FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Mobilní lexikon zvířat ZOO Praha
Student:	Bc. Petr Budík
Vedoucí:	Ing. Josef Gattermayer, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Mobilní lexikon zvířat ZOO Praha
Pražská ZOO uveřejnila v rámci portálu opendata.praha.eu monžství 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.
• Navrhněte vhodnou funkcionalitu pro mobilní aplikaci na základě dostupných dat.
• Konzultujte s vedoucím práce grafické pojetí aplikace (grafiku dodá vedoucí).
• Navrhněte a implementujte server v Python, který bude data stahovat z portálu opendata.praha.eu a přes REST API nabízet mobilní aplikaci. Implementujte i infrastrukturu pro provoz a škálování backendové služby (as a code).
• Navrhněte, implementujte a otestujte mobilní aplikaci pro iOS.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 7. ledna 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Mobilní lexikon zvířat ZOO Praha

Bc. Petr Budík

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

5. května 2021

Poděkování

Velice rád bych poděkoval vedoucímu práce Ing. Josefu Gattermayerovi, Ph.D. za jeho čas a cenné rady.

Dále děkuji své rodině a přátelům, kteří mě při studiu podporují.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při 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 předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2021 . . . . . .

Odkaz na tuto práci

Budík, Petr. *Mobilní lexikon zvířat ZOO Praha.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Účelem této práce je vytvoření doprovodné mobilní aplikace pro návštěvu Zoo Praha určené pro zařízení se systémem iOS. Tato aplikace poskytuje všechny informace o zde žijících zvířatech a plně interaktivní mapu Zoo Praha, pomocí které se může návštěvník v areálu orientovat. Data pro tuto aplikaci zpracovává server v jazyce Python, který je též potřeba implementovat.

V práci je popsán celý proces vývoje této aplikace. Nejdříve jsou popsána dostupná data a analyzovány již existující podobné aplikace, na tomto základě jsou vytvořeny požadavky na výslednou iOS aplikaci a server. Následně je popsána implementace a otestování pomocného serveru v jazyce Python, která je ukončena jeho nasazením společně s používanými databázemi na různé cloudové systémy. Nakonec je popsán návrh, implementace a testování (automatické a uživatelské) mobilní aplikace.

Klíčová slova iOS, Swift, Python, Otevřená data, Web scraping, MongoDB

Abstract

Goal of the thesis is an implementation of an iOS mobile application which is used as an assistant for visitors of Zoo Prague. The application provides

all data of animals owned by Zoo Prague and a fully interactive map of Zoo Prague which can be used to orient and navigate when visiting. Data required by the application is processed by a Python server which also has to be created.

The thesis describes the whole process of creating the application. The available data and analysis of already existing mobile applications is done first and is used as a base for specification of iOS application's and server's requirements. Then the thesis describes implementation and testing of the Python server which is finished by deployment of said server and its databases to a cloud. In the end, the thesis describes design, implementation and testing (both automated and user testing) of the iOS mobile application.

Keywords iOS, Swift, Python, OpenData, Web scraping, MongoDB

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Zoologická zahrada hl. m. Prahy (dále „Zoo Praha“) je jednou z nejznámějších zoologických zahrad v České republice, kterou jen v roce 2019 navštívilo přibližně 1,5 milionu lidí[1]. Zoo Praha poskytuje mnoho dat o zde žijících zvířatech či probíhajících akcích, které jsou z pohledu návštěvníka velmi zajímavé. Dále existuje papírová mapa, kterou může návštěvník používat k orientaci v areálu Zoo Praha. Tato data lze použít pro vytvoření doprovodné mobilní aplikace pro návštěvníky, která by jim poskytované informace zobrazovala v přehlednější formě a zároveň by nahradila papírovou mapu. V Zoo Praha taková aplikace neexistuje, a proto její vytvoření je hlavním cílem práce.

V první kapitole jsou nejprve analyzovány dostupné zdroje mapových dat a dat ze Zoo Praha, aby bylo zjištěno, co vše lze v aplikaci použít a zda je vůbec možné interaktivní mapu vytvořit. Dále se zabývá i rešerší již existujících podobných aplikací pro systémy iOS a Android. Hlavním výsledkem této kapitoly je seznam definovaných funkčních a nefunkčních požadavků pro server a mobilní aplikaci pro systém iOS.

Ve druhé kapitole je popsána implementace serveru, jehož účelem je automatizované stahování a zpracování všech potřebných dat a jejich uložení do databáze. Tato data jsou následně poskytována přes internet. Nejdříve jsou popsány všechny používané databáze, ve kterých jsou data uložena. Následně se kapitola zabývá implementací jednotlivých funkcionalit samotného serveru, které vychází ze specifikovaných funkčních a nefunkčních požadavků. Dále je popsáno automatické testování serveru. Nakonec je popsáno nasazení databází a serveru na vybrané cloudové platformy.

V poslední kapitole je popsána implementace mobilní aplikace pro systém iOS. Kapitola se nejprve zabývá návrhem uživatelského rozhraní aplikace, který obsahuje grafické zpracování jednotlivých obrazovek a přechodů mezi nimi. Následně je popsána použitá softwarová architektura. Kapitola se dále zabývá samotnou implementací aplikace, která začíná popisem postupů a technologií důležitých pro celou aplikaci, pokračuje popisem implementace ně-

kterých důležitých funkcionalit a uživatelského rozhraní. Nakonec se kapitola zabývá automatickým a uživatelským testováním.

Kapitola 1

Cíl práce

Primárním cílem práce je vytvořit funkční mobilní aplikaci pro systém iOS, která slouží jako doprovodná aplikace při návštěvě Zoo Praha. Je tedy primárně určena pro návštěvníky Zoo Praha nebo lidi, kteří o návštěvě uvažují. Pro dosažení tohoto cíle je nutné projít následujícími fázemi:

- Analýza dostupných dat, na jejíž základě je specifikována funkcionalita výsledné mobilní aplikace.
- Implementace serveru v jazyce Python, jehož účelem je automatické stahování, úprava, ukládání a poskytování dat, která mobilní aplikace používá.
- Navržení, implementace a otestování mobilní aplikace pro systém iOS.

Kapitola 2

Analýza

V této kapitole jsou shrnuta dostupná data a analyzovány některé již existující podobné aplikace. Primárním výstupem kapitoly jsou konkrétní specifikace *funkčních* a *nefunkčních požadavků* na *mobilní aplikaci* a *pomocný server*, které jsou vypracovány na základě těchto informací.

2.1 Dostupná data

2.1.1 Data z OpenData hl. města Prahy

OpenData hl. města Prahy[2] (dále OpenData Praha) je celoměstský portál pro otevřená data, který je dostupný na adrese `opendata.praha.eu`. Zoo Praha zde zveřejňuje datové sady *Návštěvnost*, *Lexikon zvířat*, *Akce v zoo* a *Adopce zvířat*. Více informací o principu otevřených dat v sekci 2.2.

Datové sady mají být aktualizovány jednou za čtvrt roku. Původně byly naposledy aktualizovány v roce 2017 a byly již značně zastaralé, v září 2020 byly aktualizovány novým automatickým skriptem[3]. Došlo k poškození všech datových sad, které místo dat obsahovaly HTML soubor s chybovou hláškou, chyba nebyla jinak indikována. Datové sady byly obnoveny na konci dubna 2021.

Lexikon zvířat

Datová sada obsahuje všechny důležité informace o zde žijících zvířatech a je primárním zdrojem dat pro mobilní aplikaci. Každý záznam zvířete má následující **atributy**:

- název,
- latinský název,
- podrobnosti,
- odkaz na fotografii zvířete,
- řád,
- třída,
- chov v zoo,
- rozmnožování,

- potrava,
- biotop,
- rozšíření,
- kontinent,
- projekty v zoo,
- proporce,
- zajímavosti,
- umístění v zoo.

Atribut „Umístění v zoo“ je užitečný pro zvířata, která se nacházejí v pavilonech, pro zvířata nacházející se mimo pavilony je příliš obecný.

Datová sada je dostupná ve formátech CSV a XLSX, které jsou uloženy přímo na adrese **opendata.praha.eu**. Datová sada má být dostupná i ve formátu JSON, který však odkazuje na již nedostupné interní API v Zoo Praha[4].

Datová sada dále obsahuje i samostatné tabulky pro jednotlivé atributy, které lze použít pro filtraci záznamů v hlavní tabulce. Mezi samostatné tabulky patří např. *seznam biotopů, seznam kontinentů, seznam lokalit,...*

Akce v zoo

Datová sada obsahuje informace o aktualitách ze Zoo Praha, a proto lze tyto data použít pro implementaci seznamu budoucích akcí. V této datové sadě je odkaz na oficiální RSS kanál Zoo Praha, který obsahuje aktuální novinky. Data ve formátu CSV jsou zastaralá a ve formátu JSON odkazují na již nedostupné interní API v Zoo Praha.

Adopce zvířat

Datová sada obsahuje seznam zvířat, která lze adoptovat. Obsahuje český a anglický název zvířete, třídu a cenu jeho adopce v Kč. Tyto data lze u jednotlivých zvířat zobrazit.

Návštěvnost

Datová sada obsahuje pouze roční návštěvnost Zoo Praha. Tato data nejsou pro běžného uživatele zajímavá.

2.1.2 Data ze Zoo Praha

Jak již bylo uvedeno výše (sekce 2.1.1), Zoo Praha zveřejnila na OpenData Praha své datové sady. Nevýhody těchto datových sad jsou:

- zastaralost dat,
- nepravidelnost aktualizace,
- případná dlouhodobá nedostupnost dat kvůli chybné aktualizaci na OpenData Praha, která byla popsána v sekci 2.1.1.

Vzhledem k těmto nevýhodám je vhodné uvažovat o dalším zdroji dat. V minulosti Zoo Praha poskytovalo vlastní API, které bohužel již není dostupné[4]. Použitím této API by data byla vždy aktuální a nebezpečí jejich nedostupnosti nízké.

Data, která jsou v projektu využita, jsou dostupná přímo na oficiálních webových stránkách Zoo Praha[5][6] pro všechny návštěvníky. Tato data je možné získat pomocí techniky zvané „Web scraping“, která je blíže popsána v sekci 2.3.

2.1.3 Data z OpenStreetMap

Projekt OpenStreetMap[7] (dále OSM) je poskytovatel mapových dat. Tyto mapové podklady jsou použity místo standardních map od firmy Apple pro implementaci zobrazované interaktivní mapy, protože pro Zoo Praha jsou mnohem přesnější a obsahují i přesné pozice všech zvířat, která se zde nacházejí. Je založen na principu otevřených dat, který byl popsán v sekci 2.2.

2.1.3.1 Historie projektu

Projekt byl založen v roce 2004 a dnes je udržován širokou komunitou[8]. Mapová data může upravovat každý přihlášený uživatel. Dnes jsou data z OpenStreetMap používány mnoha webovými službami a mobilními aplikacemi[9], mezi které patří i *Mapy.cz*.

2.1.3.2 Formát dat

Mapa je rozdělena do mapových čtverců (*Tile*) v různých stupních přiblížení (*zoom level*). Stupně přiblížení nabývají přirozených hodnot $[0; 19]$, kde hodnota 19 značí nejvyšší přiblížení. Pokud bychom chtěli načíst mapu celé Země ve stupni přiblížení S, pak bychom potřebovali načíst 4^S mapových čtverců.

Mapové čtverce jsou poskytovány mnoha různými službami (OSM tile server)[10]. Samotné OSM má také vlastní tile server, který poskytuje rastrové mapové čtverce ve formátu PNG. Existují i tile servery, které poskytují vektorové mapové čtverce ve formátech GeoJSON, TopoJSON a MVT.

Mapové čtverce lze uložit ve struktuře mnoha složek a podsložek, velmi často jsou však agregovány v jednom souboru typu *MBTiles*[11], což je technicky standardní SQLite databáze.

2.2 Princip otevřených dat

Otevřená data (*Open data*) se řídí podle tzv. „The Open Definition“[12], kterou lze zjednodušeně popsat takto:

„*Otevřená data a jejich obsah mohou být volně užívána, upravována a sdílena kýmkoliv a za jakýmkoliv účelem.*“

Mezi největší zdroje otevřených dat patří vědecká pracoviště, veřejná správa a nevládní organizace. Otevřená data dle definice uvedené výše musí splnit několik požadavků[13]:

- Musí být propagována pod nějakou svobodnou licencí.
- Měla by být volně a zdarma stažitelná z Internetu.
- Musí být strojově čitelná a musí být uložena v otevřeném formátu (JSON, XML, CSV, ...).

2.3 Web scraping

Web scraping je technika založená na získávání nestrukturovaných dat z cílové webové stránky, tato data jsou typicky strukturovaně ukládána do databáze pro další využití či analýzu. Nejčastěji je tento proces automatizován jako program, který stáhne obsah webové stránky (např. statické HTML), a pak extrahuje požadované informace[14][15].

Často je vhodnější získávat stejná data pomocí API, než pomocí web scrapingu. Výhody/nevýhody použití API a web scrapingu jsou popsány v tabulce č. 3.1.

Tabulka 2.1: Výhody/nevýhody web scrapingu

Výhody	**Nevýhody**
+ Lze získat data z webových stránek, pro která neexistuje API. + Data z API mohou být zastaralá oproti datům z webové stránky.	− Nestrukturovaná data je složitější analyzovat. − Je závislý na struktuře webové stránky, která se může často a bez varování měnit. − Není vždy legální. − Může být pomalejší než použití API.

2.4 Architektura REST

Architektura Representational State Transfer (REST) byla poprvé zmíněna v disertační práci z roku 2000, jejíž autorem je Roy Fielding[16]. Služba,

která implementuje REST architekturu, je označovaná jako „RESTful“ služba. REST definuje několika architektonických omezení:

Client-Server – server vystavuje služby, které může klient využít zasláním požadavku. Server požadavek zpracuje a klientovi posílá odpověď.

Statelessness – bezstavovost říká, že stav session mezi klientem a serverem je udržována pouze klientem. Každý požadavek od klienta tak musí obsahovat všechny informace.

Layered system – je vyžadováno, aby bylo možné rozdělit systém do vrstev tak, aby z pohledu klienta nebylo důležitá, s jako vrstvou systému v danou chvíli komunikuje.

Cacheability – je vyžadováno, aby odpověď serveru na požadavek od klienta obsahovala informaci, zda jsou data odpovědi kešovatelná. Pokud je odpověď kešovatelná, lze snížit počet interakcí mezi serverem a klientem.

Uniform interface – jednotné rozhraní umožňuje zjednodušení architektury a oddělení implementace od rozhraní. REST je definován čtyřmi omezeními rozhraní:

- Identification of resources – jednoznačná identifikace zdrojů (např. pomocí URL)
- Manipulation of resources through representations – znalost reprezentace zdroje umožňuje klientovi zdroj upravit nebo smazat
- Self-descriptive messages – zpráva obsahuje veškeré informace, které umožní adresátovi zprávu přečíst
- HATEOAS – „hypertext as the engine of application state“, v odpovědi serveru jsou s daty zároveň i odkazy na další související data

2.5 Podobné aplikace

Zoo Liberec

Oficiální aplikace pro Zoo Liberec[17]. Aplikace již bohužel nemůže stáhnout data ze Zoo Liberec, a proto již není funkční. Mezi její funkcionality patří interaktivní mapa, zobrazení služeb, aktualit a denního programu. Interaktivní mapu bylo možné použít pro navigaci v Zoo Liberec.

Údolí slonů

Aplikace[18] se zaměřuje pouze na část Zoo Praha zvanou „Údolí slonů“. Jde o čistě informativní aplikaci, která nemá žádné pokročilé funkce. Představuje

mnoho informací o zde žijících slonech – jejich způsob života, rozmnožování, jejich úlohu v náboženstvích, ...

2.6 Specifikace požadavků

Na základě dostupných dat jsou v této sekci definovány funkční a nefunkční požadavky. Identifikují nutné funkce, vlastnosti a podmínky, které by mobilní aplikace měla splňovat.

Funkční požadavky specifikují funkcionality, které daná aplikace musí umět. *Nefunkční požadavky* určují technické parametry aplikace a definují omezení na výkon, použitelnost, spolehlivost, apod.

2.6.1 Funkční požadavky mobilní aplikace

F1 – Zobrazení lexikonu zvířat: Zobrazení seznamu všech zvířat nacházejících se v Zoo Praha. Každá položka seznamu obsahuje obrázek zvířete, český název zvířete a jeho přibližné umístění v Zoo Praha. Kliknutím na položku se zobrazí detail zvířete popsaný požadavkem **F4**.

F2 – Filtrace lexikonu zvířat: Lexikon zvířat lze filtrovat pomocí třídy (např. savci), potravy a/nebo biotopu, ve kterém zvíře žije.

F3 – Vyhledávání v lexikonu zvířat: V lexikonu lze textově vyhledávat pomocí názvu zvířete.

F4 – Zobrazení detailu zvířete: Zobrazení informací o konkrétním vybraném zvířeti. Jsou zobrazeny informace ze všech atributů, které byli popsány v datové sadě *Lexikon zvířat* v sekci 2.1.1. Z detailu zvířete je možné přejít na interaktivní mapu popsanou v požadavku **F5**, vybrané zvíře je v mapě zvýrazněno.

F5 – Zobrazení interaktivní mapy Zoo Praha: Interaktivní mapa zobrazuje pouze areál Zoo Praha a jeho blízké okolí. Je založena na datech OSM popsaných v sekci 2.1.3.

Na mapě jsou zobrazeny všechny pavilony a zvířecí výběhy.

F6 – Navigace v interaktivní mapě: Interaktivní mapa popsaná v požadavku **F5** umožňuje v rámci Zoo Praha navigaci uživatele ke zvolenému zvířeti nebo pavilonu.

2.6.2 Nefunkční požadavky mobilní aplikace

N1 – Nativní mobilní aplikace: Aplikace je implementována v jazyce Swift pro mobilní zařízení se systémem iOS a tablety se systémem iPadOS.

N2 – Offline dostupnost dat: Data lexikonu zvířat a interaktivní mapy jsou uložena lokálně v mobilním zařízení. Je tak možné tyto funkcionality používat bez přístupu k Internetu.

2.6.3 Funkční požadavky serveru

SF1 – Stahování dat ze Zoo Praha: Server stahuje ze Zoo Praha dostupná data popsaná v sekci 2.1.2, konkrétně část Lexikon zvířat. Data transformuje a ukládá do vlastní databáze.

SF2 – Stahování OSM dat: Server od zvoleného poskytovatele stahuje a ukládá mapové čtverce areálu Zoo Praha a blízkého okolí. Mapové čtverce agreguje do jednoho MBTiles souboru. Vzhledem k potřebám zobrazování mapy v mobilní aplikaci musí být v souboru MBTiles uložena vektorová data ve formátu GeoJSON. Formát těchto dat je popsán v sekci 2.1.3.2.

SF3 – Automatizace stahování dat: Stahování dat ze Zoo Praha (požadavek **SF1**) a stahování OSM dat (požadavek **SF2**) dělá server automaticky v časech, které mu jsou nastaveny. Uživatelé nemohou tuto aktualizaci dat na serveru sami spustit.

SF4 – Vystavování dat pomocí REST API: Server poskytuje všechna zpracovaná dostupná data prostřednictvím REST API podle architektury REST, která je popsána v sekci 2.4.

2.6.4 Nefunkční požadavky serveru

SN1 – Jazyk: Server je napsán v jazyce Python.

SN2 – GUI serveru: Server slouží primárně jako REST API pro mobilní aplikaci, proto nevyžaduje žádné GUI.

Kapitola 3

Implementace serveru

Tato kapitola obsahuje návrh, implementaci, otestování a nasazení backend serveru v jazyce Python. Jeho účelem je automatické stahování dat ze zdrojů, jejich bezpečné předzpracování, uložení a vystavení skrz REST API primárně pro mobilní aplikaci. Důraz je kladen na maximální automatizaci serveru a nízkou cenu jeho nasazení a údržby.

3.1 Perzistence dat

Je nutné použít více typů datových úložišť, protože aplikace stahuje a uchovává dva různé typy dat:

- data ze Zoo Praha popsaná funkčním požadavkem **SF1**,
- vektorové mapové podklady z OSM popsané funkčním požadavkem **SF2** ve formě souboru typu MBTiles.

3.1.1 Hlavní databáze

Databáze MongoDB[19] slouží jako hlavní databáze pro ukládání dat. Je z rodiny tzv. NoSQL databází, které oproti relačním databázím nepoužívají tradiční tabulkové schéma. Data jsou uložená ve formátu JSON v tzv. *dokumentech*, tyto dokumenty jsou seskupeny v tzv. *kolekcích*. MongoDB je vybráno hlavně vzhledem k jejímu jednoduchému použití v jazyce Python, vysoké efektivitě vkládání/čtení dat a velmi dobré škálovatelnosti. Jedinou malou nevýhodou jsou složitější příkazy pro sjednocování kolekcí než v relačních databázích, ale ty vzhledem k ukládaným datům nejsou tolik používány.[20]

Databáze obsahuje kolekce:

- **Animals_data**, kde každý dokument obsahuje atributy z lexikonu zvířat, které byli popsány v sekci 2.1.1. Navíc obsahuje atributy:
  - *is_currently_available*, jehož hodnota je True, pokud je zvíře v Zoo Praha v tuto chvíli chováno,
  - *map_locations*, které obsahuje informace o lokacích, kde se zvíře na mapě nachází. Jde hlavně o ID hodnoty a geografické souřadnice. Tyto jsou použity pro hledání zvířat v mapě.
- **Animal_pens**, kde každý dokument obsahuje ID zvířecího výběhu a jména zvířat, která se ve výběhu nacházejí. Jména zvířat jsou v množném a jednotném čísle, aby je bylo možné propojit s dokumenty z kolekce *animals_data*. Tato data pocházejí z mapových podkladů a slouží pro naplnění atributu *map_locations* v dokumentech kolekce *animals_data*.
- **Singular_plural**, kde každý dokument obsahuje slovo v množném čísle a jeho odpovídající tvary v jednotném čísle.
- **Zoo_parts**, kde každý dokument obsahuje ID a jméno budovy nebo části Zoo Praha, ve které se mohou zvířata nacházet. Tato data pocházejí z mapových podkladů a slouží pro naplnění atributu *map_locations* v dokumentech kolekce *animals_data*.
- **Metadata** obsahující právě jeden dokument, ve kterém jsou informace důležité pro automatizované stahování dat.
- **Roads**, kde každý dokument obsahuje předzpracovanou trasu v mapě v okolí Zoo Praha. Tato data pocházejí z mapových podkladů a společně tvoří graf cest, po kterých dokáže iOS aplikace navigovat, což vyžaduje funkční požadavek **F6**.
- **Road_nodes**, kde každý dokument obsahuje informace o jednom uzlu na trase v mapě v okolí Zoo Praha. Tato data pocházejí z mapových podkladů.
- **Zoo_houses**, jde o dokumenty pavilonů z kolekce *zoo_parts*.

3.1.2 Úložní prostor pro objekty

Amazon Simple Storage Service[21] (AWS S3) slouží primárně pro ukládání mapových podkladů ve formě souboru typu MBTiles. Jde o škálovatelný objektový úložní prostor, který jako službu nabízí cloud *Amazon Web Services (AWS)*. AWS S3 je vybrán, protože jde o jednu z nejčastějších technologií používaných pro tento účel. Jeho cena je přijatelná vzhledem k malé velikosti mapových podkladů.

3.2 Server

Vzhledem ke zkušenostem autora je server napsán v jazyce Python. Všechny požadavky na server jsou sepsány v sekci 2.6.3 a v následujících sekcích jsou podrobněji rozepsány jejich implementace.

3.2.1 Stahování OSM dat

Funkční požadavek **SF2** je implementován primárně v balíčku *scrapers* v modulu *map_downloader*, který se stará o stahování mapových podkladů v souboru MBTiles a jeho perzistentní uložení. Dále z těchto mapových podkladů získává data o umístění zvířat v Zoo Praha a ukládá je v databázi. Tyto data slouží k propojení dat ze Zoo Praha a mapových podkladů.

Jako zdroj dat pro vektorové datové podklady je použit Mapzen tile server[22], protože pochází od tvůrců knihovny `Tangram ES` použité pro zobrazování mapy v aplikaci iOS (viz. sekce 4.3.10.1). Je tak zaručeno, že stahovaná data mají správný formát. Pro samotné stahování těchto mapových podkladů je použita již existující veřejná knihovna zvaná *tilepacks*[23] od stejných tvůrců.

Pomocí knihovny *tilepacks* jsou staženy vektorové mapové podklady jak ve formátu MBTiles, tak i jako strom adresářů s GeoJSON soubory. Soubor MBTiles je okamžitě aktualizován v úložním prostoru pro objekty popsaném v sekci 3.1.2. GeoJSON soubory jsou použity pro vyhledání a uložení informací o pavilonech, částech Zoo Praha a výbězích zvířat. Tato data jsou použita pro nalezení umístění jednotlivých zvířat z lexikonu Zoo Praha v mapě. Některé pavilony mají trochu rozdílná jména v těchto uložených datech a datech z lexikonu Zoo Praha, proto je nutné na konci skriptu provést manuální úpravu těchto dat pomocí souboru *config/zoo_parts.csv*.

3.2.1.1 Příprava grafu cest

Pro splnění funkčního požadavku **F6** byl implementován skript, který z mapových podkladů ve formátu GeoJSON získá a předzpracuje všechny potřebné trasy do kolekcí *roads* a *road_nodes*, které tak tvoří výsledný graf cest, ve kterém následně iOS aplikace dokáže jednoduše vyhledávat. Je vhodné, aby tento graf předpracoval server, protože je tak ušetřen výpočetní čas na jednotlivých iOS zařízeních.

Na ukázce č. 1 je viditelná jedna cesta grafu z kolekce *roads*. Každá cesta je tvořená z uzlů, které jsou uloženy v poli `geometry.coordinates`. Pole uzlů je seřazeno tak, aby bylo možné plynule přejít z počátečního bodu do koncového bodu. Každý uzel je zároveň dokumentem v kolekci *road_nodes*. Konektor je uzel, který patří více než jedné cestě a je tak pro tyto cesty spojovacím bodem.

```
{
    "_id" : 467901842,
    "geometry" : {
        "type" : "LineString",
        "coordinates" : [
            {
                "_id" : 204519474,
                "lon" : 14.40353795,
                "lat" : 50.116612,
                "road_ids" : [
                    30000153,
                    467901842,
                    305384159
                ],
                "is_connector" : true
            },
            ...
        ]
    },
    "type" : "Feature",
    "properties" : {
        ...
    }
}
```

Listing 1: Ukázka jedné z cest z kolekce *roads*

3.2.2 Stahování dat ze Zoo Praha

Funkční požadavek **SF1** je implementován v balíčku *scrapers*, jehož moduly se starají o stahování dat ze Zoo Praha, jejich předzpracování a ukládání do hlavní databáze.

Jako zdroj dat jsou použita data z oficiální stránky Zoo Praha[1] místo původně předpokládaného zdroje OpenData hl. města Prahy[2] kvůli problémům, které jsou blíže popsány v sekci 2.1.2.

3.2.2.1 Modul zoo_scraper

Modul *zoo_scraper* slouží ke stahování dat z lexikonu zvířat ze Zoo Praha[5] pomocí techniky web scraping, která je blíže popsána v sekci 2.3.

[1] https://www.zoopraha.cz/
[2] https://opendata.praha.eu/organization/zoo

Jak je viditelné z ukázky hlavní části skriptu v bloku č. 2, nejdříve jsou z databáze získána data o umístění všech zvířat v Zoo Praha a je vytvořena dočasná kolekce, kam se stažená data z lexikonu budou ukládat. Je získán seznam konkrétních odkazů na webové stránky všech zvířat, která se v lexikonu nachází. Webové stránky na těchto odkazech jsou postupně stahovány a data z nich jsou získávána pomocí python knihovny *beautifulsoup4*[24], která slouží pro pomoc s parsováním dat z HTML a XML dokumentů. Získaná data jsou okamžitě ukládána do dočasné kolekce. Po získání dat ze všech odkazů je stará kolekce obsahující lexikon zvířat nahrazena daty z dočasné kolekce.

Je nutné, aby tento skript nezatížil server Zoo Praha příliš častými dotazy. Takové chování by v krajním případě mohlo vést k zablokování přístupu ze strany Zoo Praha, a proto je mezi dotazy umístěno čekání v rozmezí 10-20 s, které je nastavitelné pomocí systémové proměnné MIN_SCRAPING_DELAY. Vzhledem k počtu zvířat v lexikonu tak provedení celého skriptu trvá řádově 2 h až 4,5 h. Tento požadovaný čas nelze snížit a určuje jistá omezení v rámci automatizace stahování dat, která jsou vysvětlena v sekci 3.2.3.

```python
def run_web_scraper(session: requests.Session, db_handler:
↪  DBHandlerInterface,
    collection_name: str, min_delay: float = 10, **kwargs):
    animal_pens: list[dict] = db_handler.find(filter_={},
    ↪  collection_name='animal_pens')
    buildings: list[dict] = db_handler.find(filter_={},
    ↪  collection_name='zoo_parts')
    tmp_coll_name: str = f'tmp_{collection_name}'
    db_handler.update_one({'_id': 0}, {'$set':
    ↪  {'last_update_start': datetime.now()}}, upsert=True,
    ↪  collection_name='metadata')
    db_handler.drop_collection(collection_name=tmp_coll_name)

    for i, url in enumerate(get_animal_urls(session)):
        page = session.get(url.geturl())
        start_time: float = time.time()
        soup: BeautifulSoup = BeautifulSoup(page.content,
        ↪  'html.parser')

        try:
            animal_data = parse_animal_data(soup, url,
            ↪  animal_pens, buildings)
            db_handler.insert_one(animal_data.__dict__,
            ↪  collection_name=tmp_coll_name)
        except:
            logger.error(f'Error occured when parsing:
            ↪  {url.geturl()}')
            logger.error(traceback.format_exc())
            continue

        elapsed_time: float = time.time() - start_time
        time_to_sleep: float = min_delay - elapsed_time
        if time_to_sleep > 0:
            time.sleep(time_to_sleep)

    db_handler.drop_collection(collection_name=collection_name)
    db_handler.rename_collection(
        collection_new_name=collection_name,
        collection_name=tmp_coll_name)
    db_handler.update_one({'_id': 0}, {'$set':
    ↪  {'last_update_end': datetime.now()}}, upsert=True,
    ↪  collection_name='metadata')
```

Listing 2: Hlavní část skriptu pro stahování dat z lexikonu zvířat

3.2.3 Automatizace stahování dat

Funkční požadavek **SF3** je implementován v balíčku *automator*, jehož moduly *scheduler* a *worker* obstarávají automatické spuštění skriptů pro stahování dat, které byly popsány v sekcích 3.2.1 a 3.2.2. Implementace tohoto skriptu je silně provázaná s cloudem, na který je server nasazen. Jde např. o omezení času pro dokončení skriptu nebo nutnost ovlivňovat cloudové prostředí. Další informace v sekci 3.4.2.

Modul **scheduler** je hlavní řídící skript, jehož účelem je:

- Plánovat spuštění dlouhotrvajících aktualizací a delegovat je na *worker* modul. Jde hlavně o skripty pro aktualizaci lexikonu zvířat ze Zoo Praha (sekce 3.2.2.1) a aktualizaci mapových podkladů (sekce 3.2.1).

- Sám provést krátké aktualizace.

Dlouhotrvající aktualizace nemůže skript spouštět přímo, protože čas jeho dokončení může být omezen. Navíc je tento přístup pro dlouhotrvající práce obecně výhodný, pokud je lze rozdělit na více částí, které by bylo možné v cloudovém prostředí nezávisle zpracovat.

Při plánování se skript chová jako konečný stavový stroj na obrázku 3.1 s těmito stavy:

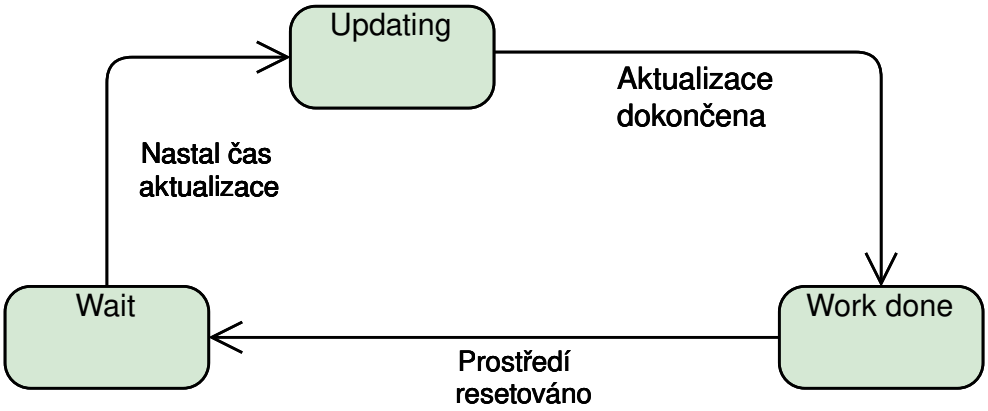
- **WAIT**, kdy skript zkontroluje, zda nastal čas spustit dlouhotrvající aktualizace. Pokud ano, spuštění skriptů je delegováno na *worker* modul a v cloudu je zapnut pracovní kontejner.

- **UPDATING**, který indikuje, že dlouhotrvající aktualizace stále probíhá.

- **WORK_DONE**, který indikuje, že dlouhotrvající aktualizace skončila. V cloudu je vypnut pracovní kontejner.

Datum aktualizací je určeno pomocí python knihovny croniter[25], která umožňuje nastavit datum nové aktualizace pomocí „cron“ výrazu uloženého v systémové proměnné CRONTAB_SCHEDULE. Je tak možné jednoduše nastavit frekvence aktualizací pomocí rozšířeného standardu, pro pomoc se specifikací výrazu lze použít i mnohé online nástroje[26]. Jako vhodná frekvence aktualizací bylo zvoleno nastavení 1x týdně v sobotu okolo 02:00.

Skript pomocí python knihovny *heroku3*[27] vypíná a zapíná pracovní kontejnery, ve kterých jsou v cloudu spuštěny dlouhotrvající aktualizační skripty. Důvody a další informace k tomuto přístupu se nacházejí v sekci 3.4.2.



Obrázek 3.1: Konečný stavový stroj pro plánování dlouhých aktualizací

3.2.4 REST API

Funkční požadavek **SF4** je implementován v balíčku *rest* pomocí webového python microframeworku *FastAPI*[28]. Existuje mnoho webových frameworků v jazyce Python, mezi nejznámější a zdaleka nejpoužívanější patří framework Django a microframework Flask[29]. Pro server je vhodnější použít microframework, protože jeho úkolem je poskytovat REST API a s dalšími funkcionalitami (např. GUI) se dle nefunkčního požadavku **SN2** nepočítá.

FastAPI bylo nakonec zvoleno místo Flask, protože jde o modernější webový microframework, který poskytuje mnoho výhod. Všechny jeho výhody/nevýhody oproti Flask jsou shrnuty v tabulce č. 3.1. Jeho největší výhodou je využívání nového ASGI[30] standardu, který mu umožňuje být jedním z nejrychlejších python web frameworků. Navíc syntaxe FastAPI je velmi podobná syntaxi Flask, a tak přechod mezi nimi není složitý.

Na URI ve tvaru `<server_address>/api/<resource>` jsou vystavena data. Tento tvar je v REST architektuře často používaný. Vzhledem k povaze dat, způsobu aktualizace dat a požadavkům iOS aplikace stačí z REST architektury implementovat nad všemi těmito zdroji pouze operace pro čtení (HTTP metoda GET). Server vystavuje data pod následujícími adresami:

- `/api/animals` – Vystavuje data všech zvířat z kolekce *animals_data*.
- `/api/animals/{animal_id}` – Vystavuje data jednoho zvířete z kolekce *animals_data*, jehož atribut *animal_id* je zadán.
- `/api/classes` – Vystavuje seznam tříd zvířat v databázi (např. Savci).
- `/api/biotops` – Vystavuje seznam biotopů zvířat v databázi.
- `/api/foods` – Vystavuje seznam typů potravy, která zvířata v databázi potřebují.

- `/api/zooHouses` – Vystavuje seznam pavilonů.
- `/api/map/data` – Vystavuje mapové podklady Zoo Praha ve formě souboru MBTiles.
- `/api/map/metadata` – Vystavuje další data o mapových podkladech Zoo Praha. Jde primárně o data z kolekcí *roads* a *road_nodes*, které společně tvoří graf cest pro navigaci.

Vzhledem k použití frameworku FastAPI dochází k automatické dokumentaci celého rozhraní. Na adrese `<server_address>/docs` je dostupná úplná dokumentace, případně je přiložena v příloze k práci jako soubor ve standardním formátu OpenAPI. V dokumentaci lze dohledat všechny existující adresy, operace nad nimi a formát výstupních dat.

Tabulka 3.1: Výhody/nevýhody FastAPI oproti Flask

Výhody	**Nevýhody**
+ Je to microframework. + Jeden z nejrychlejších python web frameworků. Podobně rychlý jako NodeJS. + Automaticky generovaná dokumentace ve formátu OpenAPI. + Podporuje asynchronní zpracování. + Podporuje jednoduché testování.	− Je poměrně nový, a proto má menší uživatelskou základnu.

3.2.4.1 Implementační detaily

V bloku č. 3 je viditelný kód jedné z funkcí frameworku FastAPI, které obsluhují požadavky klientů na dané URI. Pomocí dekorátoru `api_router.get` je určeno, že tato funkce obsluhuje cestu `/api/animals/{animal_id}` pro HTTP GET požadavky od klientů. Funkce jako vstup dostává proměnné:

- `animal_id` z tzv. „path proměnné“
- `include_currently_unavailable` z tzv. „query proměnné“
- `settings` obsahující globální nastavení

Nejdříve jsou z hlavní databáze získána metadata, následně jsou získána data o zvířeti, které vyhovuje filtru. Pokud není zvíře nalezeno, je vrácena

chybová odpověď HTTP 404, jinak jsou metadata a data zvířete zabalena do výstupní zprávy, která je poslána klientovi.

```
@api_router.get('/animals/{animal_id}',
↪  response_model=AnimalsResult)
async def animal(animal_id: int, include_currently_unavailable:
↪  bool = False, settings: SimpleNamespace =
↪  Depends(get_settings)):
    with settings.handler_class(**settings.config_data) as
    ↪  db_handler:
        metadata: dict = db_handler.find({'_id': 0},
        ↪  collection_name='metadata')[0]
        filter_ = {'_id': animal_id} if
        ↪  include_currently_unavailable else
        ↪  {'is_currently_available': True, '_id': animal_id}
        data: list[dict] = db_handler.find(filter_,
        ↪  collection_name='animals_data')
        data: list[AnimalDataOutput] = [AnimalDataOutput(**d)
        ↪  for d in data]

    if(len(data) == 0):
        raise HTTPException(status_code=404, detail="Item not
        ↪  found")

    res =
    ↪  AnimalsResult(metadata=Metadata(**metadata),data=data)
    return res
```

Listing 3: Funkce obsluhující cestu `/api/animals/{animal_id}`

3.3 Testování

Jednotlivé skripty, které server obsahuje, jsou otestovány pomocí unit testů vytvořených ve frameworku `pytest`[31]. Tento framework je jedním z nejčastěji používaných v jazyce Python.

Unit test je všeobecný typ testu, jehož účelem je testovat konkrétní funkcionalitu jedné funkce nebo třídy. Tyto testy je nutné provádět v simulovaném kontextu, kdy např. místo HTTP požadavku jsou použity data ze souboru. Je tak zajištěno, že výsledek testu je závislý pouze na testované funkcionalitě. Tento postup se nazývá *mockování* a jeho použití je blíže popsáno v následujících sekcích.

3.3.1 Testování skriptů

V bloku č. 4 je ukázka jednoho z unit testů pro skript, který slouží pro stahování dat o zvířatech z lexikonu zvířat Zoo Praha (viz. sekce 3.2.2.1). Konkrétně je testována metoda `zoo_scraper.run_web_scraper`, jejíž kód je viditelný v bloku č. 2. Problémy pro vytvoření unit testu pro tuto metodu jsou:

- Metoda posílá HTTP GET požadavky na oficiální stránky Zoo Praha.
- Je potřeba přesně určit zvířata, jejichž informace mají být staženy, aby bylo možné výsledek metody otestovat.
- Metoda používá funkci `time.sleep`, která zpomaluje test.

Posílání HTTP GET požadavku je potřeba metodě během testu zakázat, protože jinak by test nebyl spolehlivý. Bez tohoto zákazu by test byl závislý na neměnnosti cílové stránky, která není zaručena. Tento problém je vyřešen pomocí python knihovny `betamax`[32]. Tato knihovna při prvním spuštění unit testu nahraje odpovědi na HTTP GET požadavky a uloží je do souboru. Při následných spuštěních unit testu je použit tento soubor, a tak k HTTP GET požadavkům již nikdy nedojde.

Metoda `scrapers.zoo_scraper.get_animal_urls` v původní metodě určuje zvířata, jejichž informace mají být staženy, respektive je seznam těchto zvířat nalezen v lexikonu zvířat Zoo Praha. V unit testu je pomocí funkce `mocker.patch` tato metoda „nahrazena“. Způsobí, že místo původní metody je zavolána funkce, která vrátí kolekci `urls`[33]. Je tak zaručeno, že dojde ke zpracování informací o zvířatech, které unit test očekává. Stejným způsobem je odstraněna funkce `time.sleep`.

Testované metodě je na vstupu jako objekt obstarávající ukládání dat do databáze předán objekt `BaseTestHandler`, který je speciálně pro unit testy vytvořen. Zajišťuje, že výsledky jsou uloženy v kolekci `output`, jejíž obsah je následně kontrolován.

3.3.2 Testování REST API

REST API nabízené serverem lze též otestovat pomocí unit testů, protože je implementováno pomocí frameworku FastAPI, jak bylo zmíněno v sekci 3.2.4. V bloku č. 5 je unit test funkce, která obsluhuje jednu z cest. Otestování této funkce je umožněno pomocí zabudovaného testovacího klienta, který se chová stejně jako při HTTP požadavcích. V testu dochází k ověření HTTP status kódu a zda bylo vráceno správné zvíře.

FastAPI umožňuje manuálně testovat jednotlivé cesty REST API na adrese `<server_address>/docs`.

REST API lze testovat i pomocí externích nástrojů, jedním z nich je SOAPUI[34]. Tento nástroj umožňuje posílat požadavky na REST API a automa-

ticky analyzovat odpovědi. Všechny tyto nástroje je výhodné použít při testování složitějších REST API, a proto nakonec není žádný z nich použit pro testování tohoto serveru, protože jeho REST API je dostatečně jednoduché pro testování pomocí unit testů. Veškeré výhody/nevýhody použití SOAPUI oproti unit testům pro tento server jsou shrnuty v tabulce č. 3.2.

Tabulka 3.2: Výhody/nevýhody použití SOAPUI pro tento server

Výhody	**Nevýhody**
+ Rychlé vytváření testů. + Testy jsou přehlednější, hlavně u složitějších REST API. + Lze použít pro vytvoření dalších typů testů, např. performance testy. + Nástroj je obecně použitelný pro jakékoliv REST API.	– REST API serveru je velmi jednoduché. – Testování nelze automaticky spustit pomocí CI/CD, které je zmíněno v sekci 3.4.2.

3.4 Nasazení na cloud

Server a používané databáze musí být přístupné z internetu, aby je bylo možné použít. Pro tuto aplikaci je nejvhodnější použít jedno z mnoha existujících cloudových řešení.

3.4.1 Databáze

Od tvůrců databáze MongoDB existuje cloudová databáze *MongoDB Atlas*[35], které je doporučeno mnoha cloudovými prostředími jako Heroku nebo AWS[3]. Na MongoDB Atlas je databáze nasazena v tzv. „clusteru“, v jednom clusteru může být zároveň nasazeno více databází. Hlavní databáze serveru používá cluster úrovně M0, který je bezplatný za podmínky dodržování několika omezení[36]. Nabízená bezplatná verze je pro potřeby serveru dostatečná.

K nasazené databázi se lze připojit pomocí adresy, která je dostupná v informacích o clusteru. Tuto adresu lze serveru předat pomocí systémové proměnné `MONGODB_URI`. Cluster je fyzicky nasazen v regionu `eu-central-1`, který je umístěn ve Frankfurtu. Jde o geograficky nejbližší umístění k České republice.

[3] Amazon Web Services

3.4.2 Server

Pro nasazení serveru bylo vybráno cloudové prostředí *Heroku*[37]. Jde o PaaS[4] cloud založený na kontejnerech, který podporuje jazyk Python. Heroku bylo vybráno hlavně kvůli bezplatnému základnímu plánu, který nabízí. Dle tohoto plánu každý účet může měsíčně použít bezplatně až 1000 hodin, které jsou nasazenými aplikacemi čerpány, což je pro server naprosto dostačující. Základní plán má několik dalších omezení, které nejsou z pohledu serveru důležité.

Všechny nasazené aplikace běží uvnitř tzv. „dyno“ (Linuxový kontejner)[38]. Aplikace mohou používat více kontejnerů různých typů. Pro všechny své funkce potřebuje server použít 3 typy kontejnerů – *web*, *worker* a *one-off* dyno.

Ve **web dyno** běží část serveru poskytující REST API, která je popsána v sekci 3.2.4. Tento typ kontejneru se v základním plánu sám uspává, že pokud během 30 minut nedostane žádný požadavek od klientů. Při spánku nevypotřebovává bezplatné hodiny účtu, což je velká výhoda. Nevýhodou tohoto přístupu je mnohem pomalejší odezva požadavku, který kontejner probudí.

Ve **worker dyno** běží skripty pro dlouhotrvající aktualizace dat. Jak bylo zmíněno v sekci 3.2.3, skript pro automatizaci stahování dat tento kontejner zapíná v době aktualizace, jinak je vypnuté, aby zbytečně nebyly vypotřebovávány bezplatné hodiny účtu.

Ve **One-off dyno** je spouštěn skript pro automatické plánování spouštění skriptů pro stahování dat, který je zmíněn v sekci 3.2.3. Tento skript je periodicky spouštěn pomocí Heroku Scheduler[39] add-on, který je nabízen na Heroku. Tento add-on je možné používat bezplatně, ale umožňuje skript spouštět v periodách jednou za 10 min, hodinu nebo den. Pro dlouhotrvající aktualizaci dat je potřeba volit delší periodu, a proto se samotné plánování těchto aktualizací vykonává pomocí stavového stroje, který je zmíněn v sekci 3.2.3.

Pro nasazení nové verze na cloud je použit princip CI/CD[5] pipeline zprostředkovaný pomocí služby Travis CI[40]. Nová verze serveru vložená do GitHub úložiště je pomocí Travis CI automaticky otestována dostupnými testy. Je-li nová verze úspěšně otestována, pak je automaticky nasazena na Heroku.

Na adrese `https://budikpet-zoo-prague.herokuapp.com/` je dostupný nasazený server.

[4] Platform as a Service

[5] Continuous Integration / Continuous Deployment

```python
def test_run_web_scraper_basic(betamax_session: requests.Session, mocker: MockerFixture):
    sleep_time: float = 5
    get_animal_id_spy = mocker.spy(zoo_scraper, 'get_animal_id')

    # Patch the get_animal_urls function
    base = 'www.zoopraha.cz/zvirata-a-expozice/lexikon-zvirat'
    urls: list[str] = [
        f"{base}?d=643-tygr-ussurijsky&start=643",
        ...
    ]
    urls: list[ParseResult] = [urlparse(url) for url in urls]
    mocker.patch('scrapers.zoo_scraper.get_animal_urls', return_value=urls)

    # Patch time.sleep function
    unmocked_sleep = time.sleep
    sleep_lambda = lambda secs: True if (sleep_time - 2 <= secs <= sleep_time) else unmocked_sleep(secs)
    mocker.patch('time.sleep', new=sleep_lambda)

    # Act
    find_res: dict[str, list] = {
        'animal_pens': [
             {"_id": 4, "name": "tygři, tučňáci", "singular_names": ["tygr", "tučňák"]},
            ...
        ]
    }

    output: list[dict] = list()
    zoo_scraper.run_web_scraper(betamax_session, db_handler=BaseTestHandler(output, find_res), min_delay=sleep_time, collection_name="tmp")
    output: list[AnimalData] = [AnimalData(**d) for d in output]

    # Assert
    assert get_animal_id_spy.call_count == len(urls)
    assert len(output) == len(urls)

    tygr: AnimalData = next(filter(lambda animal: 'tygr' in animal.name.lower(), output), None)
    assert tygr.is_currently_available
    assert tygr.about_placement_in_zoo_prague is None
    assert tygr.location_in_zoo is None
    assert tygr.food_detail is None
    assert compare_lists(tygr.map_locations, [4])
```

Listing 4: Zkrácený unit test skriptu, který používá web scraping

```python
from fastapi.testclient import TestClient
client = TestClient(app)

def test_animal_ok():
    animals_data = [AnimalDataOutput(_id=0,
    ↪  is_currently_available=True).dict(),
    ↪  AnimalDataOutput(_id=1,
    ↪  is_currently_available=False).dict()]
    for animal_data in animals_data:
        animal_data['_id'] = animal_data.pop('id')

    find_res: dict[str, list] = {
        'metadata': [metadata],
        'animals_data': animals_data
    }
    output: list = list()
    res = {
        'handler_class': handler,
        'config_data': {
            'output': output,
            'find_output': find_res
        }
    }
    app.dependency_overrides[get_settings] = lambda:
    ↪  SimpleNamespace(**res)

    # Act
    response = client.get("/api/animals/0")
    response_data: dict = response.json()

    # Assert
    assert response.status_code == 200
    assert len(response_data['data']) == 1
    assert response_data['data'][0].get('_id') == 0
```

Listing 5: Unit test funkce obsluhující cestu `/api/animals/{animal_id}`

Kapitola 4

Implementace mobilní aplikace

Tato kapitola obsahuje návrh, implementaci a otestování mobilní aplikace pro iOS. Popisuje postup navržení uživatelského rozhraní na základě specifikovaných požadavků v sekci 2.6.1, navržení architektury aplikace, popis některých důležitých knihoven a technologií použitých pro její implementaci a následné automatické a uživatelské testování.

4.1 Návrh uživatelského rozhraní

Je důležité předem navrhnout uživatelské rozhraní aplikace (dále *UI*), je tak stanoveno, jak by měla aplikace vypadat a jak by měla implementovat jednotlivé funkcionality. Dle funkčních požadavků specifikovaných v sekci 2.6.1 jsou navrženy tyto základní obrazovky:

- interaktivní mapa,
- lexikon všech zvířat,
- detail vybraného zvířete,
- seznam filtrů, které lze na seznam zvířat aplikovat.

Interaktivní mapa a seznam zvířat jsou obrazovky první úrovně, ke kterým má uživatel přístup okamžitě po spuštění aplikace. Detail zvířete vyžaduje uživatelský vstup, proto nemůže být obrazovkou první úrovně. Účelem seznamu filtrů je upravit seznam zobrazených zvířat, proto je vhodnější tuto obrazovku umístit pod obrazovku se seznamem zvířat. Mezi obrazovkami první úrovně může uživatel volně přepínat v navigačním menu, které je v systému iOS standardně řešeno pomocí aktivního prvku zvaného „Tab Bar“ [41]. Ten je umístěn ve spodní části obrazovky, každá obrazovka první úrovně na něm má vlastní tlačítko.

4.1.1 Lexikon zvířat

Tato obrazovka implementuje funkční požadavek **F1**, je ukázána na obrázku č. 4.1. Na obrazovce je zobrazen seznam všech zvířat, která jsou k vidění v Zoo Praha. Zvířata jsou seřazena podle abecedy vzestupně. Kliknutím na položku seznamu se zobrazí obrazovka detailu vybraného zvířete popsaná v sekci 4.1.3. Každá položka seznamu zobrazuje tyto základní informace o zvířeti:

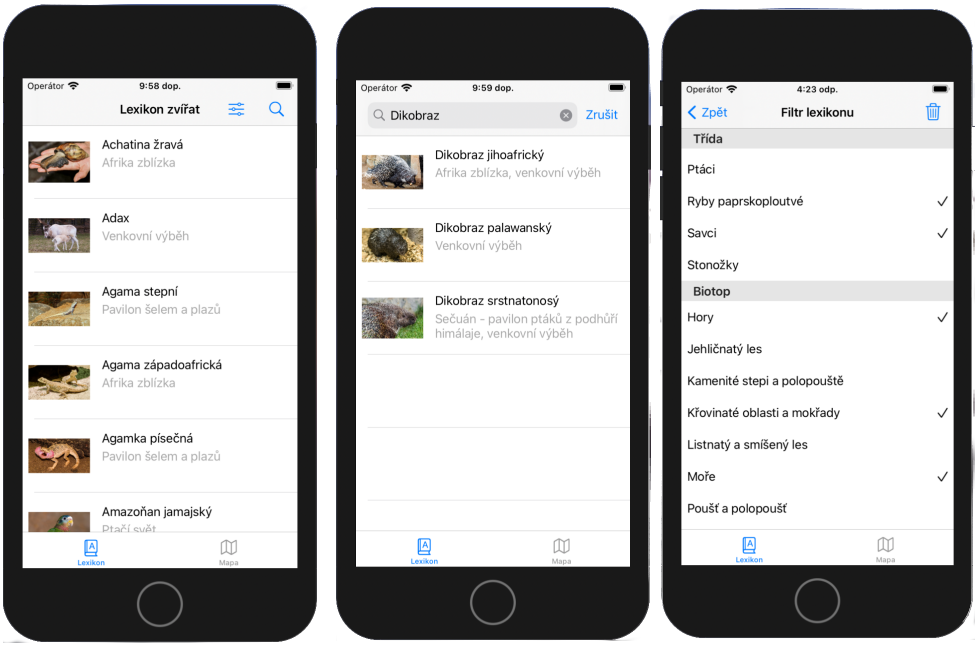
- český název zvířete,
- umístění v Zoo Praha (je-li známé),
- obrázek zvířete (pokud existuje).

Dále obrazovka implementuje funkční požadavek **F3**, umožňuje textové vyhledávání zvířat v seznamu podle jejich názvu. Při psaní je dynamicky filtrován seznam zvířat, uživatel tak okamžitě na obrazovce vidí výsledek vyhledávání. Vyhledávání je možné vyvolat pomocí kliknutí na lupu, která je umístěna v horní části obrazovky. V systému iOS je tato funkcionalita standardně řešena pomocí aktivního prvku zvaného „Search Bar“ [42].

Dále je v horní části obrazovky umístěno tlačítko pro přechod na obrazovku se seznamem filtru zvířat, která je popsána v sekci 4.1.2.

4.1.2 Seznam filtrů

Tato obrazovka implementuje funkční požadavek **F2**, je ukázána na obrázku č. 4.1. Umožňuje uživateli vybrat filtry a ovlivnit tak typy zvířat, které se objeví na obrazovce Lexikon zvířat popsané v sekci 4.1.1. Hodnoty jsou rozděleny podle typu filtru, vybraná hodnota je označena obrázkem odškrtnutí. Kliknutím na odpadkový koš v horní části obrazovky lze všechny nastavené filtry zrušit.



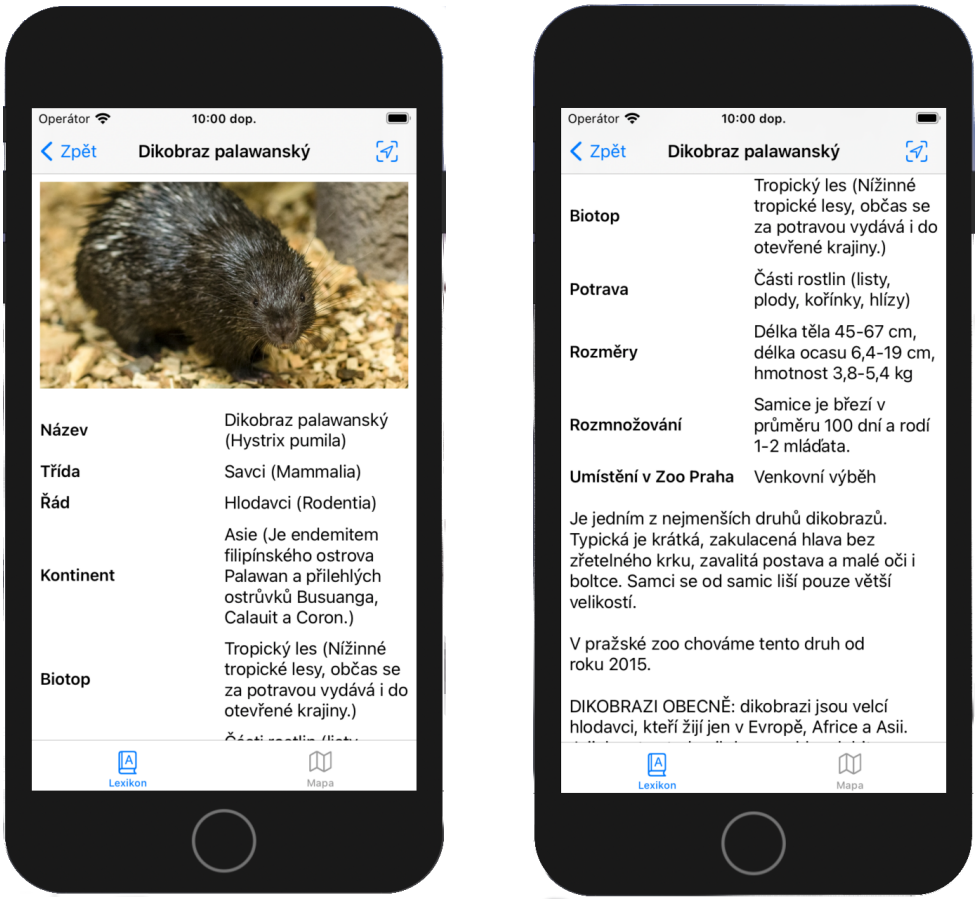
Obrázek 4.1: Lexikon zvířat (vlevo), textové vyhledávání a seznam filtrů (vpravo)

4.1.3 Detail zvířete

Tato obrazovka implementuje funkční požadavek **F4**, je ukázána na obrázku č. 4.2. Na jedné obrazovce jsou zobrazeny všechny informace o jednom vybraném zvířeti, jde o všechny atributy zvířete popsané v sekci 2.1.1.

Obrazovka začíná obrázkem zvířete, pokud obrázek pro vybrané zvíře existuje. Následně jsou zobrazeny základní údaje o zvířeti v tabulkové formě. Jde o krátké údaje typu jméno zvířete, potrava nebo kde je možné zvíře v Zoo Praha najít. Dále může být k dispozici delší text obsahující popis, zajímavosti o zvířeti nebo informace o chovu v Zoo Praha. Tento souvislý text je umístěn za základními údaji na konci obrazovky.

Z detailu zvířete může uživatel okamžitě přejít na obrazovku první úrovně s interaktivní mapou popsanou v sekci 4.1.4, pokud má vybrané zvíře specifikováno, kde v Zoo Praha se nachází. K tomuto účelu slouží tlačítko v horní části obrazovky. Po jeho stisknutí uživatel přejde na obrazovku s mapou, která zvýrazní místo/místa, kde se zvíře v Zoo Praha nachází. Následně může uživatel pracovat se zvýrazněným místem stejným způsobem, jako by ho sám v mapě označil.



Obrázek 4.2: Detail zvířete

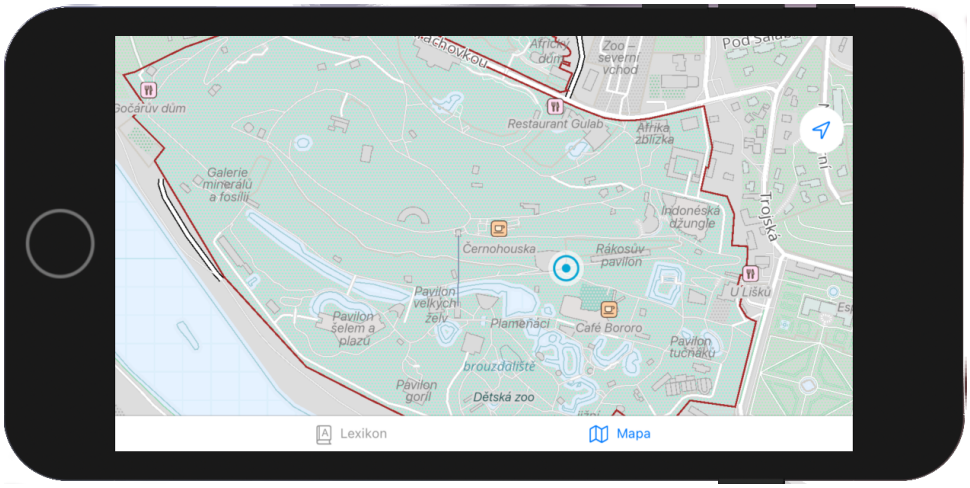
4.1.4 Interaktivní mapa

Tato obrazovka implementuje funkční požadavek **F5**, je ukázána na obrázku č. 4.3. Jde o zobrazení mapy založené na OSM datech, která je omezená pouze na areál Zoo Praha a jeho blízké okolí. Uživatel s mapou zachází stejně jako se standardními iOS mapami. V mapě je možné najít všechny venkovní výběhy zvířat, pavilony, restaurace a některé další zajímavé objekty v areálu Zoo Praha. Dále v pravém horním rohu obrazovky lze použít tlačítko pro přesun mapy na aktuální pozici uživatele.

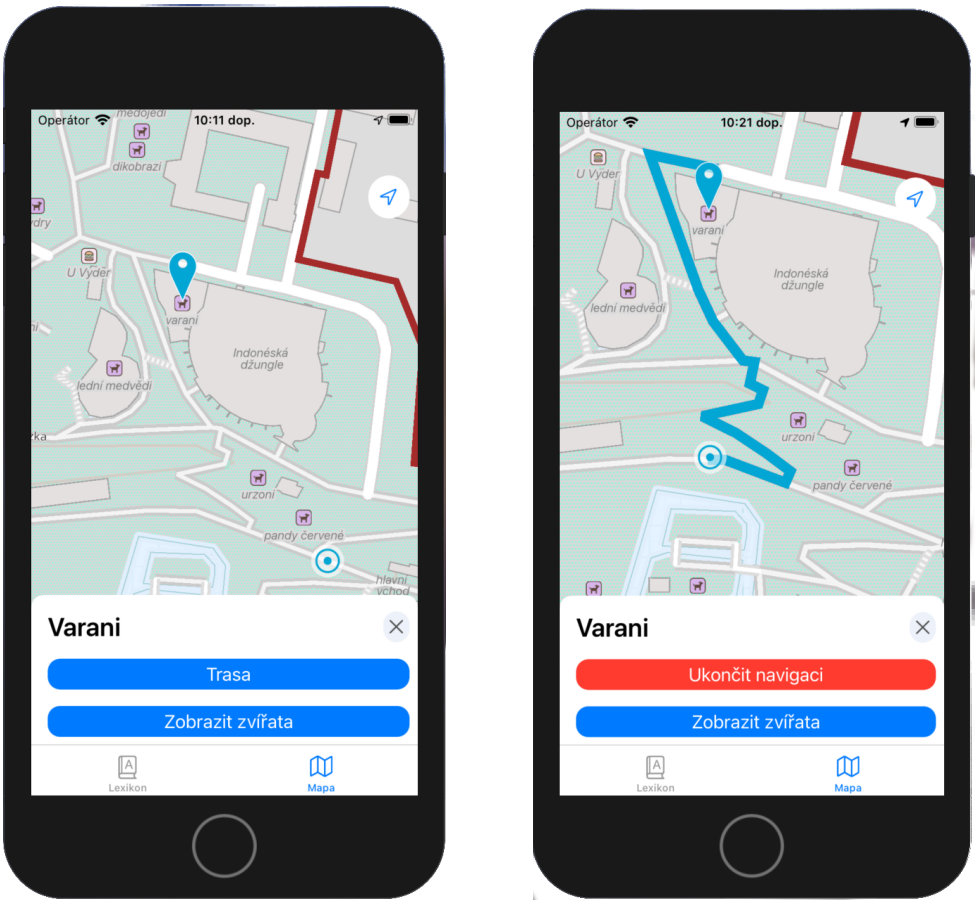
Dle funkčního požadavku **F6** mapa umožňuje označit venkovní výběhy a pavilony zvířat, jak je ukázáno na obrázku č. 4.4. Označením objektu se objeví okno, která umožňuje uživateli zobrazit trasu k označenému objektu nebo zobrazit všechna zvířata spojená s označeným objektem/objekty. Uživatel toto okno může zavřít buď kliknutím na zavírací tlačítko v pravém horním okraji okna, nebo kliknutím na jiné místo na mapě. Pokud probíhá navigace, pak je možné zavřít okno pouze zavíracím tlačítkem, dojde tak i k přerušení navigace.

Kliknutím na tlačítko „Zobrazit zvířata“ se zobrazí obrazovka detailu zvířete popsaná v sekci 4.1.3, pokud je s označeným objektem spojeno právě jedno zvíře. Jinak se zobrazí obrazovka lexikonu zvířat popsaná v sekci 4.1.1 omezená pouze na seznam zvířat spojených s označeným objektem/objekty. Oproti úplné obrazovce lexikonu zvířat tato neumožňuje filtraci ani textové vyhledávání.

Nakonec je důležité znovu zdůraznit, že mapa je omezena pouze na areál Zoo Praha a blízké okolí. Pokud se uživatel nachází mimo tento omezený prostor, tak nemůže použít navigaci k vybranému mapovému objektu nebo tlačítko pro přesun mapy na aktuální pozici uživatele. Ostatní funkcionality mapy nadále používat může.



Obrázek 4.3: Zobrazení mapy Zoo Praha



Obrázek 4.4: Zvýrazněný výběh varanů a ukázka navigace k tomuto výběhu

4.1.5 Graf přechodů

Na obrázku č. 4.5 je možné vidět graf přechodů mezi jednotlivými obrazovkami aplikace. V horní části obrázku se nachází obrazovky první úrovně lexikon zvířat a interaktivní mapa, mezi kterými je možné přepínat pomocí Tab Baru, což je zde znázorněno přechodem. Aplikace začíná na obrazovce lexikon zvířat, ze které může přejít na obrazovku se seznamem filtrů, nebo na obrazovku s detailem zvířete.

Z obrazovky s interaktivní mapou je možné zobrazit všechna zvířata spojená s označeným objektem. V případě právě jednoho zvířete je možné přejít okamžitě na obrazovku s detailem zvířete, jinak je nutné nejdříve přejít na obrazovku s lexikonem zvířat omezeným pouze na zvířata označeného objektu. Jak je z obrázku patrné, k zobrazení omezeného množství zvířat je použita obrazovka pro lexikon zvířat na druhé úrovni od obrazovky s interaktivní mapou.

Obrázek 4.5: Graf přechodů mezi obrazovkami

4.2 Návrh architektury aplikace

Na obrázku č. 4.6 je možné vidět high-level náhled architektury iOS aplikace a její propojení se serverem pomocí REST rozhraní popsaného v sekci 3.2.4. Pro uložení dat ze serveru je použita lokální databáze a úložiště souborů. Z diagramu je patrné, že celá iOS aplikace je rozdělena do dvou vrstev/komponent – *datové vrstvy* a *UI vrstvy*. Předávání dat mezi těmito dvěma vrstvami je řešeno pomocí mnoha tříd typu *ViewModel*, které jsou blíže popsány v sekci 4.2.1.

Rozdělení aplikace do více vrstev je často používané a přináší mnoho výhod:

- zvýšení přehlednosti kódu,
- mezi komponentami existují rozhraní, která:
  - umožňují jednoduché úpravy konkrétní implementace,
  - umožňují jednodušeji vytvářet automatické unit testy kvůli technice zvané *mocking*, viz. sekce 4.4.1.

- lze předejít některým chybám.

Odpovědností **UI vrstvy** je zobrazování dat a předávání interakce s uživatelem do datové vrstvy. V iOS aplikacích se v této vrstvě nachází pouze třídy zvané *ViewController*, které tvoří UI. Detailní popis *ViewControllerů* se nachází v sekci 4.3.9.2.

Odpovědností **datové vrstvy** je poskytování dat zbytku programu, získávání dat z vnějších zdrojů a ukládání těchto dat do lokální databáze a úložiště. Tato vrstva se skládá z mnoha služeb, kde každá služba má na starosti konkrétní funkcionalitu. Jedna ze služeb se stará o stažení dat z REST API poskytované serverem, který byl implementován v kapitole 3. Další služba tato stažená data zpracuje a uloží v lokální databázi a úložišti. Dále každá obrazovka zmíněná v sekci 4.1 má vlastní třídu typu *ViewModel*, ze kterého jsou obrazovce poskytována data, více v sekci 4.2.1.

Je důležité, aby dlouhotrvající operace, hlavně ty vyžadující internetové připojení, probíhaly asynchronně a neblokovaly tak hlavní vlákno aplikace. Jinak by mohlo dojít k vizuálnímu „zaseknutí“ aplikace, což si uživatel často vyloží jako chybu.

Teoreticky by bylo možné z nynější datové vrstvy oddělit vrstvu oddělit vrstvu tzv. *business logiky*. Tato nová vrstva by pak obsahovala veškerou výpočetní logiku / algoritmy a data by získávala z datové vrstvy skrz jednotné rozhraní, celkově by se tedy jednalo o třívrstvou architekturu. Toto rozdělení však není tak důležité jako oddělení UI vrstvy od zbytku aplikace, protože aplikace příliš mnoho business logiky neobsahuje.

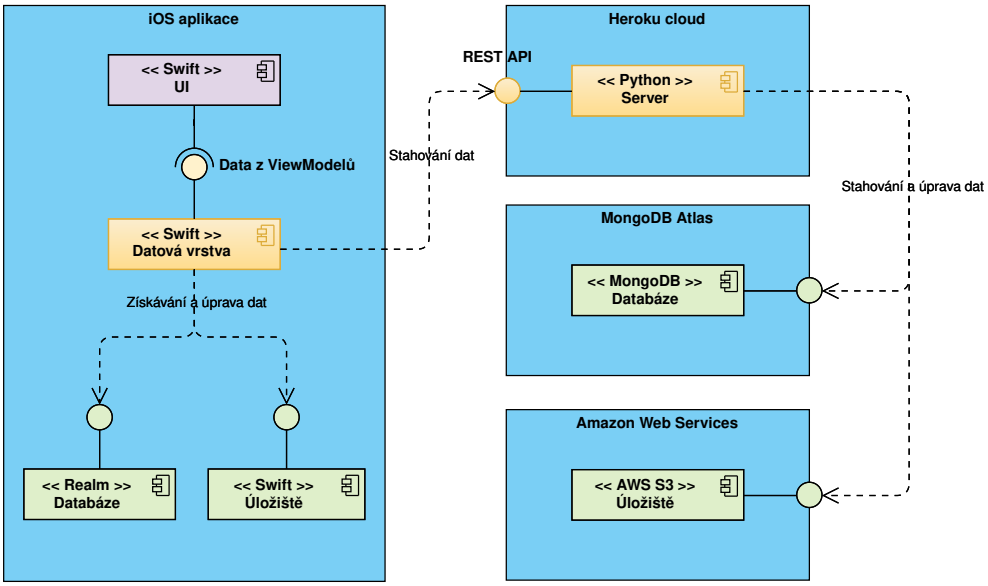
4.2.1 Architekturní vzor Model-View-ViewModel

Každá obrazovka zmíněná v sekci 4.1 je implementována pomocí vzoru `Model-ViewViewModel` (dále *MVVM*). Konkrétně v iOS aplikacích umožňuje předcházet tomu, aby veškerá logika byla umístěna přímo ve *ViewControllerech*. Proto tento vzor usnadňuje rozdělení aplikace na UI vrstvu a datovou vrstvu, které jsou popsány v sekci 4.2. Jednotlivé části tohoto vzoru jsou:

Model je součástí datové vrstvy. Jde většinou o jednoduché třídy, které drží data.

View je součástí UI vrstvy, v iOS tedy jde výhradně o třídy zvané *ViewController* popsané v sekci 4.3.9.2. Zobrazení je založeno na datech, která pocházejí výhradně z jeho vlastního ViewModelu.

ViewModel drží stav zobrazení a logiku daného View. Je tak v podstatě součástí datové vrstvy a pro každé View nejčastěji existuje právě jeden ViewModel. Jde o jednoduché třídy.



Obrázek 4.6: High-level náhled na architekturu projektu

4.3 Implementace

Tato sekce se zabývá detailním popisem implementace iOS aplikace, který je založen na návrhu UI a architektury popsaných v sekcích 4.1 a 4.2. Popisuje implementaci jednotlivých funkcionalit/částí aplikace a některé důležité knihovny, které jsou v iOS aplikaci použity. Jako základ iOS projektu byla použita upravená MVVM šablona od firmy Ackee[43], se kterou má autor aplikace dobré zkušenosti. Byl použit programovací jazyk **Swift**, který je dnes standardně používán při programování iOS aplikací pro iOS, iPadOS, MacOS apod.

4.3.1 Management závislostí

Závislostmi jsou v tomto případě myšleny knihovny, které aplikace při implementaci využívá. Tyto závislosti jsou obsluhovány tzv. „Dependency manažery“, v případě iOS aplikací mezi nejznámější patří *Cocoapods*[44] a *Carthage*[45].

Cocoapods je starším Dependency manažerem, který je používán téměř všemi knihovnami a nastavení závislostí je více automatizované než u *Carthage*. Jeho nevýhodou je nutnost transformovat základní iOS projekt tak, aby bylo možné knihovny připojit. Dále je během programování častěji sestavovat jednotlivé knihovny, což může znatelně zvýšit čas sestavení celé aplikace.

Carthage je méně automatizované než *Cocoapods*, knihovny jsou staženy a sestaveny, ale jejich zapojení do projektu musí provést programátor sám.

Jeho výhodou je jednoduchost a zvýšení rychlosti sestavení aplikace, protože sestavené statické knihovny není třeba znovu sestavovat. Bohužel ne všechny knihovny ho podporují, což je jeho největší nevýhoda.

Tato aplikace používá oba Dependency manažery najednou. Cocoapods je použit pouze pro knihovny, které Carthage přímo nepodporují.

4.3.2 Reaktivní programování

Před popisem implementace jednotlivých funkcionalit je vhodné zmínit knihovnu `ReactiveSwift`[46], která je použita na mnoha místech celé iOS aplikace. Tato knihovna umožňuje využít principu tzv. *Funkcionálního reaktivního programování (FRP)*[47]. FRP je založeno na práci s asynchronními datovými toky reprezentující změny hodnot v čase. Tyto změny jsou v programu pozorovány (*observe*), což znamená, že nějaká funkcionalita je provedena pouze při změně hodnot v toku. Pozorování změn v hodnotách je využito např. k propojení ViewModelů a ViewControllerů, jak je popsáno v sekci 4.3.9.

Vzhledem ke zkušenostem autora je pro implementaci aplikace vybrána právě knihovna *ReactiveSwift*. Pro iOS existuje několik dalších knihoven umožňujících využít princip FRP, mezi nejznámější patří *Combine*[48] od firmy Apple nebo *RxSwift*[49].

4.3.3 Dependency injection

4.3.3.1 Obecné informace k této technice

Pokud třída *A* pro vykonání svého účelu potřebuje použít jinou třídu *B*, lze třídu *B* označit jako *závislost třídy A*. Třída *A* potřebuje vlastnit instanci třídy *B*. *Dependency injection* (*vkládání závislostí*, dále *DI*) je technika vkládání těchto závislostí do jednotlivých tříd. V DI jsou vytvářeny instance závislostí na jednom místě a zbytku programu jsou předávány. Tento postup má mnoho výhod:

- Pro celý program jsou vytvářeny instance závislostí pouze jednou.
- Zjednodušené vytváření závislostí.
- Závislosti jsou do tříd vkládány. Lze tak ovlivnit, jaká konkrétní implementace je třídě předávána, což například umožňuje dělat *mocking* (viz. sekce 4.4.1).

4.3.3.2 Služby

V iOS aplikaci pomocí *DI* „uskladněny“ třídy, které jsou souhrnně nazvány `Služby`. Je implementováno několik služeb, které vykonávají různé činnosti, příkladem je služba pro získávání dat ze serveru zmíněná v sekci 4.3.5. Jde o jednoduché třídy, kde každá má přesně definované užití. V bloku č. 6 se nachází ukázka obecné struktury služeb.

4.3.3.3 Implementace DI v iOS aplikaci

DI je v iOS aplikaci implementováno pomocí skládání protokolů[50]. V bloku č. 6 je ukázka služby, se kterou se má pracovat pomocí techniky *DI* a která zároveň vyžaduje právě jednu závislost pomocí protokolu `HasRealm`. Všechny závislosti jsou uloženy v singletonu `AppDependency`, ukázka takové třídy je v bloku č. 7. *AppDependency* nyní obsahuje instance dvou závislostí, služba *ExampleService* také vyžaduje závislosti, které jsou jí předány při inicializaci.

Existují knihovny, které *DI* implementují. Postup skládání protokolů však nepotřebuje další knihovnu, je jednoduchý a pro tuto aplikaci naprosto dostačující.

```
1  // Protokol, pomocí kterého lze v dalších službách
   ↪  specifikovat, že má jako závislost službu ExampleService
2  protocol HasExampleService {
3      var exampleService: ExampleServicing { get }
4  }
5
6  // Protokol (rozhranní) služby
7  protocol ExampleServicing {
8      func doSomething() -> Int
9  }
10
11 // Konkrétní implementace služby
12 final class ExampleService: ExampleServicing {
13     // Služba má právě jednu závislost
14     typealias Dependencies = HasRealm
15     private let realm: Realm
16
17     init(dependencies: Dependencies) {
18         self.realm = dependencies.realm
19         ...
20     }
21
22     // Nějaká funkce, kterou služba musí poskytovat
23     func doSomething() -> Int {
24         ...
25     }
26 }
```

Listing 6: Ukázka obecné struktury služeb

```
1  /// Kontejner pro všechny závislosti v aplikaci
2  final class AppDependency: HasRealm, HasExampleService {
3      lazy var realm: Realm = self.initRealm()
4      lazy var exampleService: ExampleServicing =
       ↪  ExampleService(dependencies: self)
5  }
6
7  let appDependencies = AppDependency()
```

Listing 7: Ukázka třídy s instancemi závislostí

4.3.4 Jazyková lokalizace

Aplikace samotná je lokalizována pro český a anglický jazyk. Pro automatizaci, zpřehlednění a zjednodušení vytváření této lokalizace je využito knihoven `SwiftGen` a `ACKLocalization`.

ACKLocalization[51] není knihovna používaná přímo v iOS aplikaci. Tato knihovna umožňuje stahovat data o lokalizacích z online souboru typu Excel, jehož tvar je ukázán v tabulce č. 4.1. Následně data zpracuje do souboru *localizable.strings*, což je standardní formát lokalizace pro iOS zařízení.

Tento přístup přináší mnoho výhod. Umožňuje dělat jednotnou lokalizaci pro zařízení s různými operačními systémy nezávisle na sobě, tedy např. i pro OS Android. Pouze musí existovat podobná knihovna, která dokáže data z Excel tabulky zpracovat. Také umožňuje, aby na lokalizaci pracovalo najednou více lidí.

SwiftGen[52] slouží pro zpracování vytvořeného souboru *localizable.strings* do enumerací, která jsou následně používána na všech místech v iOS aplikaci. Bez této knihovny by bylo pro použití lokalizovaného textu vždy nutné specifikovat klíč ze sloupce *key_ios*, což může vést ke špatně dohledatelným chybám. Stejným způsobem lze vytvořit enumerace pro některé další zdroje.

Bohužel všechna data ze Zoo Praha specifikovaná v sekci 2.1.2 existují pouze v českém jazyce, a proto je primárně užívána česká lokalizace. Avšak je vhodné tento systém lokalizace vytvořit, protože v případě existence dat v dalších jazycích je jednoduché lokalizaci doplnit.

key_ios	**EN**	**CS**
viewAnimals	View animals	Zobrazit zvířata
...	...	...

Tabulka 4.1: Ukázka Excel tabulky s lokalizací

4.3.5 Komunikace s API serveru

Aplikace získává data ze serveru implementovaného v kapitole 3 pomocí REST rozhraní popsaného v sekci 3.2.4. Pro posílání požadavků a zpracování odpovědí je použita knihovna `Alamofire`[53]. Tato knihovna obecně slouží ke zpracování HTTP komunikace a je v iOS aplikacích velmi často používána. Internetová komunikace může být nespolehlivá a dlouhotrvající, a proto musí být stahování dat provedeno asynchronně. Toho je docíleno převedením výsledku z knihovny Alamofire na asynchronní datový tok pomocí knihovny *ReactiveSwift*.

Pomocí služby *ZooAPIService* jsou implementovány funkce pro zpracování dat ze všech endpointů REST rozhraní, jedna z těchto funkcí je ukázána v bloku č. 8. Jde o jednoduchou funkci, která využívá další služby *JSONAPIService* a *Network* pro zaslání obecného požadavku na REST rozhraní a zpracování výsledku ve formátu JSON, který je vracen v asynchronním datovém toku. JSON data v toku jsou ve funkci přeměněna na *Detached** struktury (viz. sekce 4.3.6), upravený datový tok je funkcí vrácen pro další zpracování.

```
1  func getAnimals() -> SignalProducer<(DetachedMetadata,
   ↪  [DetachedAnimalData]), RequestError> {
2      os_log("Fetching all animals.", log:
       ↪  Logger.networkingLog(), type: .info)
3      return jsonAPI.request(path: "/api/animals").compactMap {
       ↪  response in
4          let responseData = response.data
5          let metadataDict = responseData["metadata"]
6          let animalDict = responseData["data"]
7
8          let metadata: DetachedMetadata =
           ↪  DetachedMetadata(using: metadataDict)
9          let animalData = animalDict.map() {
           ↪  DetachedAnimalData(using: $0) }
10         return (metadata, animalData)
11     }
12 }
```

Listing 8: Zjednodušená funkce pro zpracování požadavku na stažení všech zvířat z REST API

4.3.6 Perzistence dat

Vzhledem k nefunkčnímu požadavku **N2** aplikace potřebuje lokálně ukládat data z externí MongoDB databáze zmíněné v sekci 3.1.1. Jako lokální data-

báze je použita knihovna `RealmSwift`, která má o něco lepší vlastnosti než zabudovaná knihovna *Core Data* od firmy Apple. Je rychlejší a lépe se s ní pracuje.

Dále aplikace potřebuje ukládat soubor typu MBTiles zmíněný v sekci 2.1.3.2, ve kterém jsou uloženy podklady pro zobrazení interaktivní mapy. K tomu je využito lokální úložiště aplikace.

4.3.6.1 Modely

Data z databáze jsou používána v ostatních částech aplikace pomocí modelů. V bloku č. 9 je ukázán model cesty z grafu cest, který je zmíněn v sekci 3.2.1.1. Jak je z kódu vidět, pro data z databáze reprezentující cestu existuje třída `Road` a struktura `DetachedRoad`. Tímto způsobem je vytvořena většina modelů. Oba obsahují stejná data, jejich použití se však liší:

- **DetachedRoad** je struktura, do které se vkládají data získaná ze serveru (viz. sekce 4.3.5). Dále je se s touto strukturou jednodušeji pracuje v datových tocích.
- **Road** je Realm model, pomocí kterého se ukládají a upravují data v lokální databázi. Dále je možné na těchto objektech pozorovat změny v lokální databázi, proto je vhodné je přímo použít při zobrazování UI.

```
// Realm model
class Road: Object {
    @objc dynamic public var _id: Int64 = -1
    public let nodes = List<RoadNode>()

    override public static func primaryKey() -> String? {
        return "_id"
    }

    public convenience init(_ detachedRoad: DetachedRoad) {
        self.init()
        self._id = detachedRoad._id

        for node in detachedRoad.nodes {
            self.nodes.append(RoadNode(node))
        }

    }
}

// Jednoduchá struktura
struct DetachedRoad {
    let _id: Int64
    var nodes: [DetachedRoadNode] = []

    // Sestavení struktury z JSONU
    init(using dict: [String: Any]) {
        self._id = dict["_id"] as! Int64

        ...
    }

    init(using road: Road) {
        self._id = road._id
        for node in road.nodes {
            self.nodes.append(DetachedRoadNode(using: node))
        }
    }
}
```

Listing 9: Model cesty z grafu cest

4.3.7 Aktualizace dat

Služba `StorageManager` zbytku aplikace poskytuje funkce, které zjednodušují některé úkony s Realm databází a lokálním úložištěm. Mezi nejdůležitější úkoly patří aktualizace dat v databázi a lokálním úložišti.

Při aktualizaci využívá upravené asynchronní datové toky ze služby *ZooAPIService* popsané v sekci 4.3.5. Protože server aktualizuje data v pravidelných intervalech (viz. sekce 3.2.3), tak nejdříve služba *StorageManager* zjistí, zda se nová data na serveru nachází. Je-li aktualizace nutná, pak naváže na datové toky ze služby *ZooAPIService*, transformuje jednotlivé *Detached** struktury na Realm modely, které uloží do lokální databáze.

V aplikaci jsou také přibalena všechna data, která aplikace stahuje ze serveru a používá v aplikaci. Tato data jsou použita při prvním spuštění aplikace, než jsou stažena aktuální data. Je tak zajištěno, že aplikace bude fungovat, pokud nebude server v danou chvíli dostupný. Jde tak o další krok k zajištění nefunkčního požadavku **N2**.

4.3.8 Navigace v interaktivní mapě

Obrazovka s interaktivní mapou zmíněná v sekci 4.1.4 umožňuje navigaci uživatele k vybranému objektu (funkční požadavek **F6**). Samotné zobrazení navigace v mapě je řešeno v sekci 4.3.10.1. V této sekci je popsána služba `ZooNavigationService`, jejíž účelem je vyhledat nejkratší trasu mezi dvěma body v mapě. Využívá k tomu připravený graf cest zmíněný v sekci 3.2.1.1.

Vstupem služby jsou dva body v mapě, každý reprezentovaný zeměpisnou šířkou a délkou. Výsledkem je pole bodů v mapě, které tvoří trasu od pozice uživatele k pozici cíle. Algoritmus probíhá v několika krocích:

1. Je nalezena cesta, na které se nachází uživatel a nejbližší cesta u cílového bodu.
2. Na uživatelově/cílové cestě je nalezen bod, který leží na cestě a je nejbližším bodem vzhledem k uživatelově/cílové pozici. Toto je nutné, protože pozice uživatele/cíle se nachází mimo cestu. Jde o **počáteční/koncový bod**.

    a) Je-li počáteční a koncový bod na stejné cestě, jsou okamžitě propojeny a je vrácen výsledek.

3. Na dvou nalezených cestách jsou nalezeny všechny spojovací body[6].
4. Pomocí algoritmu A* je nalezena nejkratší trasa mezi jedním ze spojovacích bodů na uživatelově cestě a jedním ze spojovacích bodů na cílové cestě. Je tvořena pouze spojovacími body.

[6]Spojovací bod je bod, který je součástí alespoň dvou cest.

5. Mezi jednotlivými spojovacími body nalezené trasy jsou doplněny ostatní, nespojovací body.
6. Počáteční a koncový bod je napojen na nalezenou trasu, je vrácen výsledek.

Nalezení nejbližší cesty je provedeno nalezením nejbližšího uzlu k pozici uživatele/cíle. Nalezení počátečního a koncového bodu na nejbližších cestách je provedeno pomocí knihovny `Turf for Swift`. Jde o knihovnu pro prostorovou analýzu, což je zjednodušeně řečeno metoda sloužící k analýze geografických dat.

A* [54] je efektivní algoritmus pro nalezení cesty mezi dvěma body, v bloku č. 10 je ukázka zde použitého algoritmu v pseudokódu. Dá se říct, že jde o Dijkstrův algoritmus[54] vylepšený o heuristiku. Algoritmus je iterativní, ukládá si informace o již navštívených (otevřených) a uzavřených uzlech. V každé iteraci vybere z otevřených uzlů uzel s nejnižší hodnotou, kde výsledná hodnota je součtem těchto proměnných:

- *Vzdálenost od počátku* je délka nejkratší trasy mezi aktuálním a počátečním uzlem. Je tedy o opravdovou vzdálenost od počátečního uzlu. Tato délka je nastavována při navštívení uzlů.
- *Přibližná vzdálenost k cíli* je vzdálenost „vzdušnou čarou“ mezi aktuálním uzlem a cílem. Toto je heuristická část hodnoty, která odlišuje algoritmus A* od Dijkstrova algoritmu.

Každý uzel si tak musí pamatovat svoji přibližnou vzdálenost k cíli, vzdálenost od počátečního uzlu a předchozí uzel na nejkratší cestě k počátečnímu uzlu. Iterace je ukončena ve chvíli, kdy je z otevřených uzlů vybrán jeden z koncových uzlů. V tu chvíli je možné sestavit výslednou nejkratší cestu mezi počátečním a koncovým uzlem.

```
func computeShortestPath(origins: Set, destinations: Set) ->
↪  List {
    openedNodes: Set = origins
    closedNodes: Set = emptySet

    while(opened is not empty) {
        // Vyhledání otevřeného uzlu s nejnižší hodnotou
        currentNode: Node = openedNodes.nodeWithLowestValue

        // Uzavření vyhledaného uzlu
        openedNodes.remove(currentNode)
        closedNodes.add(currentNode)

        if(currentNode is in destinations) {
            // Cesta nalezena, vytvoř pole uzlů
            return constructPath(from: currentNode)
        }

        // Najdi všechny spojovací uzle, které sousedí s
        ↪  vyhledaným uzlem

        for neighbourNode in currentNode.neighbourNodes {

            if(origins contains neighbourNode) {
                continue
            } else if(closedNodes contains neighbourNode) {
                // Soused byl již uzavřen
                continue
            }

            if(openedNodes contains neighbourNode) {
                if(current value of neighbour node is better) {
                    // Byla nalezena lepší trasa do sousedního
                    ↪  uzlu, je potřeba aktualizovat jeho
                    ↪  hodnotu
                    update values of neighbourNode
                }
            } else {
                // Sousední uzel byl otevřen
                compute values of neighbourNode
                openedNodes.add(neighbourNode)
            }
        }
    }

    // Trasa nebyla nalezena
    return nil
}
```

Listing 10: Popis použitého algoritmu A*

4.3.9 Uživatelské rozhraní

V této sekci jsou obecně popsány způsoby implementace UI a blíže popsány pojmy *ViewController*, *ViewModel* a *Flow koordinátor*. Na základě těchto informací jsou v sekci 4.3.10 implementovány jednotlivé obrazovky.

4.3.9.1 Implementace uživatelského rozhraní v iOS

UI v této aplikaci je vytvořeno **přímo v kódu** v jazyce Swift. Programátor pro jednotlivé obrazovky sám vytvoří instance jednotlivých grafických prvků, které má obrazovka obsahovat. Tyto prvky jsou v obrazovce uspořádány na základě omezení (*constraints*), které programátor též musí specifikovat, k tomu slouží pomocná knihovna `SnapKit`. Jednotlivé obrazovky jsou propojeny pomocí tzv. *Flow koordinátoru*, který je popsán v sekci 4.3.9.4. Zde je k vidění ukázka kódu pro vytvoření UI:

```
let tableView = UITableView()
tableView.dataSource = self
tableView.delegate = self

tableView.register(CellVC.self, forCellReuseIdentifier: "cell")
view.addSubview(tableView)
self.tableView = tableView

// Nastavení omezení pro tableView.
// Zde je řečeno, že tableView zaplňuje celou obrazovku.
tableView.snp.makeConstraints { (make) in
    make.edges.equalTo(self.view)
}
```

Dalšími možnostmi tvoření UI jsou **Storyboards**[55] a **SwiftUI**[56]. Jde o nástroje vytvořené firmou Apple, které jsou zabudované ve vývojovém prostředí Xcode. Oba tyto nástroje umožňují vytvářet UI a okamžitě pozorovat změny bez nutnosti zapínat celou aplikaci, což je oproti přímému tvoření UI v kódu velká výhoda. Storyboards je starší nástroj, ve kterém je každá obrazovka vytvořena jako XML soubor, který je následně aplikací použit pro sestavení UI za běhu. Je tak méně flexibilní a u větších aplikací i méně přehledný, proto není použit. SwiftUI je naopak velmi nový nástroj, který je bližší vytváření UI přímo v kódu. Není použit, protože na rozdíl od předchozích dvou způsobů vytváření UI s ním zatím nemá autor dostatečné zkušenosti.

4.3.9.2 ViewController

ViewController je třída reprezentující jednu obrazovku zobrazenou na displeji zařízení, proto pro každou obrazovku zmíněnou v sekci 4.1 existuje právě jeden ViewController.

Aplikace pro vytváření obrazovek používá architekturní vzor *MVVM* (viz. sekce 4.2.1), kde ViewController spadá do části *View*, a proto se v této aplikaci stará čistě o zobrazování a interakci s uživatelem. V kódu těchto tříd je vytvořeno UI způsobem popsaným v sekci 4.3.9.1. Veškeré výpočetní úkony či změny ve zobrazení provedené na základě interakce s uživatelem jsou delegovány na ViewModel daného ViewControlleru.

ViewController se řídí tzv. *životním cyklem*. Během svého života prochází různými stavy podle toho, jak uživatel prochází aplikací. Obsahuje proto funkce, které se provádí v závislosti na průchodu těmito stavy. Tyto funkce může programátor přetížit a specifikovat kód, který se musí v těchto stavech provést. Těmito akcemi může být např. zastavení probíhajících animací při přepnutí na jinou obrazovku nebo uvolnění paměti.

4.3.9.3 ViewModel

ViewModel je jednoduchá třída, ve které je držen „stav zobrazení“ pro daný ViewController. Také jsou zde prováděny veškeré výpočetní úkony vyplývající z interakce s uživatelem, a proto ViewModel obsahuje veškeré závislosti dané obrazovky nabízené skrz *DI*, jak je zmíněno v sekci 4.3.3.

ViewController získává stav zobrazení pomocí reaktivního pozorování hodnot ve ViewModelu, jak je zmíněno v sekci 4.3.2. Využívá k tomu již zmíněnou knihovnu `ReactiveSwift`, pro propojení datových toků s UI používá knihovnu `ReactiveCocoa`[57]. Použitím těchto dvou knihoven pak lze dosáhnou velmi přehledného propojení změn stavu ViewModelu se zobrazením ve ViewControlleru. Toto propojení se nazývá „*binding*“, který je ukázán na následujícím kódu:

```
// Propojení hodnoty se stavem akce
filterItem.reactive.isEnabled <~ viewModel.updateDB.isExecuting
searchItem.reactive.isEnabled <~ viewModel.updateDB.isExecuting
                                    .map() { return !$0 }
```

Znak `<~` značí, že reaktivní proměnná nalevo je napojena na datový proud napravo. Proměnné `filterItem` a `searchItem` jsou instancemi grafických prvků ve ViewControlleru, část *reactive* je těmto instancím dána knihovnou `ReactiveCocoa` a *isEnabled* je reaktivní proměnná, jejíž hodnota nyní bude určena hodnotami v datovém toku. Proměnná *updateDB* je akce z knihovny `ReactiveSwift`, zde je zjišťováno, zda je tato akce zrovna spuštěna, což je datový tok hodnot true/false.

4.3.9.4 Flow koordinátor

Flow koordinátor je jednoduchá třída, která zajišťuje přechody mezi jednotlivými obrazovkami. Pouze ve Flow koordinátorech dochází k inicializaci ViewControllerů a ViewModelů jednotlivých obrazovek, což se týká i obrazovek

první úrovně. Jde tedy v podstatě o implementaci grafu přechodů zmíněném v sekci 4.1.5, který je možné vidět na obrázku č. 4.5.

Pro navigaci mezi obrazovkami první úrovně je použit tzv. *Tab bar*, jak je zmíněno v sekci 4.1, a proto hlavní Flow koordinátor musí tento Tab bar implementovat. K tomu slouží enum třída `TabBarPage`, ve které jsou specifikovány jednotlivé části Tab Baru a funkce zjednodušující jeho implementaci ve Flow koordinátoru.

Přechod mezi obrazovkami je následně řešen delegací Flow koordinátoru. Pokud z obrazovky lze přejít na jinou obrazovku, pak ViewController této obrazovky specifikuje protokol s funkcemi popisujícími tyto přechody. Následně Flow koordinátor tento protokol implementuje.

4.3.10 Implementace obrazovek uživatelského rozhraní

Implementace uživatelského rozhraní je založena na návrhu, který je zmíněn v sekci 4.1. V této sekci jsou popsány implementace konkrétních obrazovek.

4.3.10.1 Interaktivní mapa

Pro zobrazení interaktivní mapy podle návrhu ze sekce 4.1.4 je použita knihovna `Tangram ES`[58]. Tato knihovna funguje podobně jako zabudovaná knihovna *MapKit* od firmy Apple, ale používá mapové podklady z OpenStreetMap zmíněné v sekci 2.1.3. V aplikaci je nutné použít tyto mapové podklady, protože jak již bylo v sekci 2.1.3 zmíněno, pro Zoo Praha jsou mnohem přesnější a navíc obsahují lokace venkovních výběhů. Ukázku této mapy je možné vidět na obrázku č. 4.3.

Tangram ES je jedna z mála knihoven, která používá jako základ OSM data, je open-source a zároveň nepotřebuje internetové připojení, respektive dokáže číst lokálně uložený soubor s mapovými podklady typu MBTiles. Způsob zobrazení mapy je plně nastavitelný pomocí přibaleného konfiguračního souboru `bubbleWrapStyle.yaml`, tzv. *soubor scény*. Programátor může dle dokumentace[59] tento soubor měnit a významně tak změnit vizuální podobu mapy. Aplikace používá upravený soubor scény *BubbleWrap*[60]. Mapa je pro tuto aplikaci omezena pouze na zobrazování areálu Zoo Praha a blízkého okolí.

Mapu je možné aktualizovat za běhu programu, což je využito při zvýraznění mapových objektů a navigaci k nim. Toho je docíleno pomocí systému vrstev (layers) specifikovaných v souboru scény. Každá vrstva má specifikovaný zdroj geografických dat, pro jaké typy dat má být použita a v podstatě určuje, jakým způsobem jsou tato data v mapě zobrazena. Vrstvy zobrazující stále viditelné prvky (cesty, domy, řeky...) mají jako zdroj dat specifikovaný soubor MBTiles. Pro interaktivní vrstvy jsou zdroje dat naplňovány za běhu, mapa je následně zobrazí. Ukázku zvýraznění a navigace je možné vidět na obrázku č. 4.3. V případě zvýraznění budovy je i zvýrazněna celá budova.

Práce s mapou je z pohledu uživatele blízká standardním iOS mapám, knihovna nabízí všechny typy gest. V aplikaci jsou implementována následující gesta:

- Tažení jedním prstem pro pohyb v mapě.
- Dvojité kliknutí pro přiblížení/oddálení mapy.
- Jedno kliknutí pro zvýraznění mapového objektu, jde-li o pavilon nebo zvířecí výběh. Jinak zruší zvýraznění. Pokud zrovna probíhá navigace, tak jedno kliknutí nedělá nic.

Na obrazovce se kromě mapy nachází tlačítko, které způsobí přesun a vycentrování mapy na uživatelovu pozici. Je deaktivováno, pokud se uživatel nachází mimo mapu nebo nedal souhlas s GPS lokalizací.

Zvýraznění objektu v mapě způsobí zobrazení informačního panelu, což je grafický prvek implementovaný třídou `MapOptionsPanelView`. Je viditelný na obrázku č. 4.4. Umožňuje zapnutí/vypnutí navigace a zobrazení všech zvířat, která jsou spojená s mapovým objektem nebo objekty. Je-li zvýrazněn jeden objekt, pak v nadpisu je jméno objektu, jinak je vypsán počet zvýrazněných objektů.

Jak je v návrhu v sekci 4.1.4 specifikováno, po kliknutí na tlačítko „Zobrazit zvířata“ obrazovka v závislosti na počet zvířat přejde buď na obrazovku s detailem zvířete, nebo na obrazovku se seznamem zvířat. Implementačně se jedná o zjednodušené verze obrazovek Detail zvířete (viz. sekce 4.3.10.3) a Lexikon zvířat (viz. sekce 4.3.10.2), které neobsahují navigační tlačítka pro přesun do mapy, filtraci či vyhledávání v seznamu. Graficky však vypadají úplně stejně.

Tlačítko navigace je deaktivováno, pokud se uživatel nachází mimo mapu, pokud nedal souhlas s GPS lokalizací nebo pokud je zvýrazněn více než jeden mapový objekt. Po zapnutí navigace je vypočítána nejkratší cesta mezi pozicí uživatele a zvýrazněným objektem, tuto funkcionalitu poskytuje služba zmíněná v sekci 4.3.8. Navigaci je možné přerušit buď kliknutím na tlačítko „Ukončit navigaci“, nebo uzavřením celého panelu.

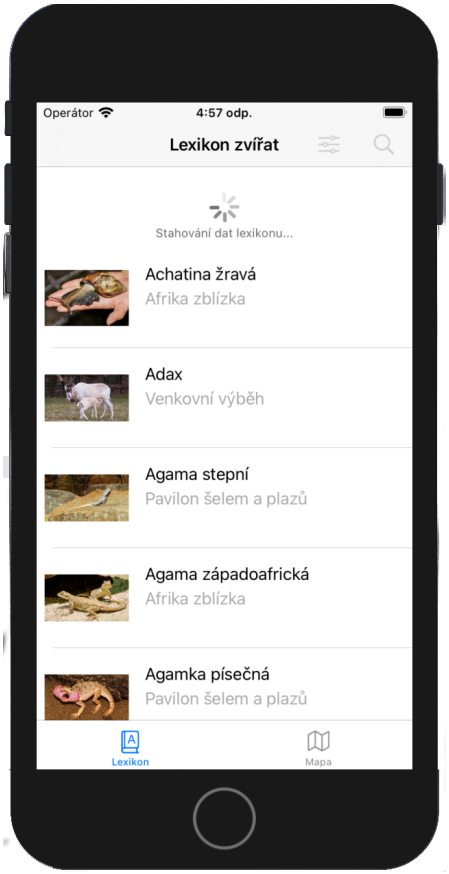
4.3.10.2 Lexikon zvířat

Obrazovka zobrazující seznam zvířat z lexikonu zvířat dle návrhu ze sekce 4.1.1. Pro zobrazení seznamu je použit standardní grafický prvek *TableViewController*, který používá položky implementované ve třídě `LexiconItemCellVC`. Každá buňka zobrazuje jméno zvířete, jeho obrázek a umístění v Zoo Praha. Po kliknutí na položku seznamu se zobrazí obrazovka s detailem vybraného zvířete, jejíž implementace je popsána v sekci 4.3.10.3.

Protože aplikace používá `Realm` databázi (viz. sekce 4.3.6, je možné jednoduše pozorovat (*observe*) změny v datech a v případě těchto změn měnit zobrazení.

V horní části obrazovky v navigačním panelu se nachází tlačítka pro spuštění vyhledávání v seznamu a přechod na obrazovku s filtry, jak je ukázáno na obrázku č. 4.1. Vyhledávání v seznamu je implementováno pomocí standardního grafického prvku `UISearchBar`[42]. Při vyhledávání jsou okamžitě zobrazována zvířata, jejichž jméno obsahuje vyhledávaný řetězec. Informace o obrazovce s filtry lze najít v sekci 4.3.10.4.

Obrazovka obsahuje aktivní prvek `PullToRefresh`, který umožňuje spustit nějakou funkci přetažením seznamu dolů. Tím se zobrazí informace o probíhající aktualizaci, jak je ukázáno na obrázku č. 4.7. Zde konkrétně slouží pro spuštění aktualizace dat ze serveru, což je funkcionalita popsaná v sekci 4.3.6. Během aktualizace jsou tlačítka v navigačním panelu deaktivovaná. Protože aktualizace v případě prvního spuštění použije data přibalená k aplikaci, tak je seznam zvířat vždy viditelný. Obrázky jsou stahovány z internetu a cachovány pomocí knihovny `SDWebImage`. Je tak zajištěno plynulé zobrazování celého seznamu. Pokud obrázek není u zvířete specifikován, zůstává viditelný výchozí obrázek.



Obrázek 4.7: Ukázka aktivního prvku `PullToRefresh`

4.3.10.3 Detail zvířete

Obrazovka zobrazující všechny informace o jednom vybraném zvířeti dle návrhu ze sekce 4.1.3. Jak je z obrázku č. 4.2 patrné, všechny informace se nacházejí na jedné stránce. Celou stránku lze v podstatě rozdělit na tři části – obrázek zvířete, tabulkové informace a obecné textové informace. Není-li obrázek zvířete specifikován, stránka obsahuje pouze dvě části.

Zobrazení je vytvořeno pomocí několika grafických prvků sdružených pod jedním prvkem `StackView`. Tento prvek se často používá pro zobrazení statických dat, která vyžadují použití mnoha grafických prvků.

V horní části obrazovky v navigačním panelu se nachází tlačítko pro zvýraznění zvířete v mapě. Po stisknutí tohoto tlačítka se uživatel přesune na mapu první úrovně, tedy na stejnou instanci zobrazení mapy, na kterou se přesune kliknutím na příslušný *Tab Bar*. Zároveň dojde ke zvýraznění všech mapových objektů, které jsou s tímto zvířetem spojeny. Pokud není specifikováno umístění v Zoo Praha, tak je tlačítko deaktivováno.

4.3.10.4 Seznam filtrů

Obrazovka zobrazující seznam filtrů dle návrhu ze sekce 4.1.1. Tyto filtry lze použít pro filtrování zvířat zobrazených na obrazovce Lexikon zvířat, jejíž implementace je uvedena v sekci 4.3.10.2. Pro zobrazení seznamu je použit standardní grafický prvek *TableViewController*, který používá položky implementované ve třídě `AnimalFilterItemCellVC`.

Hodnoty v seznamu jsou rozděleny do sekcí, každá sekce obsahuje hodnoty vázané na jeden typ filtru. Kliknutím lze filtr označit/odznačit, což je graficky znázorněno odškrtnutím, jak je viditelné na obrázku č.4.1. Jsou-li vybrány dvě hodnoty A a B ze stejné sekce, jedná se o ekvivalent logické formule $A \vee B$. Jsou-li hodnoty A a B vybrány z různých sekcí, jedná se o ekvivalent logické formule $A \wedge B$. Informace o označení/odznačení filtru je uložena v databázi, k uložení dochází těsně před opuštěním obrazovky zpět na obrazovku Lexikon zvířat.

V horní části obrazovky v navigačním panelu se nachází tlačítko pro odznačení všech filtrů.

4.4 Testování

Pro iOS aplikaci jsou vytvořeny automatické testy a aplikace byla podrobena uživatelskému testování, jak je popsáno v následujících sekcích. Pro testování byla použita primárně fyzická zařízení uvedená v tabulce č. 4.2, je tak učiněno z důvodu mnohem horší funkčnosti implementované interaktivní mapy v emulátorech, což je malá nevýhoda oproti aplikacím používajícím standardní knihovnu *MapKit*. Nakonec byly použity i emulátory uvedené v tabulce č. 4.3,

aby byla aplikace otestována i na tabletech a zařízeních s jinou velikostí obrazovek.

Název	Typ zařízení	Verze OS	Rozlišení displeje
iPhone 7	Telefon	14.4	1334 x 750 px
iPhone SE 1st gen.	Telefon	14.4	1136 × 640 px

Tabulka 4.2: Přehled používaných fyzických zařízení.

Název	Typ zařízení	Verze OS	Rozlišení displeje
iPhone 7	Telefon	13.6	1334 x 750 px
iPhone 12	Telefon	14.4	1170 x 2532 px
iPad 8th gen.	Tablet	14.4	2160 x 1620 px

Tabulka 4.3: Přehled používaných emulátorů.

4.4.1 Jednotkové testy

V této sekci jsou popsány automatické unit testy pro iOS aplikaci. V podstatě stejný typ testování byl proveden na serveru, jak bylo popsáno v sekci 3.3. Jde tedy o všeobecný typ testů, jejichž účelem je **otestovat konkrétní funkcionalitu jedné funkce nebo třídy**. Testy je nutné provádět v simulovaném kontextu, kdy např. místo HTTP požadavku jsou použita data z lokálního souboru. Je tak zajištěno, že výsledek testu je závislý pouze na testované funkcionalitě. Tento postup se nazývá *mockování*.

Mockování v této iOS aplikaci je prováděno primárně pomocí *DI* popsaného v sekci 4.3.3. V aplikaci je instance každé závislosti vytvořena v jednom souboru *AppDependency*, jehož ukázka je viditelná v bloku č. 7. Pro unit testy byla vytvořena stejná třída, ve které jsou pomocí testovacích tříd nahrazeny některé v aplikaci používané závislosti. Toto je nejvíce viditelné na inicializaci Realm databáze, kde pro unit testy je vždy vytvořena nová prázdná databáze.

V této iOS aplikaci je pro vytvoření unit testů použit zabudovaný framework `XCTest`. V bloku č. 11 je vidět ukázka jedné třídy s unit testem. Tato třída s testy se nazývá *Test suite* a měla by obsahovat testy pro jednu konkrétní funkcionalitu. Zde konkrétně testuje funkci ze služby `ZooNavigationService` zvanou `computeShortestPath`. Tato služba slouží pro výpočet nejkratší cesty mezi dvěma body v mapě (viz. sekce 4.3.8), tato konkrétní funkce implementuje algoritmus A^*, jehož pseudokód je ukázán v bloku č. 10.

V kódu je ukázána příprava celého *Test suite* a jeden z unit testů. Z třídy s testovacími závislostmi je získána mockovaná Realm databáze, která je zpočátku prázdná. Toto je jediná závislost testované služby. První dvě přetížené funkce zvané `setUp` slouží pro přípravu před spuštěním testů, kde první z nich je spuštěna před všemi unit testy ve třídě a druhá před každým z těchto

unit testů zvlášť. Tímto způsobem je inicializována Realm databáze zkušebními daty, které jsou v testované službě používané. Unit testy jsou funkce, které vždy musí začínat slovem „*test*“. Tento unit test zjišťuje, zda je vrácena správná cesta mezi dvěma uzly v grafu cest. Toho je docíleno porovnáním dvou polí s identifikátory uzlů cest.

V praxi je vhodné vytvářet mnoho unit testů, protože je tak i zaručeno, že změny v kódu nezpůsobily chyby. Avšak nemá smysl unit testy vytvářet pro každou třídu, protože některé třídy jsou příliš provázány se zbytkem aplikace. To platí např. pro *ViewControllery*, které jsou se zbytkem aplikace silně provázány a jejich mockování by bylo mimořádně složité. Toto je další z výhod použité architektury MVVM (viz. sekce 4.2.1), protože *ViewControllery* pak v podstatě neobsahují téměř žádný kód, který by bylo nutné pomocí unit testů testovat.

V této aplikaci má smysl vytvářet testy pro služby a některé části ViewModelů. Po zvážení byla pečlivě otestována primárně již zmíněná služba `ZooNavigationService` obstarávající navigaci.

4.4.2 Testy uživatelského rozhraní

Kromě unit testů lze pomocí frameworku `XCTest` vytvořit tzv. UI testy, které **testují správnost uživatelského rozhraní**. Tyto testy spouští a používají úplně stejně jako uživatel. Test spustí aplikaci a jeho průběh lze pozorovat, je tedy mnohonásobně pomalejší než unit testy. Nemají přístup k vnitřnímu stavu aplikace, ale mohou do aplikace posílat dotazy na stav UI, dotýkat se tlačítek, psát do textových polí apod.

V bloku č. 12 je vidět ukázka jedné třídy s UI testem. Tento test konkrétně kontroluje, zda je aktualizována tabulka s lexikonem zvířat po výběru jednoho z filtrů. Spuštěná aplikace je ovládána skrz třídu `XCUIApplication`. Stejně jako u unit testů je nutné, aby UI testy probíhaly nad simulovaným kontextem. Toho je docíleno předáním argumentu „*–uitesting*“ při spuštění aplikace, protože samotné UI testy nemají přístup k jejímu vnitřnímu stavu. Aplikace si tento argument kontroluje a v případě jeho nastavení je spuštěna s testovacími závislostmi podobně jako v unit testech. Pomocí příkazu `tap()` test klikne na tlačítko a spustí tak jeho funkcionalitu, tedy v tomto případě přechod na obrazovku s filtry.

Těmito testy byly pokryty některé často používané průchody aplikací pracující s lexikonem zvířat. Bohužel část aplikace s interaktivní mapou se nepodařilo otestovat, protože docházelo k pádům způsobeným s největší pravděpodobností přímo knihovnou `Tangram ES`, pomocí které je mapa implementována.

```
class ComputeShortestPathTests: XCTestCase {
    // Mockované hodnoty
    let realm: Realm = testDependencies.realm
    lazy var zooNavigationService: ZooNavigationService =
    ↪ ZooNavigationService(dependencies: testDependencies)

    var roadNodes: Results<RoadNode> =
    ↪ realm.objects(RoadNode.self)
    var roads: Results<Road> = realm.objects(Road.self)

    // Příprava před spuštěním testů

    override class func setUp() {
        testDependencies.testRealmInitializer.updateRealm()
    }

    override func setUp() {
        self.continueAfterFailure = false
    }

    func testShortPath() {
        // Příprava vstupních dat
        let destinationNode = roadNodes.nodeWithId(2816)
        let originNode = roadNodes.nodeWithId(5314)

        // Provedení funkcionality
        let result =
        ↪ zooNavigationService.computeShortestPath(origins:
        ↪ [originNode], destinations: [destinationNode)])

        // Porovnání výsledku
        XCTAssertNotNil(result)

        let resultPathIds: [Int64] = result!
                                .map { $0.node._id }
        let expectedResults: [Int64] = [5314, 9996, 2816]

        XCTAssertEqual(resultPathIds.count,
        ↪ expectedResults.count)
        XCTAssertEqual(resultPathIds, expectedResults)
    }
}
```

Listing 11: Ukázka zjednodušeného iOS unit testu

```
class MastersThesisIOSUITests: XCTestCase {
    private var app: XCUIApplication!

    override func setUp() {
        super.setUp()
        continueAfterFailure = false

        app = XCUIApplication()
        app.launchArguments.append("--uitesting")
    }

    func testFilterSelectedFlow() throws {
        app.launch()

        let lexiconTable = app.tables["LexiconVC_TableView"]
        let filtersTable =
        ↪  app.tables["AnimalFilterVC_TableView"]
        let filtersItem =
        ↪  app.navigationBars.buttons["Filters_NavItem"]

        // Nacházíme se na obrazovce s lexikonem?
        XCTAssertTrue(lexiconTable.exists)
        XCTAssertTrue(filtersItem.exists)

        filtersItem.tap()

        // Nacházíme se na obrazovce s filtry?
        XCTAssertTrue(filtersTable.exists)

        // Kliknout na filtr
        let filterItem =
        ↪  filtersTable.cells.containing(NSPredicate(format:
        ↪  "label == %@", "Paryby")).firstMatch
        XCTAssertTrue(filterItem.exists)

        filterItem.tap()

        // Návrat na obrazovku s lexikonem
        app.navigationBars.buttons.element(boundBy: 0).tap()

        // Byl seznam aktualizován?
        XCTAssertTrue(lexiconTable.exists)
        XCTAssertTrue(filtersItem.exists)
        XCTAssertEqual(lexiconTable.cells.count, 1)
    }
}
```

Listing 12: Ukázka zjednodušeného iOS UI testu

4.4.3 Uživatelské testování

Uživatelské testování je testování prováděné člověkem (tzv. *testerem*), který vyzkouší aplikaci podle předem daného scénáře. Nejde tedy o automatické testovaní. Ideálně je prováděno lidmi, kteří nejsou s rozhraním předem seznámeni, takoví lidé se pak chovají v podstatě jako reální uživatelé. Tímto způsobem je možné zjistit, zda je UI správně navrženo tak, aby bylo srozumitelné a jednoduše použitelné i neinformovaným uživatelem. Jeho účelem je tedy primárně **nalezení chyb v návrhu UI**.

Testování bylo provedeno pěti testery. Dle [61] jde o ideální počet lidí pro tento typ testování kvality uživatelského rozhraní, který odhalí téměř stejné množství problémů jako při testování s více testery. Testování proběhlo na funkční iOS aplikaci, v přítomnosti tvůrce aplikace a na fyzických zařízeních vypsaných v tabulce č. 4.2.

4.4.3.1 Scénáře

Jak již bylo výše řečeno, testeři při uživatelském testování postupují dle předem daného scénáře. Jde o jednoduchou sadu instrukcí, které testerovi říkají, čeho má v aplikaci dosáhnout. Z těchto instrukcí musí být jasný cíl, kterého se tester snaží dosáhnout. Zároveň by instrukce neměli být příliš konkrétní, aby testerovi nenapovídali, jak tohoto cíle má dosáhnout.

Scénáře by měly být navrženy tak, aby se průchod aplikací co nejvíce podobal jejímu reálnému použití a zároveň bylo vyzkoušeno co nejvíce funkcionalit. Byly tak vytvořeny tyto scénáře:

1. Najdi na mapě Pavilon šelem a plazů.
2. Najdi všechny savce, kteří žijí v mořích nebo sladkých vodách.
3. Znovu zobraz všechna zvířata v lexikonu.
4. Pokus se aktualizovat seznam zvířat.
5. Zjisti, jakou potravu přijímá Dikobraz jihoafrický.
6. Zvýrazni všechny pavilony a/nebo venkovní výběhy, ve kterých se Dikobraz jihoafrický nachází.
7. Jedním ze zvířat ve zvýrazněném pavilonu z předchozího bodu je Bodlín Telfairův. Na jakém kontinentu je možné ho najít?
8. Přejdi zpět na mapu a zruš všechna zvýraznění.
9. Najdi vlastní pozici na mapě. Není-li to možné, zkus vysvětlit proč.
10. Spusť navigaci k Pavilonu šelem a plazů.
11. Přeruš navigaci.

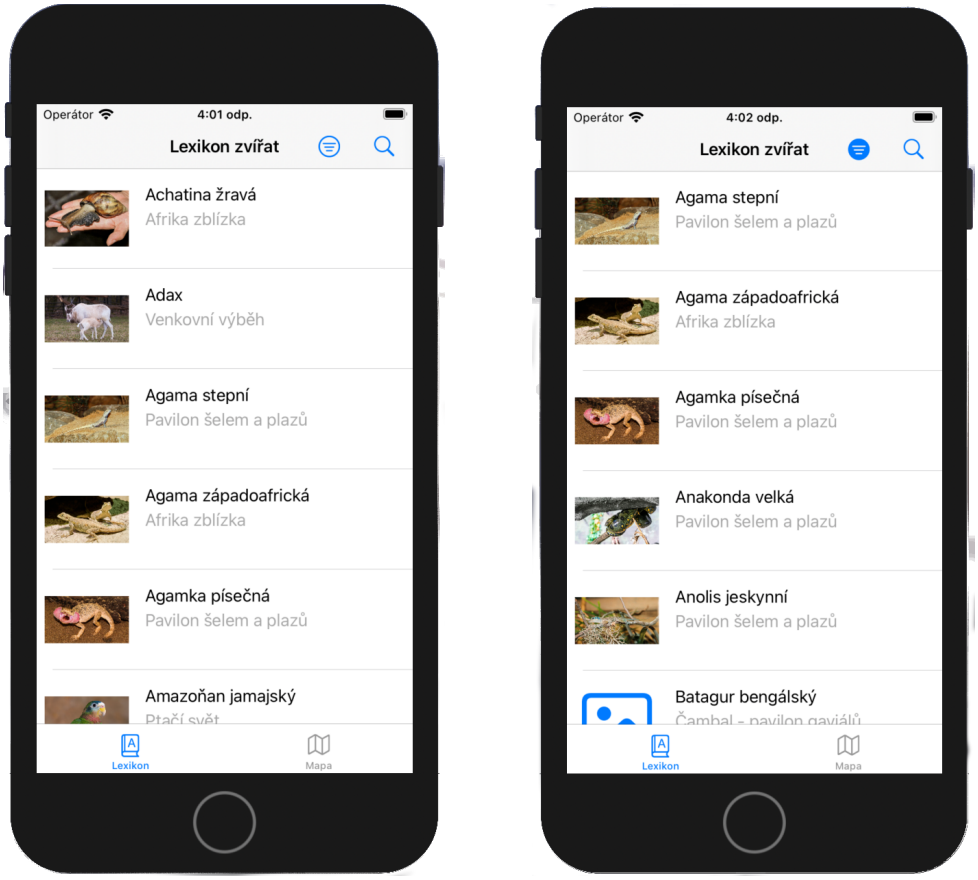
4.4.3.2 Výsledek testu

Po otestování aplikace všemi testery byly nalezeny tyto problémy:

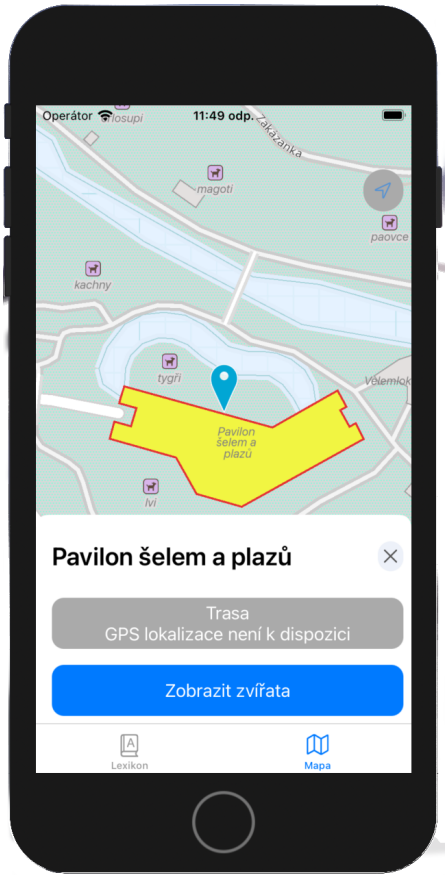
1. Na obrazovce s lexikonem zvířat není indikováno, zda je aktivní nějaký filtr. Na tento problém poukázali všichni testeři.
2. Jeden z testerů narazil na problém, kdy při označení jednoho z filtrů došlo k označení dalšího filtru.
3. Všichni testeři v případě 9. scénáře správně odpověděli, že není možné najít vlastní pozici v mapě, protože se nachází mimo areál Zoo Praha. Avšak tři z testerů řekli, že by bylo vhodné tento stav indikovat i jinak než pouze deaktivací prvků.
4. Jeden tester se pokusil přejít mimo mapu a všiml si grafického „zadrhávání“, které tento pokus způsobí.
5. U jednoho s testerů v jednu chvíli mapa přestala reagovat a objevilo se grafické „zadrhávání“. Toto bylo způsobeno větší obrazovkou použitého fyzického zařízení.
6. Třem testerům chyběla možnost filtrovat pomocí pavilonů.

Dle těchto nalezených problémů byly vytvořeny následující úpravy:

1. Byla změněna ikona pro přechod na seznam filtrů zvířat. Nyní se mění podle toho, zda je nějaký filtr nastaven, jak je vidět na obrázku č. 4.8.
2. Šlo o chybu v aplikaci, která by měla být opravena.
3. Nyní je přímo v tlačítku pro spuštění napsáno, proč je deaktivováno, jak je vidět na obrázku č. 4.9
4. Omezení interaktivní mapy na okolí Zoo Praha je bohužel problém, který se zatím nepodařilo zcela vyřešit. Použitá knihovna toto bohužel neumožňuje. Bylo zvětšeno mapou zobrazované okolí, aby uživatel tento problém nemusel řešit.
5. Bylo zvětšeno mapou zobrazované okolí.
6. Přidáno filtrování pomocí pavilonů.



Obrázek 4.8: Žádný filtr není vybrán (vlevo) vs. nějaký filtr je vybrán (vpravo).



Obrázek 4.9: Informace, proč je tlačítko deaktivováno.

Závěr

Cílem práce bylo vytvořit doprovodnou aplikaci pro Zoo Praha, která poskytuje všechny informace o zde žijících zvířatech a plně interaktivní mapu, kterou lze použít pro orientaci v areálu Zoo Praha. Je tedy určena primárně pro návštěvníky nebo lidi, kteří pražskou zoo navštívit plánují. Data pro tuto aplikaci zpracovává server v jazyce Python, který je též implementován.

Nejdříve byla analyzována dostupná data ze Zoo Praha. Původně měla být tato data čerpána z portálu OpenData Praha, kde byla pražskou zoo zveřejněna ve formátech vhodných pro strojové zpracování. Bohužel tato data nebyla během práce dostupná, a proto byla získávána přímo ze stránek Zoo Praha pomocí techniky *Web scraping*. Data tak bylo možné získat a navíc byla zajištěna i větší aktuálnost oproti původním datům z portálu OpenData Praha.

Dále byly analyzovány existující mapové podklady použité pro implementaci interaktivní mapy. Zde byly vybrány volně dostupné mapové podklady z *OpenStreetMap*, na kterých jsou založené například *Mapy.cz*. Tyto jsou mnohem přesnější než mapy od firmy Apple, navíc obsahují i pozice jednotlivých zvířat.

Na základě dat ze Zoo Praha, mapových podkladů a rešerše existujících aplikací byly specifikovány funkcionality, které mobilní aplikace a server poskytují.

Následně byl popsán vývoj pomocného serveru v jazyce Python, jehož hlavním účelem je automatické stahování, úprava, ukládání a poskytování dat, která mobilní aplikace používá. Nejprve byly vybrány databáze pro uložení dat. Následně byly popsány implementace funkcionalit pro stahování jednotlivých typů dat, způsob automatizace celého serveru a specifikace architektury REST rozhraní, pomocí kterého server data poskytuje. Dále byly popsány vytvořené automatické testy. Poslední popsanou částí bylo nasazení serveru a databází na cloudové platformy. Pro hlavní používanou MongoDB databázi byla použita platforma Atlas, pro ukládání souborů byla použita služba AWS S3 a samotný server je nasazen na platformě Heroku.

Nakonec byl popsán vývoj samotné mobilní aplikace pro zařízení se systé-

mem iOS. Nejprve byl proveden návrh uživatelského rozhraní, jehož výstupem bylo grafické zpracování jednotlivých obrazovek a přechodů mezi nimi. Tento návrh se řídil primárně jednotlivými specifikovanými požadavky a obecnými postupy vytváření uživatelského rozhraní v iOS aplikacích. Následně je popsána použitá softwarová architektura. Dále byla blíže popsána samotná implementace aplikace od postupů a technologií důležitých pro celou aplikaci až po konkrétní implementaci uživatelského rozhraní.

Nakonec bylo provedeno automatické a uživatelské testování mobilní aplikace. Chyby nalezené během uživatelského testování byly popsány a v aplikaci opraveny.

Zoo Praha nabízí mnoho dalších informací, o které by bylo možné aplikaci vylepšit. Jde minimálně o informace o nadcházejících akcích, které by bylo možné též v mapě zobrazovat.

Bibliografie

1. BOBEK, Miroslav. *Rákosův pavilon, skvělé odchovy i rekordní návštěvnost, takový byl rok 2019 v Zoo Praha* [online]. 2019-12-31 [cit. 2021-04-27]. Dostupné z: https://www.zoopraha.cz/aktualne/ostatni-clanky/12113-rakosuv-pavilon-skvele-odchovy-i-rekordni-navstevnost-takovy-byl-rok-2019-v-zoo-praha.
2. *O projektu OpenData Praha* [online] [cit. 2020-09-12]. Dostupné z: https://opendata.praha.eu/about.
3. *Aktivita u datasetu Lexikon zvířat* [online]. 2020-09-08 [cit. 2020-09-12]. Dostupné z: https://opendata.praha.eu/dataset/activity/zoo-lexikon-zvirat.
4. *Odkaz na interní API Zoo Praha* [online]. 2017-02-10 [cit. 2020-09-12]. Dostupné z: https://opendata.praha.eu/dataset/zoo-lexikon-zvirat/resource/4f2ea9e7-e0ab-4eb9-a775-6d3ef152f401.
5. *Lexikon zvířat ze Zoo Praha dostupný online* [online] [cit. 2020-09-16]. Dostupné z: https://www.zoopraha.cz/zvirata-a-expozice/lexikon-zvirat.
6. *Seznam zvířat ze Zoo Praha, u kterých je možná adopce* [online] [cit. 2020-09-16]. Dostupné z: https://www.zoopraha.cz/jak-pomoci/adopce/seznam-zvirat-pro-adopci.
7. *O projektu OpenStreetMap* [online] [cit. 2020-09-12]. Dostupné z: https://www.openstreetmap.org/about.
8. *Historie projektu OpenStreetMap* [online] [cit. 2020-09-12]. Dostupné z: https://welcome.openstreetmap.org/about-osm-community/history-of-osm/.
9. *Služby a aplikace založené na projektu OpenStreetMap* [online] [cit. 2020-09-12]. Dostupné z: https://wiki.openstreetmap.org/wiki/List_of_OSM-based_services#Services.

10. *Služby poskytující mapové čtverce* [online] [cit. 2020-09-12]. Dostupné z: https://wiki.openstreetmap.org/wiki/Tiles#Base_maps.

11. *Formát MBTiles* [online] [cit. 2020-09-12]. Dostupné z: https://wiki.openstreetmap.org/wiki/MBTiles.

12. *The Open Definition – základní informace* [online] [cit. 2020-09-12]. Dostupné z: https://opendefinition.org/.

13. *The Open Definition – plné znění* [online] [cit. 2020-09-12]. Dostupné z: https://opendefinition.org/od/2.1/en/.

14. MITCHELL, Ryan. Web Scraping with Python. In: [online]. Second. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2018, s. ix–xi [cit. 2020-09-23]. ISBN 9781491985571. Dostupné z: https://books.google.cz/books?id=TYtSDwAAQBAJ&printsec=frontcover#v=onepage&q&f=false.

15. DOJČINOVSKI, Milan; KUCHAŘ, Jaroslav. *Web Data Mining: Data Access and Acquisition Methods 2* [online]. ČVUT, 2019 [cit. 2020-09-23].

16. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, 2000. Dissertation. University of California, Irvine.

17. *Mobilní aplikace Zoo Liberec pro iOS* [online]. 2014-09-26 [cit. 2020-09-12]. Dostupné z: https://apps.apple.com/us/app/zoo-liberec/id912298394.

18. *Mobilní aplikace Údolí slonů pro Android* [online]. 2017-03-05 [cit. 2020-09-12]. Dostupné z: https://play.google.com/store/apps/details?id=aitom.zoo&hl=cs.

19. *Oficiální stránka MongoDB* [online] [cit. 2021-02-21]. Dostupné z: https://www.mongodb.com.

20. *NoSQL explained* [online] [cit. 2021-02-21]. Dostupné z: https://www.mongodb.com/nosql-explained.

21. *AWS S3 overview* [online] [cit. 2021-02-21]. Dostupné z: https://aws.amazon.com/s3/?nc=sn&loc=1.

22. *Mapzen vector tiles* [online] [cit. 2020-09-12]. Dostupné z: https://www.mapzen.com/projects/vector-tiles/.

23. *Tilepacks Github* [online] [cit. 2021-02-24]. Dostupné z: https://github.com/tilezen/tilepacks.

24. *BeautifulSoup4* [online] [cit. 2021-02-24]. Dostupné z: https://pypi.org/project/beautifulsoup4/.

25. *Croniter GitHub* [online] [cit. 2021-02-24]. Dostupné z: https://github.com/kiorky/croniter.

26. *Nástroj pro pomoc s nastavením „cron“ výrazu CronGuru* [online] [cit. 2021-02-24]. Dostupné z: https://crontab.guru/.
27. *Heroku3 GitHub* [online] [cit. 2021-02-24]. Dostupné z: https://github.com/martyzz1/heroku3.py.
28. *FastAPI Documentation* [online] [cit. 2021-02-25]. Dostupné z: https://fastapi.tiangolo.com/.
29. *Which python web frameworks do you use?* [Online] [cit. 2021-03-08]. Dostupné z: https://www.jetbrains.com/lp/devecosystem-2020/python/.
30. *Dokumentace ASGI standardu* [online] [cit. 2021-03-08]. Dostupné z: https://asgi.readthedocs.io/en/latest/introduction.html.
31. *Pytest GitHub* [online] [cit. 2021-03-01]. Dostupné z: https://github.com/pytest-dev/pytest.
32. *Betamax GitHub* [online] [cit. 2021-03-01]. Dostupné z: https://github.com/betamaxteam/betamax.
33. *UnitTest patchers* [online] [cit. 2021-03-01]. Dostupné z: https://docs.python.org/3/library/unittest.mock.html#the-patchers.
34. *Dokumentace nástroje SOAPUI* [online] [cit. 2021-03-04]. Dostupné z: https://www.soapui.org/getting-started/rest-testing/.
35. *Dokumentace MongoDB Atlas* [online] [cit. 2021-02-28]. Dostupné z: https://docs.atlas.mongodb.com/.
36. *MongoDB Atlas cluster* [online] [cit. 2021-02-28]. Dostupné z: https://docs.atlas.mongodb.com/cluster-tier/.
37. *Základní informace o Heroku* [online] [cit. 2021-02-25]. Dostupné z: https://www.heroku.com/about.
38. *Heroku dyno* [online] [cit. 2021-02-28]. Dostupné z: https://devcenter.heroku.com/articles/dynos.
39. *Heroku scheduler addon* [online] [cit. 2021-02-28]. Dostupné z: https://devcenter.heroku.com/articles/scheduler.
40. *Dokumentace Travis CI* [online] [cit. 2021-02-28]. Dostupné z: https://docs.travis-ci.com/user/for-beginners/.
41. *Tab Bars* [online] [cit. 2021-04-20]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/ios/bars/tab-bars.
42. *Search Bars* [online] [cit. 2021-04-20]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/ios/bars/search-bars/.
43. *iOS MVVM template* [online] [cit. 2021-04-21]. Dostupné z: https://github.com/AckeeCZ/iOS-MVVM-ProjectTemplate.

44. *About Cocoapods* [online] [cit. 2021-04-21]. Dostupné z: https://cocoapods.org/about.

45. *Carthage* [online] [cit. 2021-04-21]. Dostupné z: https://github.com/Carthage/Carthage.

46. *Knihovna ReactiveSwift* [online] [cit. 2021-04-21]. Dostupné z: https://github.com/ReactiveCocoa/ReactiveSwift.

47. ESCOFFIER, Clement. *5 things to know about functional reactive programming* [online]. 2017-06-30 [cit. 2021-04-21]. Dostupné z: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/.

48. *Combine* [online] [cit. 2021-04-21]. Dostupné z: https://developer.apple.com/documentation/combine.

49. *RxSwift* [online] [cit. 2021-04-21]. Dostupné z: https://github.com/ReactiveX/RxSwift.

50. ZABŁOCKI, Krzysztof. *Using protocol composition for dependency injection* [online] [cit. 2021-04-23]. Dostupné z: http://merowing.info/2017/04/using-protocol-compositon-for-dependency-injection/.

51. *ACKLocalization* [online] [cit. 2021-04-25]. Dostupné z: https://github.com/AckeeCZ/ACKLocalization.

52. *SwiftGen* [online] [cit. 2021-04-25]. Dostupné z: https://github.com/SwiftGen/SwiftGen.

53. *Knihovna Alamofire* [online] [cit. 2021-04-21]. Dostupné z: https://github.com/Alamofire/Alamofire.

54. PATEL, Amit. *Introduction to A** [online] [cit. 2021-04-21]. Dostupné z: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

55. *Using Interface Builder* [online] [cit. 2021-04-23]. Dostupné z: https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder.html.

56. *SwiftUI* [online] [cit. 2021-04-23]. Dostupné z: https://developer.apple.com/xcode/swiftui/.

57. *ReactiveCocoa* [online] [cit. 2021-04-23]. Dostupné z: https://github.com/ReactiveCocoa/ReactiveCocoa.

58. *Tangram ES* [online] [cit. 2021-04-23]. Dostupné z: https://github.com/tangrams/tangram-es.

59. *Scene file* [online] [cit. 2021-04-25]. Dostupné z: https://tangrams.readthedocs.io/en/master/Overviews/Scene-File/.

60. *BubbleWrap style* [online] [cit. 2021-04-25]. Dostupné z: https://github.com/tangrams/bubble-wrap.

61. NIELSEN, Jakob. *How Many Test Users in a Usability Study?* [Online]. 2012-06-03 [cit. 2021-04-25]. Dostupné z: `https://www.nngroup.com/articles/how-many-test-users/`.

Příloha **A**

Seznam použitých zkratek

UI User interface

XML Extensible markup language

OSM Projekt OpenStreetMap

API Application Programmable Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

REST Representational State Transfer

OSM Projekt OpenStreetMap

AWS Amazon Web Services

CI Continuous Integration

CD Continuous Deployment

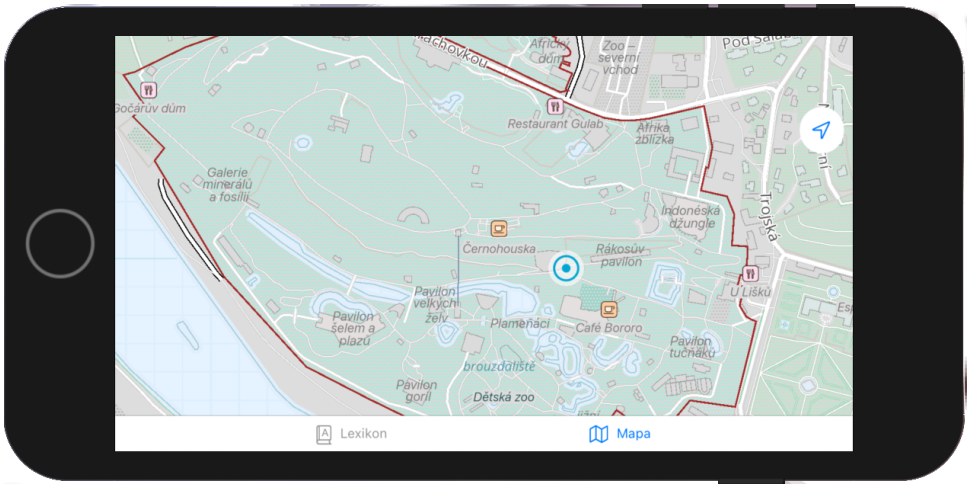
Příloha B

Obsah přiloženého CD

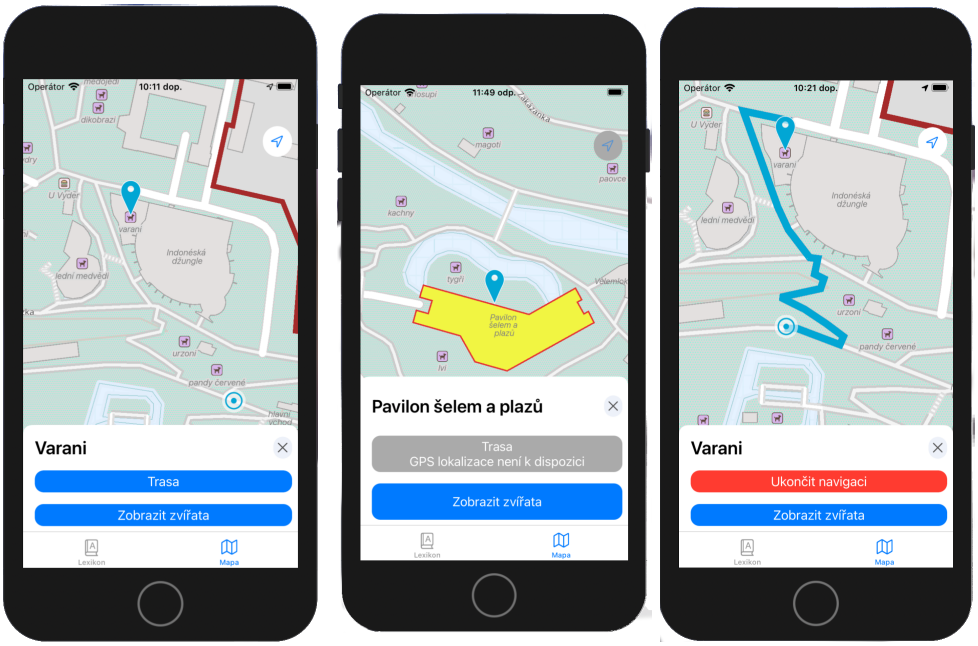
- `appShowcase.mov` ..... ukázkové video iOS aplikace
- `readme.txt` ..... stručný popis obsahu CD
- `restApiDocs.json` ..... REST rozhraní serveru ve formátu OpenAPI
- `src`
  - `app` ..... zdrojové kódy mobilní aplikace
  - `server` ..... zdrojové kódy serveru
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF

Příloha C

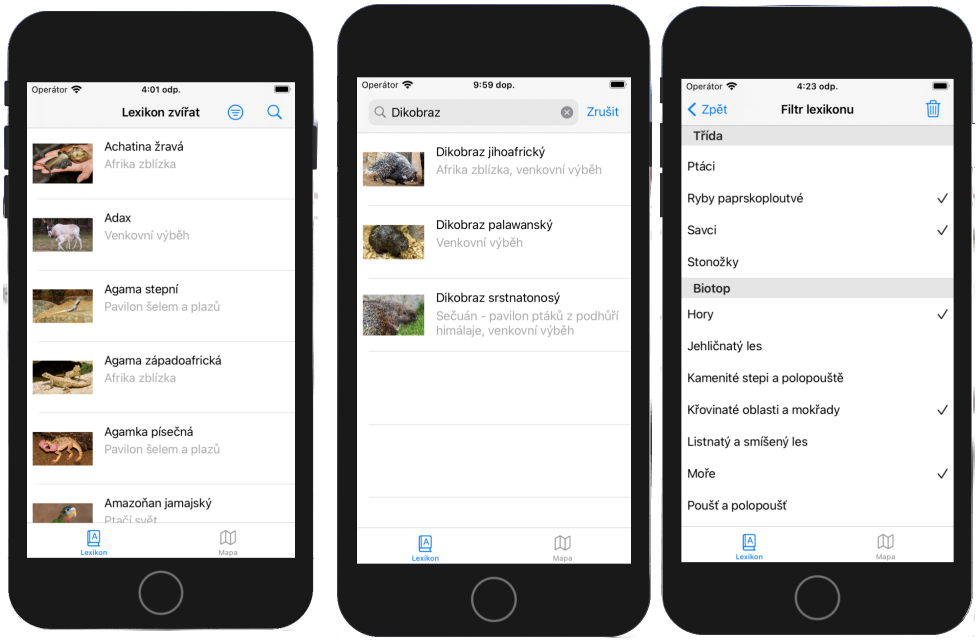
Snímky výsledné aplikace



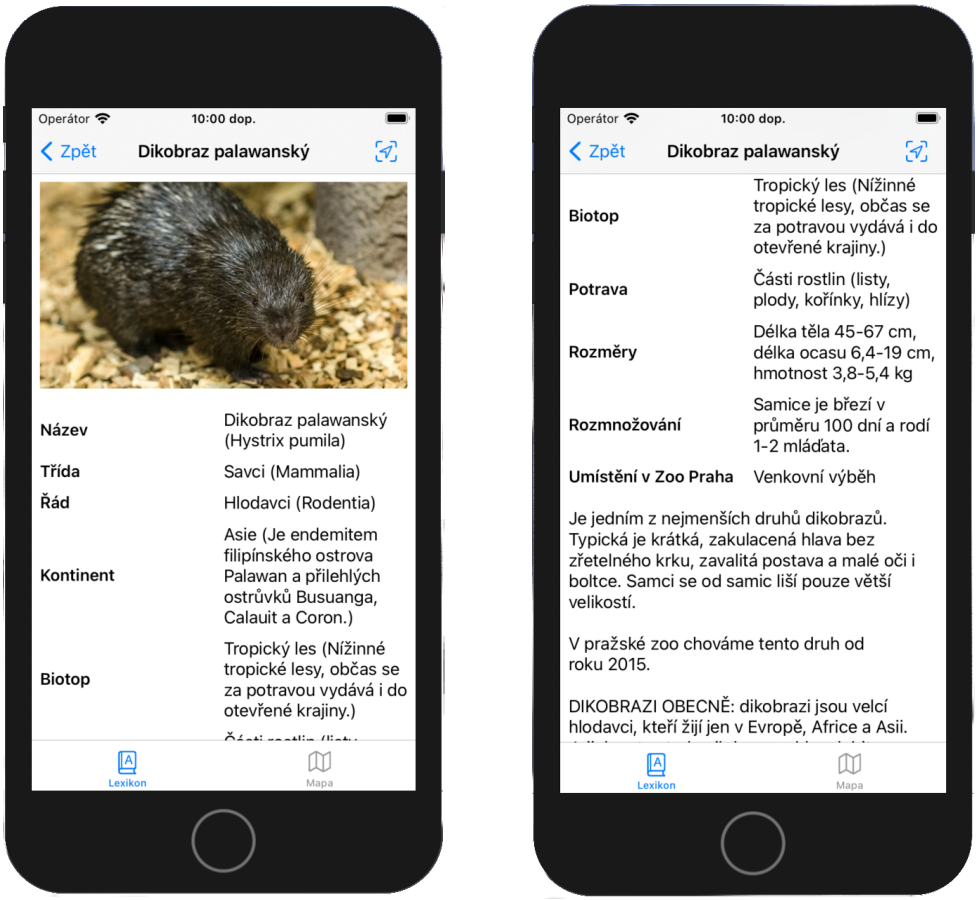
Obrázek C.1: Zobrazení mapy Zoo Praha



Obrázek C.2: Zvýrazněný výběhu/pavilonu a ukázka navigace



Obrázek C.3: Lexikon zvířat (vlevo), textové vyhledávání a seznam filtrů (vpravo)



Obrázek C.4: Detail zvířete