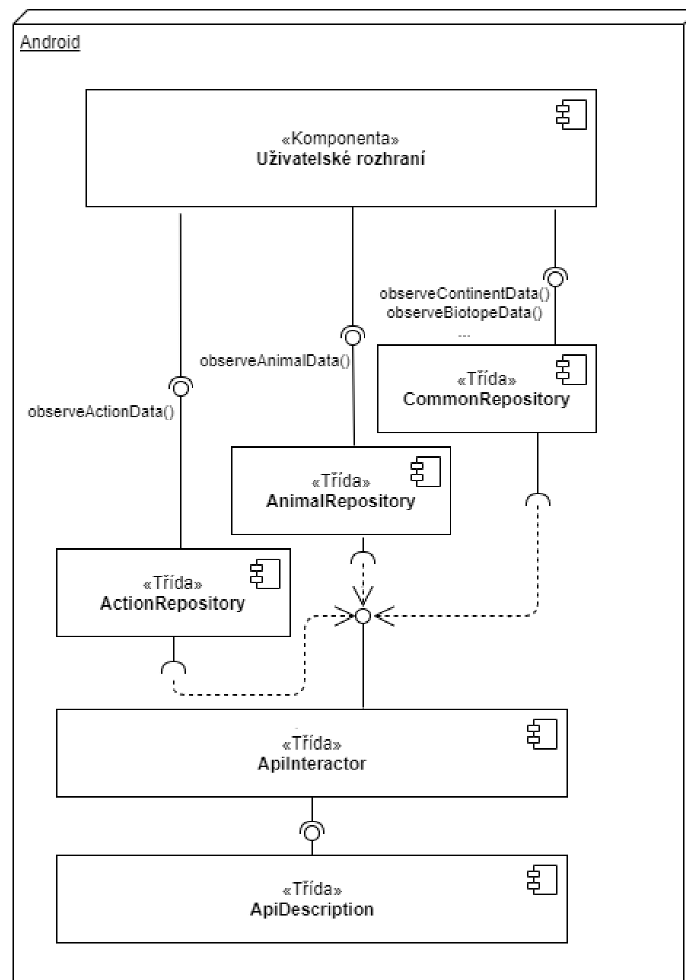
Jednoduchý náhled na srovnání těchto "vrstev" lze vidět na diagramu na obrázku 3.6.

ViewModel třídy

`Repository` třídy jsou sdružovány do několika ViewModel tříd. Tyto ViewModel třídy jsou opět uloženy v Koin modulech (jako `viewModel { ActionViewModel(repository = get())}`) a jsou přímo využívány ve fragmentech ve vrstvě uživatelského rozhraní.

3.2.2 Uživatelské rozhraní

Uživatelské rozhraní se skládá z Aktivit a Fragmentů. Jejich obecná funkce v aplikacích Android je blíže rozebrána v kapitole 3.1.3. V souvislosti s tím, jak bylo konceptuálně navrženo uživatelské rozhraní, stačí v aplikaci pouhé dvě aktivity. `MainActivity`, ve které jsou zobrazeny všechny tři typy seznamů (toho je docíleno různými fragmenty) a `AnimalDetailActivity`, která zobrazuje detailní informace o konkrétním zvířeti.



Obrázek 3.6: Vrstvy v aplikaci

MainActivity

`MainActivity` je vcelku standardní aktivita obsahující toolbar nahoře a pod ním konteiner na fragmenty. Nadpis i menu v pravém horním rohu poskytují jednotlivé fragmenty. Jeden prvek uživatelského rozhraní je ale spravován na úrovni aktivity, a to je navigační menu. Toto menu je zmíněno v kapitole 3.1.1 a vyvolává přechody mezi jednotlivými fragmenty. Toto menu je standardně přidáno do uživatelského rozhraní a k tomu je u něj pomocí metody `setNavigationItemSelectedListener` nastaveno chování v případě stisknutí odkazu na některý ze seznamů:

```
navView.setNavigationItemSelectedListener { menuItem ->
    ...
    when (menuItem.itemId) {
```

```
        R.id.nav_animals -> {
            replaceFragment(MainFragment.newInstance()
                , addToBackStack = false)
        }
        ...
    }
}
```

`when` funguje jako `switch` v jiných jazycích. Podle identifikátoru zvoleného odkazu je funkcí `replaceFragment` změněn aktuální fragment zobrazený v kontejneru. `addToBackStack` určuje, zda se má přechod přidat na zásobník obrazovek a je nastaven na `false`, protože to je zde nežádoucí (přepínáním přes navigační menu by se zvětšoval zásobník a uživatel by pak tlačítkem zpět mohl skákat stále dokola podle toho jak původně přepínal fragmenty).

Fragmenty, stejně jako typy seznamů naplánované v návrhu uživatelského rozhraní, jsou tři. `MainFragment`, `LocalitiesFragment` a `ActionsFragment`.

MainFragment

`MainFragment` je hlavní stránkou v aplikaci. Zobrazuje seznam zvířat a je v aktivitě zobrazený jako první. Fragment ve své definici uživatelského rozhraní obsahuje:

- seznam zvířat,
- definice vzhledu jedné položky seznamu zvířat,
- texty s aktivními filtry,
- tlačítko vyvolávající dialog pro filtrování,
- menu – obsahující pouze jednu položku, jíž je vyhledávání.

Jednotlivé položky v seznamu zvířat jsou definovány s využitím knihovny *Epoxy* (ta je využitá i u ostatních seznamů v jiných fragmentech). To je knihovna určená k vytváření uživatelského rozhraní v `RecyclerView` (jehož je zde využito pro zobrazení seznamu). [31] Definice položky je tedy samostatná třída, nazvaná `AnimalEpoxyModel`, která dědí z třídy `EpoxyModelWithLayout`. Má několik veřejných proměnných, které jsou označeny atributem `@EpoxyAttribute` – tyto proměnné jsou informace, které má každá položka držet. Při vytvoření seznamu jsou tyto informace nastaveny a Epoxy pak zavolá metodu `bind`. Ta je přetížená a je v ní nastaveno, co se má provést. Zde to je následující:

```
override fun AnimalLayout.bind() {
    titleText.text = title
    latinTitleText.text = latinTitle
    locationText.text = location
    animalIdLayout = animalId
    ...
}
```

`title`, `latinTitle` a další jsou výše zmíněné proměnné označené atributem, které jsou nastaveny ve fragmentu. `titleText`, `latinTitleText` a další jsou prvky uživatelského rozhraní, které je definované v třídě `AnimalLayout`. Ta je přidružená třídě `AnimalEpoxyModel` a obsahuje definici uživatelského rozhraní položky – konkrétní textová pole, relativní rozložení apod.

Informace o zvířatech jsou do této třídy nastaveny ve fragmentu v metodě `addAnimals` takto:

```
layout.epoxyRecyclerView.buildModelsWith { controller ->
        with(controller) {
            orderedAnimals?.forEach {
                animal {
                    id(it._id)
                    title(it.title)
                    latinTitle(it.latin_title)
                    location(it.localities_title)
                    imgURL(it.image_src)
                    animalId(it._id)
                }}}}
```

Jednotlivé řádky v bloku pro `animal` značí jednotlivé epoxy atributy – v závorce jsou pak předána data pro jedno každé zvíře.

Když je zobrazený seznam zvířat, uživatel v něm může vyhledávat. Textové pole pro vyhledávání je nejprve skryté. K jeho zobrazení dojde ve chvíli, kdy uživatel stiskne tlačítko pro vyhledávání. Vyhledávání je napojeno na textové pole následovně:

```
layout.searchBar.textChanges()
    .debounce(500, TimeUnit.MILLISECONDS )
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe{ search(it.toString()) }.addTo(disposables)
```

Tento úryvek znamená, že pokud se text v textovém poli půl vteřiny nezmění, je zavolána funkce `search`. Uživatel tedy nemusí aktivně stisknout pokyn pro vyhledání, ale vyhledávání probíhá automaticky ihned. Uživatel tak může najít hledané zvíře i dřív, než dopíše co zamýšlel. Funkce `search` pouze přenastaví proměnnou `filterString` a zavolá funkci `refilter`.

Funkce `refilter` je společná funkce, která aplikuje nejenom text zadaný do vyhledávacího pole, ale i ostatní filtrovací kritéria. Poté co je aplikuje, zavolá výše popsanou funkci `addAnimals`, která přidá do seznamu filtrovaná zvířata. Tím, že je funkce společná, je zajištěno, že vyhledávání probíhá v rámci aktivních filtrů. Zároveň je znovupoužitelná a je volána i v případě stisknutí tlačítka pro použití filtrů nebo pro obnovení filtrů v dialogu pro filtrování.

V těchto dvou případech je volána i funkce `redrawFilters()`. Jedná se o část aplikace, která byla přidána až po uživatelském testování (více v kapitole 3.3), která se stará o zobrazení aktivních filtrů tak, že v případě, že je nějaký filtr aktivní, je příslušný text zviditelněn.

Ve výše zmíněné třídě `AnimalEpoxyModel`, v níž je nastaveno uživatelské rozhraní pro jednotlivé položky v seznamu, je nastaveno i jejich chování. Tedy to, že při klepnutí na tuto položku je zavolána funkce:

```
view.setOnClickListener {
    val intent = AnimalDetailActivity
        .getIntent( context, animalIdLayout )
    context.startActivity( intent )
}
```

Tímto způsobem se v Androidu spouští nové aktivity. K jejich spuštění je potřeba instance třídy `Intent`. Její vytvoření zde obstarává statická funkce `getIntent` třídy `AnimalDetailActivity` (aktivita, která má být spuštěna). Forma vytváření instance třídy `Intent`, pomocí statické funkce u aktivity, která má být spuštěna je standardní postup ve vývoji pro Android. Tato třída obsahuje informace o tom, jaká aktivita má být spuštěna a s jakými úvodními daty. Protože musí být obecná pro jakoukoliv aktivitu v jakékoliv aplikaci, data jsou zde identifikována podle klíčů ve formě textu. Protože by takto vznikaly závislosti napříč kódem týkající se těchto klíčů a nastavování dat k nim, je standardem, aby `Intent` byl vytvářen ve funkci aktivity. Tím je závislost na klíčích skrytá pouze do této třídy.

Zde je do `Intent` přidán identifikátor zvířete. Tím je aktivitě předána informace, o jakém zvířeti mají být zobrazeny detaily. `Intent` je následně předán do funkce `startActivity`, čímž je aktivita spuštěna.

LocalitiesFragment

Fragment zobrazující seznam pavilonů v ZOO. Samotný fragment nemá tolik funkcí, jako fragment zobrazující seznam zvířat, protože zde není potřeba žádné filtrování ani vyhledávání. Je zde využita jiná třída pro vytvoření uživatelského rozhraní jednotlivých položek v seznamu. Tou je `LocalityEpoxyModel` a `LocalityLayout`. Třídy jsou obdobné jako třídy pro položky v seznamu zvířat – opět je využit framework Epoxy (atributy atd.).

Jednotlivé položky obsahují dvě tlačítka (ty byly zavedeny až po testování, viz kapitola 3.3). Prvním je tlačítko “Zvířata”, které funguje tímto způsobem:

```
but.setOnClickListener{
    (context as MainActivity).replaceFragment
        (MainFragment.newInstance(
            titleText.text.toString()), addToBackStack = true)
}
```

Na aktivitě, která vlastní tento fragment (zde je to aktivita `MainActivity`), je změněn aktivní fragment na již zmíněný fragment `MainFragment`. Ten je ovšem vytvořen s aktivním filtrováním podle pavilonu. Znovupoužitím fragmentu, který je již využitý na úvodní obrazovce, je zajištěno zobrazení zvířat v pavilonu, aniž by docházelo k duplikaci kódu. Zároveň je tento fragment přidán na zásobník, takže při stisknutí tlačítka zpět se uživatel navrátí k fragmentu zobrazujícímu pavilony.

Druhým tlačítkem je tlačítko "WWW". To funguje následovně:

```
wwwBut.setOnClickListener {
    val openURL = Intent(Intent.ACTION_VIEW)
    openURL.data = Uri.parse(urlLayout)
    context.startActivity(openURL)
}
```

Při stisknutí se opět vytvoří instance třídy `Intent`. Tentokrát není instance vytvořena pomocí statické metody, protože je potřeba otevřít webový odkaz a ten nebude otevřen v žádné aktivitě v této aplikaci. `ACTION_VIEW` popisuje typ akce, která se má na datech vykonat – v tomto případě prohlížení. Do dat instance se vloží webový odkaz a nakonec je spuštěna aktivita s tímto `Intent`em. Tím se provede akce otevření webového prohlížeče na tomto odkazu. Jak je zmíněno v kapitole 1.1, jedná se o odkaz na webové stránky s dalšími informacemi o tomto pavilonu.

AnimalDetailActivity

Aktivita `AnimalDetailActivity` slouží k zobrazení detailních informací o konkrétním zvířeti. Aktivita je spouštěna ze seznamu zvířat (`MainFragment`), který jí při spuštění zašle identifikátor zvířete, jež má být zobrazeno.

Aktivita obsahuje tři fragmenty, které jsou přidány do `FragmentPagerAdapter`. Tato třída je používána k obsluhování přepínání mezi fragmenty v záložkovém rozložení. Aktivita totiž obsahuje tři záložky (jak bylo navrženo v kapitole 3.1.1) a mezi těmito záložkami se přepíná přetažením do strany nebo stisknutím tlačítka pro záložku. `FragmentPagerAdapter` tedy obsahuje fragmenty korespondující s jednotlivými záložkami.

Fragmenty obsažené ve `FragmentPagerAdapter` jsou `BasicInfoFragment`, `SpreadFragment` a `InterestingFragment`. Všechny tyto fragmenty využívají napojení na `AnimalDetailViewModel`, který je uložený v Koin modulu. Tento

ViewModel obsahuje funkci `observeState`, která má jeden parametr – identifikátor zvířete a vrací `Observable` s tímto zvířetem (napojený na zmiňované `Repository` třídy).

Tento `Observable` je při vytvoření třídy sledován a v závislosti na stavu dat je na něj mapováno chování fragmentu. To je provedeno následovně:

```
viewModel.observeState(animalId)
    .observeOnMainThread()
    .subscribe { state ->
        when (state) {
                ...
            is State.Loaded -> {
                showProgress(false)
                showEmpty(false)
                showError(false)
                fillContent(state.data)
            }
        }
    }
```

V úryvku je třemi tečkami vyznačen vynechaný kus kódu, jímž je zde mapování na ostatní stavy. To je totiž podobné pro všechny stavy, s tím rozdílem, že například pro stav `State.Error` je `showError` voláno s hodnotou `true`, aby byla v uživatelském rozhraní zobrazena chybová hláška. Obdobně pro ostatní stavy.

V případě tedy, že je stav `State.Loaded`, je zneviditelněno zobrazení chyby i načítání a je zavolána funkce `fillContent`. Ta nastavuje do jednotlivých komponent uživatelského rozhraní data o zvířeti, která byla získána ze serveru. Toto se děje ve všech třech fragmentech s tím rozdílem, že každý zobrazuje sobě specifická data a má jiné rozložení komponent v uživatelském rozhraní.

3.3 Testování

Pod pojmem testování je zahrnuto více různých činností. Jednou z těchto činností jsou uživatelské testy, které se zabývají nejen chybami v programu, ale také nevhodným návrhem uživatelského rozhraní. Dokonce mohou být zaměřené pouze na vyzkoušení uživatelského rozhraní, pokud nejsou prováděny na hotovém programu, ale na navrženém rozhraní (to může být buď tzv. wireframe nebo přímo prototyp rozhraní, vytvořený jako normální rozhraní, ale bez fungující datové části aplikace). Dále existují automatické testy, které jsou prováděny strojově. Ty jsou vždy zaměřeny na to, jestli aplikace funguje správně a neobsahuje chyby. Android umožňuje tvorbu dvou typů automatizovaných testů, jimiž jsou Unit (jednotkové) testy a Instrumen-

tované testy. Bližší popis těchto dvou typů lze nalézt v příslušných sekcích této kapitoly.

3.3.1 Automatické testování

Automatické testy lze provádět jako Unit testy nebo jako Instrumentované testy. Unit, neboli jednotkové testy je všobecný typ testů který zkouší správnou funkcionalitu jedné konkrétní třídy (od toho název Unit – jednotka). Zkouší, zda se instance třídy chová správně v určitém simulovaném kontextu. Ten se tvoří pomocí tzv. mockování (doslovně lze přeložit jako "napodobení") – to znamená, že se vytvoří napodobeniny ostatních tříd, které se chovají předem daným způsobem. Stačí udělat napodobeniny těch tříd, které přímo komunikují s testovanou třídou. Díky tomu, že je známo, jak se chovají napodobené třídy, lze oveřit, že se v tomto kontextu chová testovaná třída tak, jak je očekáváno.

Například, existuje-li třída `Person`, která obsahuje jméno a věk člověka a je potřeba otestovat třídu `PersonSelector`, která má funkci na zvolení staršího člověka ze dvou zadaných a má vrátit jméno tohoto člověka. V Unit testu se nejdříve vytvoří napodobeniny dvou instancí třídy `Person`, které mají konkrétní jména a konkrétní věk. S těmito instancemi se pak zavolá zmíněná funkce. Poté dojde k ověření skutečnosti, že jméno, které tato funkce vrátila, je jménem staršího ze dvou napodobených lidí.

V praxi nelze vytvořit Unit testy ke každé třídě. Některé třídy mají velmi vysokou provázanost s jinými třídami. To znamená, že pro jejich otestování by muselo být napodobeno velké množství ostatních tříd a samotné testování by pak bylo mnohem rozsáhlejší než testovaná třída. Úsilí potřebné pro tvorbu takového testu pak převáží jeho možné benefity, a tedy vytvoření takového testu je nežádoucí. Je vždy nutné mít na paměti, že i vývoj programů se řídí návratností. A pokud čas strávený na určité činnosti nemá dostatečnou návratnost, pak je lepší se této činnosti vyhnout.

Vysoká provázanost může být způsobená špatným návrhem modelu tříd, ale také přímo systémem, ve kterém je aplikace tvořena. V tomto případě to jsou právě Aktivity a Fragmenty, zmíněné v kapitole 3.1.3, které není možné jen tak instancovat, protože pro svojí správnou funkci vyžadují velké množství kontextu. Pro testování těchto tříd jsou vhodné právě Instrumentované testy.

Unit testy jsou napsány ke třídě `ApiInteractor` a ke třídám `Repository` (těch je víc, například `AnimalRepository` nebo `ActionsRepository`). Tyto třídy mají nízkou provázanost s dalšími třídami a jsou tedy vhodné pro Unit testy.

Pro vytvoření Unit testů lze využít framework *JUnit*. Jedná se o externí knihovnu zaměřenou právě na jednoduchou tvorbu Unit testů a je tou nejpoužívanější externí knihovnou v jazyce Java (vyskytuje se ve 30 % projektů na serveru GitHub, které jsou v jazyce Java), což z ní dělá zároveň nejpoužívanější framework pro tvorbu Unit testů v Javě. [32] Aplikace je sice psaná v jazyce

Kotlin, ne v jazyce Java, ale jak bylo zmíněno v kapitole 3.1.3, Kotlin je plně interoperabilní s jazykem Java a není tedy problém využít knihovnu pro Javu. Testy jako takové mohou dokonce být stále psané v jazyce Kotlin.

Framework JUnit umožňuje jednoduché tvoření testů pomocí atributů. S nimi se v testovací třídě označí jednotlivé funkce podle jejich typu – pokud má tedy funkce být testem, označí se atributem `@Test`. Takto označené funkce jsou pak v rámci jednoho testování postupně volány. Jsou také nabízeny atributy `@Before` a `@After` – první označuje funkce, které mají být volány před každým testem, druhý označuje funkce, které mají být volány po každém testu. Obdobně i atributy `@BeforeClass` a `@AfterClass`, které značí funkce, jež mají být zavolány jednou před všemi testy a jednou po všech testech. Tyto atributy jsou využity, pokud je potřeba provést nějaká nastavení před testováním, což může být například vytvoření napodobenin tříd, které mohou být využity ve všech testech v konkrétní testovací třídě.

K vytvoření napodobenin tříd (neboli "mocků") lze využít framework *Mockito*. Jedná se opět o externí knihovnu. Je nejpoužívanější knihovnou specializovanou na vytváření napodobenin tříd v jazyce Java (vyskytuje se ve 4 % projektů na serveru GitHub, které jsou v jazyce Java, což je nejvíc z knihoven s touto funkcí). [32]

Pro vytvoření napodobeniny třídy je ve frameworku *Mockito* poskytována funkce `Mockito.mock`. Jejím parametrem je třída, která má být vytvořena. Poté lze nastavovat chování napodobeniny pomocí funkcí `when` a `thenReturn` – v překladu tedy "pokud" a "pak vrať". "Pokud" je zavolána nějaká funkce této třídy, "pak vrať" tuto odpověď. V tomto stylu lze vytvořit specifické chování napodobeniny, kterému pak musí odpovídat správně testovaná třída.

Frameworky *JUnit* a *Mockito* je nejprve nutné importovat. K tomu jsou potřeba tyto dva řádky:

```
import org.junit.*
import org.mockito.Mockito
```

Nyní lze využívat atributy funkcí a funkce Mockita. Unit testy je vhodné rozdělit do více testovacích tříd, kdy jedna testovací třída přísluší jedné třídě v aplikaci. Jelikož jsou `Repository` třídy velmi podobné, lze jejich testy pro přehlednost vložit do stejné třídy. Tímto vzniknou dvě testovací třídy: `ApiInteractorTest`, která testuje třídu `ApiInteractor`, a `RepositoryTest`, která testuje třídy `CommonRepository`, `AnimalRepository` a `ActionRepository`.

Jedním z testů v testovací třídě `ApiInteractorTest` je například test, který ověřuje, že třída `ApiInteractor` správně vrací instance potravy. Tímto testem se ověřuje, že navrácená instance nemá změněná data. Test vypadá takto:

```
@Test
fun ApiInteractorReturnsFoods(){
```

```
    val apiDescription: ApiDescription =
        Mockito.mock(ApiDescription::class.java)
    val food = Food(1, 1, "foodTest")

    Mockito.`when`(apiDescription.getFoods())
        .thenReturn(Single.just(listOf(food)))

    val interactor = ApiInteractorImpl(apiDescription)

    assertEquals(interactor.getFoods().blockingGet(),
        listOf(food))
}
```

Funkce tedy začíná atributem `@Test`, který ji označuje jako test. Jako první je v testu vytvořena napodobenina třídy `ApiDescription` pomocí volání `Mockito.mock`. To je jediná třída, se kterou komunikuje testovaná třída, a je to tedy jediná potřebná napodobenina. Následně je vytvořena jedna instance potravy s nějakými daty (na přesných datech nezáleží, protože se testuje pouze to, zda nebudou změněna). U napodobeniny je pak nastaveno, že pokud na této instanci bude zavolána metoda `getFoods()`, má instance vrátit seznam potrav o jednom prvku – o výše vytvořené instanci potravy. To je nastaveno pomocí metody `Mockito.when`. Ve chvíli, kdy je tento kontext připraven, je vytvořena instance testované třídy `ApiInteractorImpl` a do jejího konstruktoru je předána předpřipravená napodobenina. Na závěr probíhá otestování, zda testovaná třída při zavolání metody `getFoods()` vrátí seznam obsahující stejnou potravu, jaká je vrácena z `ApiDescription`.

Může se zdát, že je testováno, zda jsou vráceny stejné instance. Pro třídu `ApiInteractor` je ale v pořádku, i pokud vrátí jinou instanci se stejnými daty. Díky funkci jazyka Kotlin je ale testována rovnost dat ve třídě `Food`, protože se jedná o datovou třídu, jejíž funkcí je právě to, že není potřeba přetěžovat operátor rovnosti ručně, ale je přetížen automaticky tak, aby srovnával data instance.

Jak je uvedeno výše, druhou testovací třídou je třída `RepositoryTest`, která obsahuje testy týkající se několika tříd `Repository`. Příkladem jednoho z testů je následující funkce:

```
@Test
fun RepositoryGetsClasses(){
    val apiDescription: ApiDescription =
        Mockito.mock(ApiDescription::class.java)
    val cls = AnimalClass(1, 1, "typeTest", 2,
        "titleTest", "aliasTest")

    Mockito.`when`(apiDescription.getClasses())
```

```
        .thenReturn(Single.just(listOf(cls)))

    val interactor = ApiInteractorImpl(apiDescription)

    val repo = CommonRepositoryImpl(interactor)

    val state0 = repo.observeClassData().blockingFirst()
    Thread.sleep(500)
    val state = repo.observeClassData().blockingFirst()

    TestCase.assertTrue(state is State.Loaded)
    TestCase.assertEquals((state as State.Loaded).data,
        listOf(cls))
}
```

Funkce je podobná předchozímu příkladu. Opět je vytvořena napodobenina třídy `ApiDescription`. Poté je na této napodobenině nastaveno navrácení seznamu o jedné předpřipravené instanci v případě zavolání funkce `getClasses()`. To vše s pomocí frameworku *Mockito*. V dalším kroku je vytvořena instance třídy `ApiInteractorImpl` s touto napodobeninou v konstruktoru a poté je tato instance použita do konstruktoru třídy `CommonRepositoryImpl` (třída testovaná v této funkci), čímž je vytvořen kontext testu. Třídy `Repository` se starají o převedení z blokujícího volání na neblokující a o vracení stavů právě sledovaných dat. V testu je tedy přidáno čekání 0,5 vteřiny, protože před čekáním stav dat ještě nemusí být "Loaded" (načtený). Po čekání jsou získána data a je zkoušeno, zda současný stav dat odpovídá stavu "načtený" a zda jsou data shodná s daty, která jsou poskytována napodobeninou třídy `ApiDescription`.

Instrumentované testy

Některé funkce aplikace lze nejlépe otestovat s pomocí tzv. Instrumentovaných testů. V prostředí Androidu to znamená speciální druh testů, který běží přímo v prostředí Androidu buď na emulovaném přístroji nebo na reálném přístroji s operačním systémem Android. Tyto testy je vhodné využít tam, kde se testování hodně zjednoduší díky tomu, že je v testech možné využít prostředí Androidu a dostupných knihoven, které fungují na běžícím přístroji. Příkladem mohou být například třídy využívající `Context`. To je třída popisující aktuální kontext, ve kterém daná část aplikace běží a je tedy obtížné ji vhodně napodobit. Mezi tyto třídy se řadí právě `Activity` a `Fragment`, kde se nachází většina kódu v této aplikaci, pročež je většina testů instrumentovaných.

Výhoda instrumentovaných testů – že běží na zapnutém přístroji – je zároveň jejich nevýhodou, protože kvůli tomu jsou mnohonásobně pomalejší

než obyčejné Unit testy. Proto by se měly instrumentované testy využívat tam, kde je to vhodné a v ostatních částech programů jsou vhodnější Unit testy.

Pro tvorbu instrumentovaných testů lze využít opět framework *JUnit*, s jehož pomocí se v testovací třídě použitím atributů označí jednotlivé funkce jako testy či jako funkce provádějící nastavení před testy a po testech. Jelikož se ale nejedná již o obyčejné Unit testy, jsou využity i další funkce a knihovny.

První z nich je možnost využití generické třídy `AndroidTestRule` z balíčku `android.support.test.rule`. Tato třída umožňuje testování aktivity tím, že danou aktivitu před každým testem spustí a po testu pak zase vypne. Během testu je tak spuštěná aktivita zapnutá a lze na ní provádět testování. [33] Lze tuto třídu využít pro testování jakékoliv třídy, která rozšiřuje třídu `Activity`. K jejímu použití je přidán do testovací třídy následující řádek:

```
@get:Rule
var activityTestRule = ActivityTestRule<MainActivity>
    (MainActivity::class.java, false, false)
```

Jedná se tedy o proměnnou testovací třídy, která je označena atributem `@Rule`, označujícím tento typ třídy. V tomto případě je použita k testování aktivity `MainActivity` (díky předání odkazu na `MainActivity` do konstruktoru). Proměnná je poté využita v testovacích funkcích.

Narozdíl od Unit testů je v této aplikaci u instrumentovaných testů využitý i atribut `@Before`. Ve funkci označené tímto atributem jsou provedena dvě nastavení. První je nastavení napodobeniny API – to kvůli nezávislosti testovacího prostředí na aktuálním stavu serveru. Je nutné, aby testy probíhaly vždy stejným způsobem a aby jejich výsledek nezávisel na tom, zda je server dostupný a zda vrací určitá data. Při změně dat by totiž pravděpodobně došlo k tomu, že nahlásí chybu, přestože v aplikaci je všechno v pořádku. Proto jsou pro účely testování vytvořeny dvě falešné implementace rozhraní `ApiDescription`. Jedna je `ApiDescriptionTestingOK`, která vrací konkrétní data, využitelná k testování jako kdyby server fungoval. Druhou je `ApiDescriptionTestingFail`, ta se chová stejně jako v případě, že server je nedostupný (vrácením výjimky s kódem 404).

Druhým nastavením je přepnutí na konkrétní stránku skrz menu. Protože v aktivitě `MainActivity` je několik možných fragmentů s různými seznamy, je testování těchto fragmentů rozděleno do několika testovacích tříd. Každá třída ve svém `@Before` nastavení přejde z úvodního fragmentu do toho fragmentu, který je testován.

Funkce s atributem `@Before` vypadá v testovací třídě `ActionsActivityInstTest` následovně:

```
@Before
fun setUp() {
    val apiDescription: ApiDescription =
```

```
        ApiDescriptionTestingOK()

    StandAloneContext.loadKoinModules(listOf(module
        { single<ApiInteractor>(override = true)
        { ApiInteractorImpl(apiDescription) } }))

    activityTestRule.launchActivity(Intent())

    onView(withId(R.id.drawer_layout))
        .check(matches(isClosed(Gravity.LEFT)))
        .perform(open())

    onView(withId(R.id.nav_view))
        .perform(navigateTo(R.id.nav_actions))
}
```

Jak je zmíneno výše, nejprve se připraví napodobenina API. Ta tentokrát není vytvořena pomocí frameworku *Mockito*, ale přímo jako normální třída, která implementuje rozhraní `ApiDescription`, protože stejná napodobenina je využita ve více testovacích třídách (tím je snížena duplicita kódu). Následně je v aplikaci nahrazen *ApiInteractor* novou verzí, která využívá tuto testovací verzi. Nahrazení je provedeno přepsáním `koin` modulu (viz kapitola 3.2).

Poté je spuštěna testovaná aktivita, využitím metody `launchActivity()` na výše popsané proměnné `activityTestRule` simulující chování aktivity. Spuštění aktivity by normálně probíhalo automaticky před začátkem každého testu. To zde bylo ovšem vypnuto, protože před zapnutím aktivity je potřeba provést modifikace `koin` modulu, a proto musí být aktivita spuštěna ručně.

Protože se jedná o třídu `ActionsActivityInstTest`, která je zaměřena na testování obrazovky obsahující seznam nadcházejících akcí, je následně proveden přechod na tuto obrazovku. K tomu je využit framework espresso.

Espresso je knihovna pro Android vytvořená pro účely testování na uživatelském rozhraní, k čemuž jsou nabízeny tři hlavní komponenty knihovny: [34]

- *ViewMatchers* – tato část umožňuje vyhledávání konkrétního `View` ve ViewTree (všechny komponenty uživatelského rozhraní jsou uspořádány do stromové struktury – jde tedy o vyhledání konkrétní komponenty v tomto stromě).

- *ViewActions* – umožňuje provedení libovolné akce na konkrétní komponentě uživatelského rozhraní (komponenta nalezená pomocí *ViewMatcher*). Například kliknutí, posunutí apod.

- *ViewAssertions* – umožňuje ověření toho, že je v uživatelském rozhraní zobrazeno to, co podle testů má být zobrazeno.

Přechod na obrazovku akcí je tedy proveden dvěma kroky. Prvním krokem je otevření navigačního menu, jež se skládá z nalezení komponenty s identifikátorem `R.id.drawer_layout` (identifikátor navigačního menu, konkrétněji posouvací komponenty, která navigační menu obsahuje), z ověření, že komponenta je zavřená a z jejího následného otevření. Druhý krok je nalezení komponenty s identifikátorem `R.id.nav_view` (navigační menu) a rozkliknutí seznamu akcí.

V ukázce kódu funkce jsou použity konkrétní identifikátory, z čehož plyne, že k testování s použitím frameworku Espresso je potřeba znát kód programu. Espresso umožňuje i tzv. black box testing (testování bez znalosti kódu programu), což zde ale není potřeba využít.

Poté, co proběhne výše zmíněná funkce s atributem `@Before`, spustí se testovací funkce. Cílem jednotlivých testovacích funkcí je, zda se na uživatelském rozhraní zobrazí správné komponenty se správným obsahem v různých případech chování uživatele. Příkladem jedné testovací funkce z testovací třídy `ActionsActivityInstTest` je tato funkce:

```
@Test
fun testActionDisplayedInList(){
    onView(withText("summary")).check(matches(isDisplayed()))
    onView(withText("desc")).check(matches(isDisplayed()))
    activityTestRule.finishActivity()
}
```

Dobrým pravidlem u testovacích funkcí je popisný název, ze kterého vyplývá, co je funkcí testováno. Zde je možné přeložení názvu funkce jako "testAkceZobrazenáVSeznamu". Funkce tedy vyzkouší, zda je akce zobrazena v seznamu akcí. První řádek najde komponentu uživatelského rozhraní s textem "summary" a ověří, zda taková komponenta je zobrazena. To samé je provedeno pro text "desc". Oba tyto texty jsou konkrétně zvolené vzhledem k napodobenině API, která byla nastavena na začátku testování. Ta totiž implementuje funkci pro získání seznamu akcí tímto způsobem:

```
override fun getActions(): Single<List<Action>> {
    return Single.just(listOf(Action("2019-09-29 10:00:00",
        "2019-09-29 12:00:00", "summary", "desc")))
}
```

V seznamu je vrácena akce obsahující shrnutí s textem "summary" a popis s textem "desc". V testu tedy probíhá ověření, zda právě tyto hodnoty jsou zobrazeny. Obdobně v testech, kde je jako napodobenina API využita třída `ApiDescriptionTestingFail`, se testuje, zda je na obrazovce zobrazeno upozornění o nedostupnosti serveru.

Testovacích tříd s instrumentovanými testy je 5 a pokrývají většinu funkcionalit aplikace.

3.3.2 Uživatelské testování

Uživatelské testování je ideálně prováděno lidmi, kteří se neúčastní vývoje aplikace a s uživatelským rozhraním nejsou předem seznámeni. Tím, že člověk který testuje aplikaci (dále "tester") není zúčastněný na vývoji, se předchází subjektivitě testování, kdy by vývojář mohl podvědomě bránit svůj výtvor a ignorovat některé chyby. Navíc, jelikož tester není předem seznámený s uživatelským rozhraním, zamezuje se neobjektivitě testování způsobené tím, že je tester již na některé procesy aplikace zvyklý. Pokud tomu tak je, dojde k tomu, že tester s aplikací pracuje efektivně, přestože je uživatelské rozhraní špatně navržené.

Testování v této práci proběhlo až po implementaci aplikace. Testy probíhaly na funkční, nevydané verzi, nikoliv na finální verzi. Oproti testům zmíněným na začátku kapitoly 3.3 (testování na wireframe, před započetím implementace) má tato forma určitou výhodu v tom, že pro uživatele je jednodušší provádět činnosti na již implementované aplikaci a při procházení testovacích scénářů musejí vykonávat i všechny detaily, které nemusejí být zachyceny ve wireframe formě uživatelského rozhraní. A některé problémy odkryté v tomto testování by opravdu nebyly nalezeny v testování na wireframe.

Vytvoření scénářů

Pro uživatelské testy je potřeba nejdříve připravit určité scénáře, které v rámci testu tester plní. Jelikož se jedná o testy zaměřené na správný návrh rozhraní a na schopnost uživatelů orientovat se, neměly by být scénáře příliš popisné. Tedy neměly by testerovi dávat konkrétní instrukce jako "klikněte vpravo dole na kulatou ikonu" apod. Vhodnější jsou scénáře připravené formou určitého cíle, který má tester splnit a v rámci testování se sleduje, zda má tester nějaký problém dostat se k tomuto cíli, či jakým postupem se k cíli zkouší dostat.

Zároveň je třeba navrhnout scénáře tak, aby pokrývaly co nejvíce případů užití a hlavně ty, které jsou nejpravděpodobnější při užívání aplikace. S použitím těchto pravidel byly vytvořeny následující scénáře pro testování:

- Najdi informace o Lvu indickém.
- Nech si zobrazit všechna zvířata z Evropy.
- Jak se rozmnožuje Dikdik Kirkův?
- Kde se vyskytuje Dikdik Kirkův?
- Kde v ZOO lze najít Lva Indického?
- Najdi zajímavosti o chovu Lva Indického v ZOO Praha.
- Nechej si zobrazit zvířata z pavilonu Goril.
- Dostaň se na webové stránky o pavilonu Goril.
- Jaká je nejdřívější nadcházející akce v ZOO?

Tyto scénáře pokrývají hlavní funkcionality aplikace a jsou formulovány jako úkoly, které má tester splnit. V průběhu testů jsou testerovi pouze zadávány tyto úkoly ke splnění, ale při jejich plnění mu není poskytována pomoc ve formě nápovědy, kudy se k nějaké funkci dostat. Pokud tester není schopen nějaký z úkolů splnit, může říct, že neví jak na to. V tom případě se přistoupí k dalšímu scénáři (nesplnění je samozřejmě evidováno, protože poukazuje na problém s uživatelským rozhraním). Po splnění všech scénářů je tester ještě dotázán na problémy, které měl se splněním úkolů a případně na další věci, které mu připadají problematické, přičemž na ně při testování nedošlo.

Tímto je připravený plán pro jednotlivé testy a lze přikročit k průběhu testů.

Průběh testování

Testování proběhlo odděleně se čtyřmi testery. Probíhalo na notebooku v emulátoru telefonu s nainstalovanou současnou verzí aplikace. Průběh testů byl zaznamenán pomocí open-source programu *OBS*, který umožňuje uložení záznamu dění na obrazovce ve formě videa společně i s uložením zvuků snímaných mikrofonem. Záznamy všech testů jsou přílohami této práce.

Je možné, že více testů by odhalilo další problémy uživatelského rozhraní a neexistuje žádné přesné pravidlo pro správný počet uživatelských testů. Dobrým vodítkem je, zda se při testech objevují nové problémy, nebo zda se opakují problémy, které již byly nalezené. U čtvrtého testu již byla většina problémů s uživatelským rozhraním shodných s těmi v předešlých testech, a proto byl tento počet považován za dostatečný.

Během testů byly objeveny následující problémy s uživatelským rozhraním:

- Položky v seznamu pavilonů mají po straně ikonu zeměkoule. Ikona vede na webové stránky, přičemž zbylá oblast vede na seznam zvířat v daném pavilonu. Problémem je, že tato ikona sama nevzbuzuje dojem, že vede jinam, než zbylá oblast. Tester byl zmatený, než to zjistil, protože ho nenapadlo stisknout ikonu, když hledal webové stránky pavilonu. Jeden z testerů kvůli tomu dokonce scénář vzdal.

- Při vyhledávání Lva Indického došlo k tomu, že tester zadal do vyhledávání konkrétně “lev indicky”. Podle tohoto textu ale nedokázala aplikace Lva Indického najít, kvůli dlouhému “ý” na konci (toto je právě jeden z problémů, který by nebyl odhalen ve wireframe verzi).

- Při zapnutém vyhledávání se tester pokusil zrušit vyhledávání pomocí stisknutí Android tlačítka zpět. Tlačítko ovšem zavřelo aplikaci. Problém se vyskytnul ve třech různých testech.

- Tester měl problém najít, kde v zoo se vyskytuje Lev Indický. Problém byl způsobený nevhodnou barvou písma ve spojení s fotografií na pozadí. Fotografie byla převážně bílá a světlé písmo na ní nebylo čitelné.

- Protože všechny tři seznamy jsou na stejné úrovni v hierarchii obrazovek, při stisknutí Android tlačítka zpět v jakémkoliv z těchto seznamů dojde k zavření aplikace. Tester se tímto tlačítkem pokoušel dostat ze seznamu pavilonů do seznamu zvířat.

- Dialog s filtrováním je problematický kvůli nejasnosti dolních tlačítek. Tlačítka "zrušit" a "obnovit" mají nejasnou funkci na první pohled.

- Při stisknutí tlačítka pro obnovení filtrů nedojde automaticky k jeho použití. Je tedy potřeba stisknout "obnovit" a poté stisknout "použít".

- Tester si neuvědomoval, že filtrování je aktivní i při vyhledávání. Zatímco měl aktivní filtrování podle kontinentu Evropa, snažil se textově vyhledat Dikdika Kirkova a divil se, že nebyl nalezen. Dikdik Kirkův se totiž nachází v Africe a aplikace v tu chvíli vyhledávala pouze mezi zvířaty z Evropy.

Upravení aplikace na základě testování

Dle nalezených problémů musely být provedeny určité úpravy v aplikaci tak, aby k nim již nedocházelo. Úpravy jsou následující:

- Seznam pavilonů byl předělán. Po úpravě je v každé položce název pavilonu a dvě tlačítka – jedno je tlačítko "WWW" zastupující předchozí funkci ikony zeměkoule, druhé je tlačítko "Zvířata" zastupující předchozí funkci položky. Kliknutí mimo tato dvě tlačítka nemá žádný efekt. Touto úpravou je snížena možnost zmatení uživatele, protože obě funkce jsou ve formě tlačítek a jsou jasně oddělené.

- Algoritmus pro vyhledávání byl upraven tak, aby nebyl senzitivní vůči diakritice. Při vyhledání slova "indicky" je nyní nalezen i "Lev Indický" (obdobně pro další písmena s diakritikou). Tato úprava je provedena primárně z toho důvodu, že na mobilních přístrojích většina uživatelů píše bez diakritiky a vyhledávání je tedy tímto způsobem pro uživatele jednodušší.

- `MainFragment` přetěžuje funkci `onBackPressed` (funkce určující, co se má stát v případě zmáčknutí Android tlačítka zpět) tak, že pokud je otevřené vyhledávání, je při stisknutí tlačítka zpět pouze zavřeno a zrušeno. Jinak je zavolána funkce nadtypu, která zavře aplikaci.

- Název zvířete, latinský název a název pavilonu, kde se zvíře nachází, jsou přesunuty na jiné místo a za ně je přidáno částečně transparentní tmavé pozadí. Tímto způsobem jsou informace vždy čitelné a zároveň díky průhlednosti nezakrývají úplně fotku zvířete.

- Protože je seznam zvířat chápán jako hlavní obrazovka, je mu chování tlačítka zpět uzpůsobeno. `MainActivity` přetěžuje funkci `onBackPressed` tím způsobem, že v případě, že je aktivní fragment jiný než `MainFragment`, je `MainFragment` nastaven nově jako aktivní. Jinak je zavolána funkce nadtřídy. Aplikace tedy byla předělána tak, aby – v případě zmáčknutí tlačítka zpět v seznamu pavilonů nebo v seznamu akcí – byla aplikace přepnuta zpět na seznam zvířat.

- Tlačítka "Zrušit" a "Obnovit" byla přejmenována na "Zpět" a "Zrušit". Tlačítko "Zrušit" tedy nyní opravdu ruší filtrování, místo pouhého zrušení nastavení filtrování. To přispěje ke srozumitelnosti funkce tlačítek.

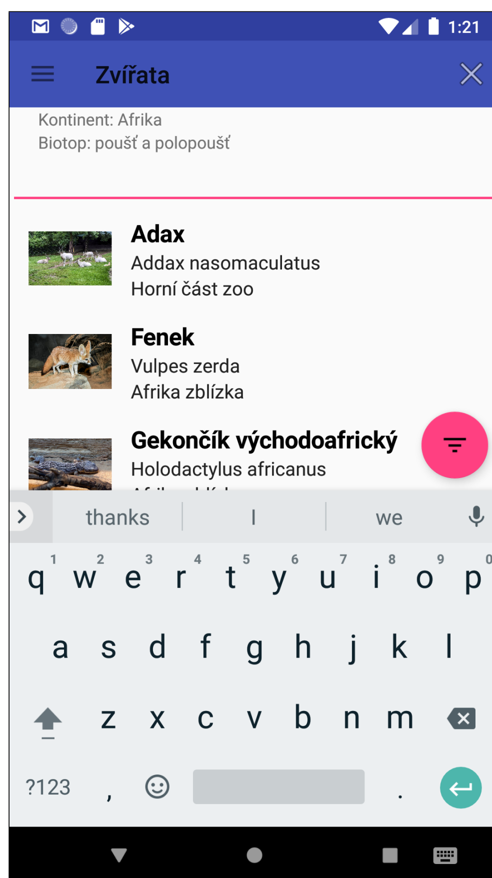
- Dřívější tlačítko "Obnovit" (nyní tlačítko "Zrušit") bylo upraveno tak, aby okamžitě aktivovalo restartovaný filtr a zavřelo dialog, namísto toho, aby pouze restartovalo hodnoty v jednotlivých dropdown menu. Takto je aplikace responzivnější a snadnější na ovládání.

- Nad textové pole pro vyhledávání byl přidán seznam aktivních filtrů. Pokud nějaký filtr není aktivní, pak není vůbec zobrazen. Tím, že jsou aktivní filtry zobrazeny přímo nad vyhledáváním, by nemělo dojít k tomu, že uživatel zapomene nějaký filtr zapnutý. I v případě, že zrovna píše do vyhledávacího pole, je hned nad tímto polem stále zobrazený seznam aktivních filtrů. To by mělo být dostatečným upozorněním pro uživatele, že filtry jsou stále aktivní i během vyhledávání. S touto změnou byla také upravena funkce `onBackPressed` tak, aby došlo k zrestartování aktivních filtrů v případě stisknutí zpětného tlačítka.

Na obrázku 3.7 lze vidět úpravu popsanou v posledním bodu, tedy jak byl přidán seznam aktivních filtrů. Další snímky obrazovek z finální verze aplikace lze najít v příloze C.

3.4 Publikace

Tato sekce práce se zabývá publikací aplikace v obchodě *Google Play* a s tím spojeným procesem. Jelikož je publikací aplikace poskytnuta koncovým uživatelům, je vhodné provést určitou přípravu aplikace, která umožní pružně reagovat na chyby v aplikaci a monitorovat její využití. V případě, že dojde k pádu aplikace z důvodu nějaké chyby, lze tuto informaci získat dříve, než z reportů uživatelů nebo z negativních recenzí (v některých případech, pravděpodobně ve většině, by informace o pádu nebyla získána vůbec, jestliže uživatel prostě odinstaluje aplikaci, aniž by psal recenzi nebo informoval vývojáře). Všechny tyto možnosti jsou samozřejmě nežádoucí, a proto je potřeba reagovat co nejrychleji a vydat novější verzi s opravou, aby se se stejnou chybou setkalo co nejméně uživatelů.



Obrázek 3.7: Vyhledávání opravené dle výsledků uživatelského testování

Špatná hodnocení, která by uživatelé v důsledku chyb zanechávali, by vedla k menšímu počtu nových uživatelů, protože hodnocení aplikací je důležitým faktorem v rozhodnutí, zda si uživatel onu aplikaci nainstaluje. Monitorování chyb je proto kritickým faktorem, který může rozhodnout o úspěchu aplikace.

Monitorování chování uživatelů (které obrazovky jsou nejčastěji zobrazené, kde straví nejvíce času,...) může poskytnout důležité informace použitelné k dalšímu vývoji aplikace v budoucnu, díky zaměření na důležitá místa, případně k odstranění nebo zjednodušení nepoužívaných obrazovek apod. Může sloužit i ke zjištění zájmu o různá zvířata a k následnému zlepšení výsledků vyhledávání, či ke zpřesnění informací o těchto zvířatech nebo k dalším podobným úpravám.

3.4.1 Firebase

Firebase je platforma zaměřená na podporu tvorby a monitorování webových a mobilních aplikací, poprvé uvedená v roce 2011. Skládá se z množství různých

produktů podporujících jak vývoj, tak monitorování a reportování. Ve Firebase není nutné využití všech produktů a lze ji využít právě na výše zmíněné přípravy – tedy na získávání informací o pádech aplikace a na monitorování chování uživatelů. [35]

Přidání aplikace do Firebase

Před použitím jednotlivých služeb Firebase, je nutné aplikaci napojit na tuto platformu. Nejprve je tedy vytvořen projekt (ten seskupuje dohromady související aplikace), do projektu je pak přidána Android aplikace. K tomu je potřeba jen několik krátkých kroků.

Zadání názvu balíčku (v tomto případě `cz.kosut.lexikon`), pojmenování aplikace (nepovinné, ale vhodné pro orientaci) a certifikát (lze vygenerovat v příkazovém řádku). Na základě těchto informací Firebase vygeneruje konfigurační soubor `google-services.json`, který je nutné přidat do projektu Android aplikace. V dalším kroku je přidána do aplikace závislost na balíčku `firebase-analytics`. Nejnovější hlavní verze je 17, tato verze ale vyžaduje, aby aplikace využívala podpůrných knihoven AndroidX. Aplikace ovšem používá podpůrné knihovny Android a konverze na AndroidX není triviální. Proto je zde přidána závislost na verzi balíčku 16, která je ihned kompatibilní s aplikací. Na závěr je aplikace zkompilována a spuštěna, přičemž Firebase automaticky zaeviduje komunikaci a tím je aplikace přidána do projektu.

Výše uvedené kroky zahrnovaly několik úprav přímo v aplikaci (konkrétně přidání závislosti na balíčku `firebase-analytics` a přidání konfiguračního souboru. Tyto úpravy jsou důvodem, proč je nutné postarat se o monitoring ještě před publikací aplikace. Samozřejmě by bylo možné tyto věci přidat po publikaci, ale vyžadovalo by to nahrání nové verze aplikace a od uvedení do této chvíle by nebyly sbírány žádné statistiky.

Crashlytics

Crashlytics je jedna ze služeb nabízených v rámci platformy Firebase od ledna 2017. [36] Jedná se o lehkou službu reportující pády aplikace v reálném čase. Automaticky seskupuje jednotlivé pády aplikace, které spolu souvisejí a poskytuje další informace pomáhající s jejich řešením a statistiky týkající se těchto chyb (například jaké chyby ovlivňují nejvíce uživatelů, a je tedy potřeba odstranit je jako první). [35]

Pro využití této služby je opět nutné provést úpravy v aplikaci. V tomto případě se jedná pouze o přidání závislosti na balíčku `crashlytics` a následné spuštění nově zkompilované aplikace, aby došlo k ověření toho, že vše správně funguje.

Po tomto nastavení je ve Firebase nově možné zobrazit dashboard s informacemi o pádech aplikace. Zde lze vidět procento uživatelů, kterým funguje aplikace bez pádů, trendy (počet pádů, zda je jich více nebo méně v porovnání

s minulostí nebo mezi různými verzemi) a také konkrétní problémy v aplikaci a jejich detailní informace.

Events

Events je další službou nabízenou v rámci platformy Firebase. Poskytuje možnost ukládat určité události, které se v aplikaci dějí. [37] Tyto informace o událostech jsou pak odesílány na servery platformy Firebase, kde je možné na ně nahlížet přes dashboard *Events*. Zde jsou uvedeny počty, kolikrát a kdy se jaká událost stala, případně jsou u nich uvedeny další parametry (například u události značící, že se uživatel přesunul do nové aktivity, je v parametrech předána informace o předchozí aktivitě a o čase stráveném v předchozí aktivitě).

Tyto události se neodesílají ihned. Pokud by tomu tak bylo, způsobovalo by zasílání událostí vysokou spotřebu baterie. Místo toho jsou informace ukládány a po jedné hodině jsou odeslány všechny naráz. Jelikož jsou informace odesílány pomocí služby od Google, dojde k odeslání i v tom případě, že do hodiny uživatel aplikaci odinstaluje. Kromě výjimečných případů tedy nedojde k tomu, že by informace o událostech byly ztraceny. [38]

Tato služba je zde využita právě pro monitorování chování uživatelů. Na několika místech v aplikaci jsou odesílány informace o tom, kde se uživatel nachází nebo co vybral. Zde není potřeba do aplikace přidávat závislost na nějakém balíčku, protože tato funkce je již zahrnuta v jednom z předchozích. Je ale potřeba přidat do programu odesílání informací o událostech na požadovaná místa.

Následující úryvek kódu uvádí, jak je událost odesílána.

```
val bundle = Bundle()
bundle.putString(FirebaseAnalytics.Param.CONTENT_TYPE, "animal")
bundle.putString(FirebaseAnalytics.Param.ITEM_ID,
    animalIdLayout.toString())
bundle.putString(FirebaseAnalytics.Param.ITEM_NAME,
    titleText.text.toString())
FirebaseAnalytics.getInstance(context)
    .logEvent(FirebaseAnalytics.Event.SELECT_CONTENT, bundle)
```

Tento úryvek je z Epoxy modelu položky pro zvíře v seznamu zvířat (více informací v *MainFragment* části v 3.2.2). Když uživatel stiskne položku zvířete, je tento kód proveden před otevřením aktivity s detailem zvířete. Na prvním řádku je inicializovaná instance třídy `Bundle`. Do této instance jsou na dalších řádcích přidány informace. Nejdříve je vložena informace, že se jedná o zvolení zvířete. Následně je uložen identifikátor zvoleného zvířete a pak také jméno zvoleného zvířete. Na posledním řádku je na instanci třídy `FirebaseAnalytics` zavolána metoda `logEvent`, do které je zadán typ události (v tomto případě `SELECT_CONTENT`, neboli “zvolení obsahu” – tato událost

označuje, že uživatel vybral nějaký obsah) a výše inicializovaná instance třídy `Bundle`. V této metodě je provedeno uložení informací o události a jejich pozdější odeslání.

Podobný kód je na dalších místech v aplikaci, samozřejmě s jinými parametry. Ukládání události je prováděno například při přechodu mezi jednotlivými seznamy na hlavní obrazovce, při vybrání a zobrazení detailu zvířete nebo při zobrazení webových stránek s informacemi o konkrétním pavilonu. To poskytne dost informací o tom, jaké obrazovky a funkce jsou nejvíce využívány.

3.4.2 Publikace na Google Play

Publikace aplikací na Google Play se skládá z řady kroků. Část z nich je právě na platformě Google Play a část je potřeba udělat v aplikaci, případně ve vývojovém prostředí.

Vytvoření balíčku

Na Google Play je potřeba nahrát aplikaci. K tomu je nutné vytvořit balíček aplikace, protože na Google Play nejsou samozřejmě nahrávány přímo zdrojové soubory. Historicky se k tomu využívá formát APK (Android Package), který lze chápat jako instalační soubor aplikace. Nově, od září 2018 (společně s vydáním Android Studio verze 3.2), nabízí Google možnost vytvoření tzv. Android App Bundle (lze přeložit jako "svazek Android aplikace"). [39]

Android App Bundle je soubor, který obsahuje zkompilovaný zdrojový kód a další zdrojové soubory aplikace (obrázky, texty, atd.). Vygenerování výše zmíněného instalačního souboru APK však nechává na serverech Google. To přináší několik výhod. [39]

Tím, že má Google k dispozici tento svazek, může vytvořit různé verze APK, které automaticky poskytne různým typům zařízení. Může totiž APK vygenerovat takovým způsobem, aby uživatel stáhnul pouze ten obsah, který sám potřebuje pro spuštění aplikace. Nemusí tedy mít rovnou všechny zdrojové soubory, které zpravidla obsahují různé velikosti ikon a obrázků na zařízení s různým rozlišením či DPI (*dots per inch* – hustota bodů na obrazovce), nebo všechny texty zapsané do všech poskytovaných jazyků. Místo toho Google vytvoří APK balíček obsahující pouze soubory relevantní pro konkrétní zařízení, které aplikaci stahuje. Tím se zmenší velikost aplikace, takže uživatel nemusí stahovat takové množství dat.

Uchovávání různých verzí pro různé typy zařízení dělali předtím navíc přímo autoři aplikací, což bylo náročné a tento nový systém přesouvá zodpovědnost na servery Google.

Google stále podporuje obě možnosti a vývojář si tak může vybrat. Jelikož je ale Android App Bundle modernější a má v porovnání s nahráním APK pouze výhody, byl v aplikaci využit právě tento formát. Jeho vygenerování se provádí ve vývojovém programu Android Studio. Zde také dojde k podpisu

aplikace (to je možné přenechat také na servery Google nebo lze udělat zde). Podpis slouží k ověření původu a je důležité uchovat klíč, kterým je aplikace podepsána. Ten je totiž později na Google Play ověřován při vydávání dalších verzí aplikace, a pokud by nová verze aplikace nebyla podepsaná stejným klíčem jako předchozí, pak jí nelze nahrát (v tom případě by nová verze musela být nahrána jako nová aplikace). Android Studio poté provede kompilaci programu a vytvoří soubor formátu AAB (zmiňovaný Android App Bundle).

Nahrání balíčku a zveřejnění

Prvním nutným krokem je vytvoření účtu vývojáře, není možné publikovat aplikace s obyčejným účtem Google. K vytvoření účtu musí vývojář zadat některé informace o sobě, souhlasit se smlouvou, která stanovuje práva a povinnosti vývojáře ve vztahu ke Googlu a uhradit poplatek 25 dolarů. Ten je jednorázový a lze následně publikovat i další aplikace na tomto účtu.

Po vytvoření účtu se lze přihlásit do *Google Play Console* (`https://play.google.com/apps/publish`). To je prostředí, skrze které se spravuje vydávání aplikací na Google Play. Zde je nutné vyplnit informace o aplikaci, které budou zobrazeny na stránce aplikace v obchodu. Některé informace jsou povinné. Mezi ně zde patří:

- Název aplikace
- Krátký popis
- Úplný popis
- Ikona – k vytvoření byla použita jedna volně dostupná ikona zvířete z `flaticon.com`. Pouze je nutné zmínit tvůrce ikony v popisu aplikace. Stejnou ikonu je pak potřeba přidat přímo do aplikace.
- Snímky obrazovky – alespoň dva snímky obrazovky aplikace. Byly nahrány 4, zobrazující seznam zvířat, akcí a detail zvířete.
- Hlavní grafika – obrázek s rozměry 1024x500 pixelů. Grafika, která bude zobrazená nahoře ve stránce v obchodu. Zde byl vytvořen obrázek s modrým pozadím, dvěma snímky obrazovky a názvem aplikace.
- Typ aplikace – zda se jedná o aplikaci nebo o hru
- Kategorie aplikace – výběr z několika kategorií, zde zvolena kategorie *Travel and Local*.
- Email – kontakt na vývojáře
- Zásady ochrany soukromí – odkaz na stránku se zásadami ochrany soukromí. Lze zvolit, že není potřeba.

Kromě těchto informací je potřeba vytvořit hodnocení obsahu. K tomu slouží dotazník, který je nutné vyplnit. V dotazníku jsou otázky zaměřené na to, zda je aplikace cílena na děti, zda obsahuje sexuální či nevhodný obsah, apod. Po vyplnění dotazníku je automaticky vytvořeno hodnocení pro všechny trhy (ESRB rating, PEGI rating atd.). Dotazník je důležité vyplnit pravdivě, protože aplikace je poté kontrolována i firmou Google a v případě neshod může dojít ke stažení aplikace z Google Play. Jelikož tato aplikace zobrazuje naučné informace o zvířatech v ZOO Praha, nebyl s vyplněním dotazníku problém a vygenerovaný rating ESRB je "E for Everyone" (vhodný pro všechny).

Dále je nutné vyplnit stránku s informacemi o monetizaci aplikace. Zde bylo nastaveno, že je aplikace zdarma. Také je zde povinné potvrzení toho, že aplikace splňuje zásady obsahu (neobsahuje zakázaný obsah, jako jsou projevy nenávisti či zneužívání dětí, splňuje zásady ochrany soukromí, apod.). Celé znění zásad obsahu je dostupné na stránce `https://play.google.com/intl/cs/about/developer-content-policy/`.

Po vyplnění výše uvedených informací a splnění všech podmínek lze přistoupit k vytvoření a vydání verze aplikace.

Google nabízí publikaci aplikace v několika různých kanálech. Ty jsou následující:

- Interní testování – tento typ umožňuje zpřístupnit aplikaci k internímu testování během minut. Verze je poskytnuta pouze konkrétním uživatelům (testerům).

- Alfa kanál (uzavřené testování) – určeno k uzavřenému testování. Publikace probíhá stejně jako u dalších typů (musí tedy projít schválením od společnosti Google). Opět je ale poskytnuta pouze konkrétním uživatelům.

- Beta kanál (otevřené testování) – určeno k otevřenému testování. Aplikace je dostupná dané podskupině uživatelů.

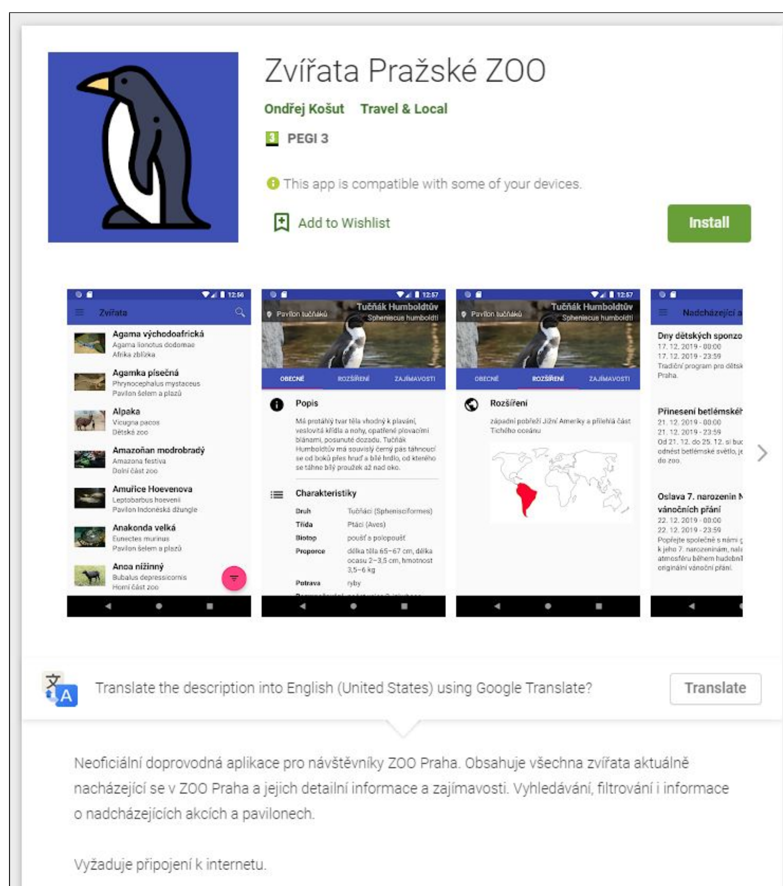
- Produkční kanál – verze, která je zde publikovaná, je dostupná všem uživatelům.

Pro vydání verze aplikace je nejprve potřeba zvolit kanál, kterým má být vydána. Následně lze verzi pojmenovat, zadat podrobnější informace o verzi (co je ve verzi za změny) a nahrát tuto verzi. K nahrání lze použít formát APK nebo AAB (formáty jsou blíže popsány výše v této sekci). Zde byl z již zmíněných důvodů použit formát AAB.

Vydání verze produkčním kanálem ovšem ještě neznamená, že je okamžitě dostupná v obchodě Google Play. Kvůli snaze poskytovat uživatelům co nejlepší služby se Google snaží zamezit publikaci podvodných, či jinak závadných aplikací. Proto je zaveden proces kontroly aplikací, který většinou trvá méně než den, kromě prvních verzí aplikací a obzvlášť aplikací u účtů,

které ještě žádnou nepublikovaly, což bylo případem zde. V takovém případě může schválení trvat až 7 dní, výjimečně déle. Do té doby je stav aplikace *Pending publication* (neboli “čeká na publikaci”). Aplikace mezitím prochází kontrolou zásad obsahu (zmíněno výše) apod. a ve chvíli, kdy kontrolou projde, je vývojář notifikován a aplikace přechází do stavu *Published* (“publikováno”).

Stránka aplikace v obchodu Google Play po publikaci je vidět na obrázku 3.8 a je dostupná z url: `https://play.google.com/store/apps/details?id=cz.kosut.lexikon`.



Obrázek 3.8: Stránka v obchodu Google Play. Ikona od uživatele Freepik z www.flaticon.com

Závěr

Tato práce se zabývala vytvořením doprovodné aplikace ZOO Praha a všemi dalšími procesy spojenými s vývojem takové aplikace.

Nejprve byla provedena analýza dostupných dat. Byly zmapovány jednotlivé datové sady, které ZOO Praha zveřejnila, byly porovnány různé formáty, ve kterých byla data dostupná a jejich aktuálnost (která se lišila mezi formáty). Byly popsány aplikace dostupné v obchodě Google Play, které svým obsahem souvisejí s Pražskou ZOO. Na základě analýzy těchto dat byl proveden detailní popis možných funkcionalit a definice funkčních a nefunkčních požadavků. Tím byl naplněn cíl práce zahrnující navržení vhodné funkcionality na základě dostupných dat.

Následně byly popsány aktivity spojené s vývojem serverové části. Nejprve byla provedena volba databázového systému. Následně byl vytvořen server jako takový. K tomu účelu byl použit jazyk Node.js, zvolený přímo v zadání práce. Node.js zde byl stručně popsán. Následně byla popsána architektura REST, využitá ve webovém rozhraní serveru. Byla detailně popsána implementace i s uvedením ukázek kódu serveru. Dále byla nasazena databáze do cloudové platformy Atlas a server byl nasazen do cloudové platformy Google Cloud. Volba cloudových platformem byla zdůvodněna, a bylo popsáno, jak bylo webové rozhraní dokumentováno a testováno. Tím byl naplněn cíl práce zahrnující navržení a implementaci serveru v jazyce Node.js.

Dále byla popsána implementace samotné mobilní aplikace. Za tímto účelem byl nejprve proveden návrh uživatelského rozhraní. Ten vycházel primárně z funkčních požadavků definovaných dříve. Jednotlivé požadavky byly přiřazeny k navrženým obrazovkám, které je naplňovaly. Zároveň došlo k diskuzi nad grafickým pojetím aplikace s vedoucím práce, který schválil využití standardního material designu, čímž byl naplněn cíl práce zahrnující konzultaci nad grafickým pojetím aplikace. Po návrhu uživatelského rozhraní došlo k návrhu architektury a zdůvodnění, proč byl zvolen k implementaci jazyk Kotlin. Tímto byl naplněn cíl práce zahrnující navržení a implementaci mobilní aplikace.

Byly popsány všechny využité knihovny, a bylo popsáno, jak byla vyřešena implementace v jednotlivých třídách, a k tomu byly i popsány úryvky zdrojového kódu z různých částí aplikace.

Dále bylo provedeno testování aplikace. Nejdříve byly popsány druhy automatických testů, a poté byla popsána implementace těchto testů, opět s ukázkami kódu. Následně byl proveden proces uživatelského testování, byly popsány chyby odhalené v aplikaci a v uživatelském rozhraní a byla provedena oprava těchto chyb, čímž byl naplněn cíl otestovat mobilní aplikaci.

Nakonec byla provedena publikace aplikace do obchodu Google Play. Ta zahrnovala nejprve implementaci základního monitoringu za účelem zlepšování aplikace v budoucnosti, dále vytvoření balíčku, a nakonec procesy na platformě Google Play, které bylo nutné podstoupit před zveřejněním aplikace. Aplikace byla zveřejněna v obchodě Google Play. Tím byl naplněn hlavní cíl práce, kterým bylo vytvořit a zveřejnit aplikaci.

Došlo tedy k naplnění všech cílů práce vytyčených v jejím zadání. Byly naplněny všechny dílčí cíle, a tím i hlavní cíl práce.

Aplikace je ve chvíli dokončení této práce dostupná zdarma v obchodě Google Play. Aplikaci lze do budoucna zlepšit přidáním některých funkcionalit, které nyní v aplikaci nejsou. Je ale třeba dát si pozor na příliš velké množství funkcionalit, které by mohlo aplikaci znehodnotit pro běžné užití. Jednou z možných nových funkcionalit je například vytvoření interaktivní mapy pavilonů, která by umožnovala navigovat uživatele ke zvolenému pavilonu nebo expozici. V tuto chvíli bohužel ale nejsou dostupná data, se kterými by bylo možné tuto funkci realizovat. Pokud dojde k jejich doplnění do datových sad, lze navigaci přidat a zvýšit tím hodnotu aplikace pro uživatele.

Literatura

[1] Szotkowská, H.: Historicky nejvyšší návštěvnost ZOO Praha. *zoopraha.cz [online]*, 2014, [cit. 2019-12-29]. Dostupné z: `https://www.zoopraha.cz/aktualne/ostatni-clanky/8987-historicky-nejvyssi-navstevnost-zoo-praha`

[2] Šteffelová, A.: ZOO Praha je pátá nejlepší ZOO na světě! *zoopraha.cz [online]*, 2017, [cit. 2019-12-29]. Dostupné z: `https://www.zoopraha.cz/aktualne/ostatni-clanky/10891-zoo-praha-je-pa-ta-nejleps-i-zoo-na-sve-te`

[3] ZOO v číslech. *zoopraha.cz [online]*, 2018, [cit. 2019-12-29]. Dostupné z: `https://www.zoopraha.cz/zvirata-a-expozice/zvirata-v-cislech`

[4] Počet uživatelů chytrých mobilních telefonů se v ČR za pět let více než ztrojnásobil. *w4t.cz [online]*, 2017, [cit. 2019-12-29]. Dostupné z: `https://www.w4t.cz/pocet-uzivatelu-chytrych-mobilnich-telefonu-se-v-cr-za-pet-let-vice-nez-ztrojnasobil-43829/`

[5] Opendata hlavního města Prahy. O nás. [online], [cit. 2019-12-15]. Dostupné z: `http://opendata.praha.eu/about`

[6] Open data. *European Commission Policies, information and services [online]*, červen 2019, [cit. 2019-12-15]. Dostupné z: `https://ec.europa.eu/digital-single-market/en/open-data`

[7] GOOGLE: Google Play. *play.google.com [online]*, 2019, [cit. 2019-12-15]. Dostupné z: `https://play.google.com/store`

[8] Number of available applications in the Google Play Store from December 2009 to September 2019. *Statista [online]*, říjen 2019, [cit. 2019-12-15]. Dostupné z: `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`

[9] AITOM: Údolí Slonů. *play.google.com [online]*, 2019, [cit. 2019-12-29]. Dostupné z: `https://play.google.com/store/apps/details?id=aitom.zoo`

[10] AITOM: Pavilon Želv. *play.google.com [online]*, 2019, [cit. 2019-12-29]. Dostupné z: `https://play.google.com/store/apps/details?id=cz.aitom.zoo.zelvy`

[11] Zavadil, M.: Lexikon zvířat Zoo Praha (Early Access). *play.google.com [online]*, 2019, [cit. 2019-12-29]. Dostupné z: `https://play.google.com/store/apps/details?id=com.zavadil.lexiconzoo`

[12] Chodorow, K.: *MongoDB: the definitive guide: powerful and scalable data storage*. "O'Reilly Media, Inc.", 2013.

[13] OpenJS Foundation: About Node.js. [online], [cit. 2019-12-29]. Dostupné z: `https://nodejs.org/en/about/`

[14] Fusik, P.: List of languages that compile to JS. *github.com [online]*, 2019, [cit. 2019-12-29]. Dostupné z: `https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js`

[15] Fielding, R. T.; Taylor, R. N.: *Architectural styles and the design of network-based software architectures*, ročník 7. University of California, Irvine Doctoral dissertation, 2000.

[16] Fielding, R. T.; Taylor, R. N.: *Architectural styles and the design of network-based software architectures*, ročník 7, kapitola 3.4.4. University of California, Irvine Doctoral dissertation, 2000.

[17] Richardson, L.; Amundsen, M.; Amundsen, M.; aj.: *RESTful Web APIs: Services for a Changing World*. "O'Reilly Media, Inc.", 2013.

[18] MongoDB, Inc: MongoDB Atlas Documentation. [online], [cit. 2019-12-29]. Dostupné z: `https://docs.atlas.mongodb.com`

[19] MongoDB, Inc: MongoDB Atlas Documentation. Sekce "Cluster Tier". [online], [cit. 2019-12-29]. Dostupné z: `https://docs.atlas.mongodb.com/cluster-tier/`

[20] MongoDB, Inc: MongoDB Atlas Documentation. Sekce "Replication". [online], [cit. 2019-12-29]. Dostupné z: `https://docs.mongodb.com/manual/replication/`

[21] Krishnan, S.; Gonzalez, J. L. U.: *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*. Springer, 2015, s. 7–12.

[22] Elias, J.: Google Cloud is generating $8 billion in revenue a year and plans to triple sales force. *cnbc.com [online]*, 2019, [cit. 2019-12-29]. Dostupné z: `https://www.cnbc.com/2019/07/25/google-cloud-at-8-billion-in-sales-a-year-and-is-tripling-sales-force.html`

[23] GOOGLE: Google App Engine Documentation. [online], [cit. 2019-12-29]. Dostupné z: `https://cloud.google.com/appengine/docs/`

[24] GOOGLE: Google App Engine Documentation, Sekce "App Engine Flexible Environment". [online], [cit. 2019-12-29]. Dostupné z: `https://cloud.google.com/appengine/docs/flexible/`

[25] Smartbear Software: OpenAPI Specification. [online], [cit. 2019-12-29]. Dostupné z: `https://swagger.io/specification/`

[26] GOOGLE: Material Design Introduction, Sekce "Principles". [online], [cit. 2019-12-29]. Dostupné z: `https://material.io/design/introduction/`

[27] Brian, M.: Google's new 'Material Design' UI coming to Android, Chrome OS and the web. *engadget.com [online]*, 2019, [cit. 2019-12-29]. Dostupné z: `www.engadget.com/2014/06/25/googles-new-design-language-is-called-material-design`

[28] JetBrains: Kotlin Documentation, Sekce "Comparison to Java Programming Language". [online], [cit. 2019-12-29]. Dostupné z: `https://kotlinlang.org/docs/reference/comparison-to-java.html`

[29] Square, Inc: Retrofit. [online], [cit. 2019-12-29]. Dostupné z: `https://square.github.io/retrofit/`

[30] Insert Koin: What is KOIN? [online], [cit. 2019-12-29]. Dostupné z: `https://github.com/InsertKoinIO/koin`

[31] AirbnbEng: Epoxy: Airbnb's View Architecture on Android. *medium.com [online]*, 2016, [cit. 2019-12-29]. Dostupné z: `https://medium.com/airbnb-engineering/epoxy-airbnbs-view-architecture-on-android-c3e1af150394`

[32] Weiss, T.: We Analyzed 30,000 GitHub Projects - Here Are The Top 100 Libraries in Java, JS and Ruby. *blog.overops.com [online]*, 2013, [cit. 2019-12-29]. Dostupné z: `https://blog.overops.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/`

[33] GOOGLE: Android Reference: ActivityTestRule. [online], [cit. 2019-12-29]. Dostupné z: `https://developer.android.com/reference/android/support/test/rule/ActivityTestRule`

[34] Vogel, L.: Android user interface testing with Espresso - Tutorial. *vogella.com [online]*, 2016, [cit. 2019-12-29]. Dostupné z: https://www.vogella.com/tutorials/AndroidTestingEspresso/article.html

[35] GOOGLE: Firebase Documentation. [online], [cit. 2019-12-29]. Dostupné z: https://firebase.google.com/docs/?authuser=0

[36] Paret, R.: Fabric is Joining Google. *fabric.io [online]*, 2017, [cit. 2019-12-29]. Dostupné z: https://fabric.io/blog/fabric-joins-google

[37] GOOGLE: Firebase Documentation: Events. [online], [cit. 2019-12-29]. Dostupné z: https://firebase.google.com/docs/analytics/events?platform=android

[38] Kerpelman, T.: How Long Does it Take for My Firebase Analytics Data to Show Up? *https://firebase.googleblog.com [online]*, 2016, [cit. 2019-12-29]. Dostupné z: https://firebase.googleblog.com/2016/11/how-long-does-it-take-for-my-firebase-analytics-data-to-show-up.html

[39] GOOGLE: Android, About The Platform: Android App Bundle. [online], [cit. 2019-12-29]. Dostupné z: https://developer.android.com/platform/technology/app-bundle

Příloha A

Seznam použitých zkratek

AAB Adroid App Bundle

API Application Programming Interface

APK Android Package

CSV Comma-Separated Values

DPI Dots per Inch

ESRB Entertainment Software Rating Board

GCP Google Cloud Platform

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

ICT Information and Communications Technology

I/O Input/Output

JSON JavaScript Object Notation

NoSQL Not Only SQL

OAS3 OpenAPI Specification

PaaS Platform as a Service

PEGI Pan European Game Information

REST Representational State Transfer

RSS Rich Site Summary

URI Uniform Resource Identifier

XLSX Excel Microsoft Office Open XML Format Spreadsheet file

XML Extensible markup language

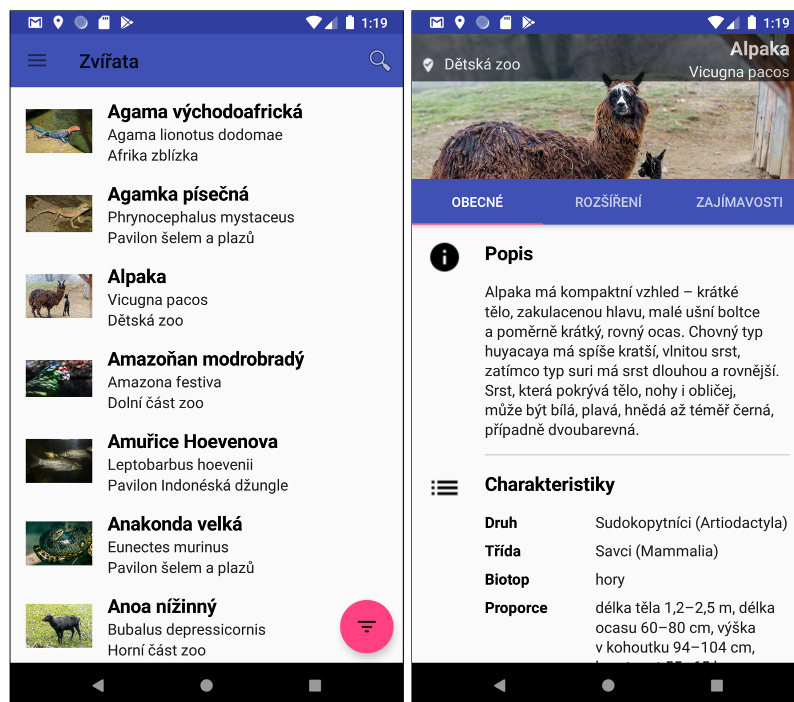
Příloha B

Obsah přiloženého CD

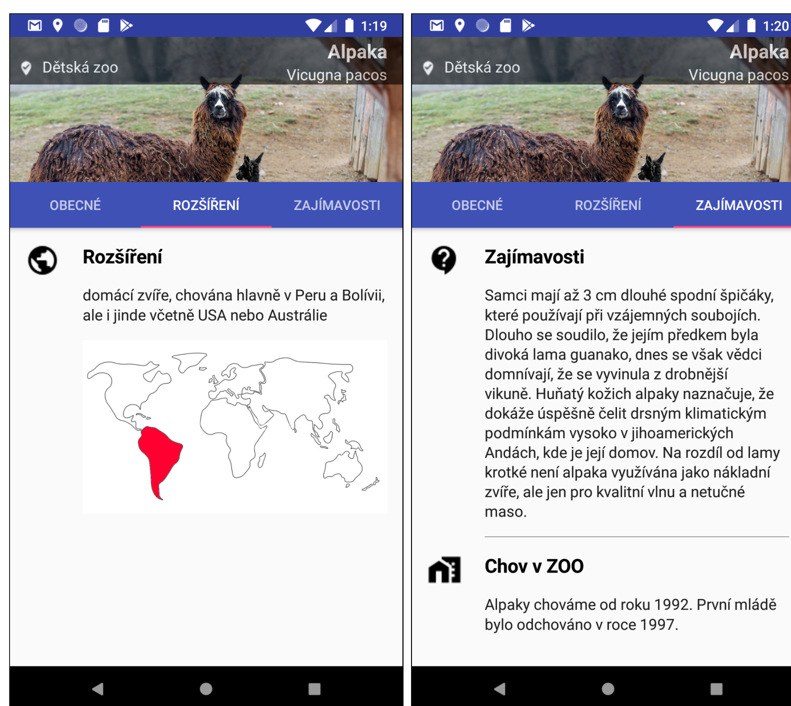
```
readme.txt ............................ stručný popis obsahu CD
abb ................................ adresář aplikací ve formátu ABB
src
  app ................................ zdrojové kódy aplikace
  server ............................. zdrojové kódy serveru
  thesis ............... zdrojová forma práce ve formátu LaTeX
  wireframe ....... zdrojový soubor wireframe ve formátu Axure RP
test ................................... dokumenty testování
  user ........................ nahrávky uživatelských testů
  api ............................ výsledek testování API
text ............................................ text práce
  thesis.pdf ..................... text práce ve formátu PDF
```

Příloha C

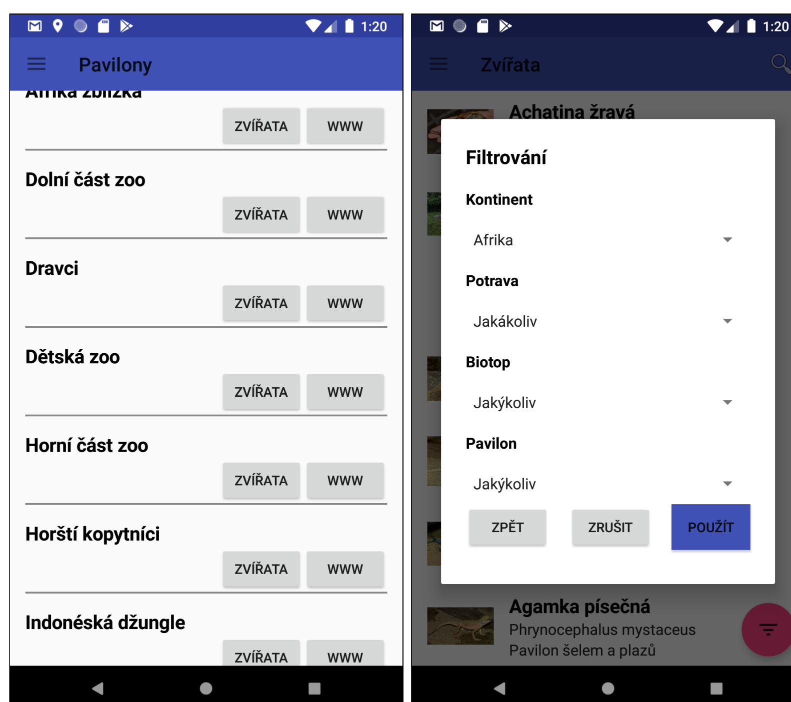
Snímky obrazovky aplikace



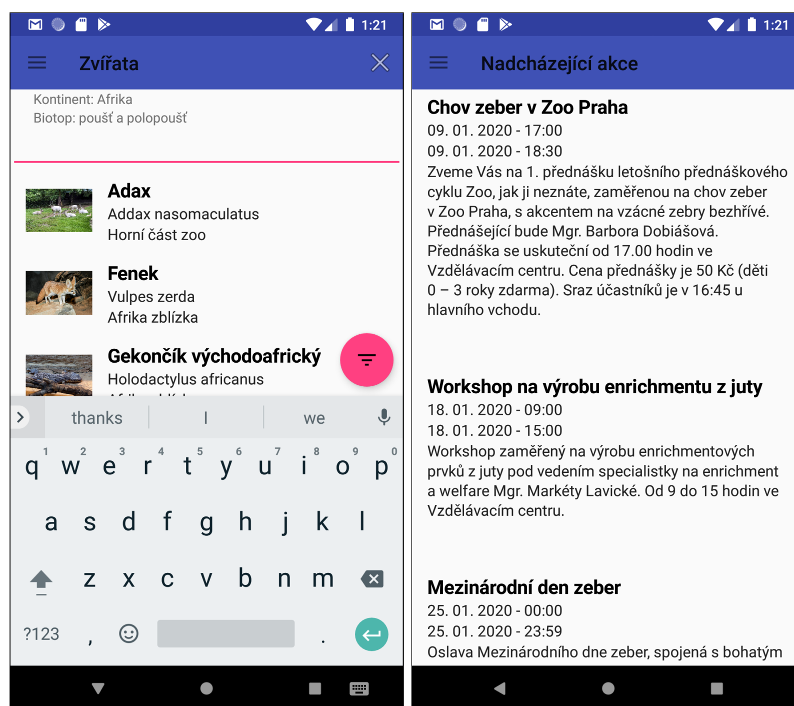
Obrázek C.1: Seznam zvířat (vlevo) a Obecné informace o zvířeti (vpravo)



Obrázek C.2: Rozšíření zvířete (vlevo) a Zajímavosti o zvířeti (vpravo)



Obrázek C.3: Seznam pavilonů (vlevo) a Dialog filtrování zvířat (vpravo)



Obrázek C.4: Vyhledávání se zapnutými filtry (vlevo) a Nadcházející akce (vpravo)