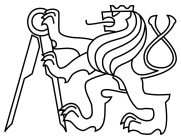
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Mobilní lexikon zvířat ZOO Praha pro Android |
| **Student:** | Bc. Martin Zavadil |
| **Vedoucí:** | Ing. Josef Gattermayer |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce zimního semestru 2017/18 |

## Pokyny pro vypracování

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

1. Navrhněte vhodnou funkcionalitu pro mobilní aplikaci na základě dostupných dat. Vytvořte wireframy a několikrát je s vedoucím iterujte.

2. Konzultujte s vedoucím práce grafické pojetí aplikace (grafiku dodá vedoucí).

3. Navrhněte a implementujte server v Node.js, který bude data stahovat z portálu opendata.praha.eu a přes REST API nabízet mobilní aplikaci. Případně bude poskytovat další nutnou funkcionalitu pro běh iOS aplikace, která je realizována v samostatné bakalářské práci.

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

## Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 27. září 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

# Mobilní lexikon zvířat ZOO Praha pro Android

***Bc. Martin Zavadil***

Vedoucí práce: Ing. Josef Gattermayer

9. ledna 2018

# Poděkování

Rád bych poděkoval svému vedoucímu Ing. Josefu Gattermayerovi za trpělivost a rady, a svým blízkým za trpělivost a podporu.

# Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, 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 způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). 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 způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2018 .....

### Odkaz na tuto práci

Zavadil, Martin. *Mobilní lexikon zvířat ZOO Praha pro Android.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

# Abstrakt

Tato diplomová práce se zabývá implementací mobilní aplikace Lexikon zvířat pro operační systém Android. Data aplikace pochází z otevřených dat Zoo Praha. Součástí práce je i server poskytující data, implementovaný v Node.js. Práce popisuje zdrojová data, použité technologie, samotnou implementaci aplikace a serveru, dokumentaci a testování. Mobilní aplikace je implementována v technologii Xamarin.Android.

**Klíčová slova** lexikon, zoo, xamarin, android, node.js, mobil, aplikace

# Abstract

This diploma thesis deals with implementation of mobile application Lexikon of Animals for Android operating system. Application data comes from the open data of the Prague Zoo. Thesis includes alse the server providing data, implemented in Node.js. The thesis describes source data, used technologies, application and server implementation itself, documentation and testing. The mobile application is implemented in Xamarin.Android technology.

**Keywords** lexicon, zoo, xamarin, android, node.js, mobile, application

# Obsah

# Seznam obrázků

# Úvod

## Motivace

V současné době je možný přístup k mnoha různým a často zajímavým informacím. Zároveň má v současnosti téměř každý tzv. chytrý mobilní telefon, který umožňuje rychlý přístup k internetu a tedy k mnoha informacím. Pokud tyto informace nejsou nijak zpracované, mohou být nesrozumitelné a tedy nepoužitelné.

Příkladem takovýchto informací jsou data vydané Zoo Praha na portálu *opendata.praha.eu/organization/zoo*[1]. Tato data se přímo nabízejí ke zpracování právě v mobilní aplikaci, čímž se zpřístupní veřejnosti.

## Cíl

Cílem této práce je analyzovat data z webového portálu *opendata.praha.eu/organization/zoo*[1]. Pro tyto data navrhnout a implementovat server v javascriptovém serverovém frameworku Node.js, který bude tyto data stahovat, zpracovávat a poskytovat jako REST API (nejen) mobilní aplikaci.

Další částí je návrh uživatelského rozhraní v podobě wireframů, které budou následně sloužit jako podklad pro samotnou mobilní aplikaci.

Nakonec navrhnout a implementovat mobilní aplikaci pro operační systém Android, která bude data stahovat a zobrazovat koncovým uživatelům.

Všechny implementované části je nutné řádně otestovat a zdokumentovat.

## Struktura práce

Tato práce je rozdělena do několika kapitol, které popisují jednotlivé části tvorby serverové a mobilní části této práce.

První kapitola přibližuje problematiku otevřených dat a analyzuje zdrojová data poskytovaná Zoo Praha.

Druhá kapitola popisuje návrh funkcionality, vytváření návrhu uživatelského rozhraní (wireframů) a samotný návrh serverové a mobilní části této práce.

Třetí kapitola popisuje samotnou implementaci serverové i mobilní části.

Čtvrtá kapitola popisuje způsob dokumentace a tvorbu testů pro obě části této práce.

Pátá kapitola popisuje způsob nasazení serverové části této práce do cloudové služby pro zprovoznění implementovaného REST API. Dále popisuje nasazení mobilní aplikace do obchodu Google Play a zprovoznění prostředí pro průběžnou integraci.

Šestá kapitola představuje použité technologie a přibližuje srovnání mezi některými technologiemi pro vývoj mobilních aplikací.

Poslední, sedmá kapitola popisuje některé řešené problémy a výhledy do budoucna.

V přílohách lze nalézt seznam použitých zkratek, strukturu obsahu přiloženého CD, obrázky wireframů a UML diagramy tříd rozdělené podle implementovaných aktivit.

## Současná řešení

Lexikon zvířat lze najít na stránkách Zoo Praha na *www.zoopraha.cz/zvirata-a-expozice/lexikon-zvirat*. V mobilní verzi stránek chybí nabídka pro listování mezi zvířaty. Stránku lze přepnout na klasické zobrazení, které ovšem není pro menší obrazovky mobilních telefonů přívětivé. Nativní aplikace navíc umožní rychlejší odezvu než webové stránky, možnost offline procházení dat i více funkcí.

Zoo Praha měla svoji mobilní aplikaci ve zkušebním provozu. Ta byla v průběhu psaní této práce z obchodu Google Play odebrána nebo skryta. Lexikon zvířat však neobsahovala.

Kapitola 1

# Analýza

Tato kapitola se zabývá *opendaty* a analýzou dostupných dat.

## 1.1 Opendata

*Opendata*, neboli *otevřená data* jsou [2]:

- komukoliv zdarma a volně dostupná ke stažení na Internetu,
- zveřejněna v úplné podobě bez záměrného odstraňování údajů,
- strojově čitelná,
- použitelná bez omezení (tj. pro komerční i nekomerční účely).

Tato data také musí být [2]:

- katalogizovaná v Národním katalogu otevřených dat,
- zveřejněná dle Standardů publikace a katalogizace otevřených dat Ministerstva vnitra ČR dokumentovaných na tomto webu.

## 1.2 Pražská opendata

Pražská opendata se nachází na webovém portálu *http://opendata.praha.eu*[3]. Zde zveřejňuje data Magistrát hl. m. Prahy, různé příspěvkové organizace, městské části a další subjekty[2]. Mimo jiné se zde nachází data od Zoo Praha, kterými se tato práce dále zabývá.

## 1.3 CKAN

„*CKAN* je přední světová open source platforma pro datové portály.

CKAN je kompletní softwarové řešení, které je ihned připravené k nasazení, pomáhající k tomu, aby data byla dostupná a využitelná. To je dosaženo pomocí nástrojů, které usnadňují publikaci, sdílení, vyhledávání a využívání dat (včetně ukládání dat a zpřístupnění dat pomocí robustního API). CKAN je určen poskytovatelům dat (orgány státní správy a samosprávy, podniky a organizace), kteří chtějí, aby jejich data byla dobře dostupná.

CKAN je využíván vládami a skupinami uživatelů po celém světě a je to platforma mnoha oficiálních a komunitních datových portálů, mezi které patří portály veřejné správy na regionální, národní a nadnárodní úrovni, jako jsou např. portály Velké Británie data.gov.uk a Evropské Unie publicdata.eu, Brazílie dados.gov.br, portály veřejné správy v Holandsku, jakož i portály měst a obcí v USA, Velké Británii, Argentině, Finsku a jinde.“[2]

Více o CKAN lze nalézt na *https://ckan.org* [4].

## 1.4 Datové API

Opendata obsahují vlastní datové API řešené pomocí CKAN. V této práci však není použito, protože všechny použité datové sady odkazují na web Zoo Praha. Na ty se nedá použít CKAN API a je potřeba je stáhnout a zpracovat celé.

Datové API je vždy popsané u konkrétní datové sady ve zdrojových opendatech (viz [3]). Protože se v této práci nevyužívá, nebude zde popsáno.

## 1.5 Analýza zdrojových dat

Tato práce se zabývá daty poskytnutými od Zoo Praha (viz [1]). Ta má na pražských opendatech 4 datové sady, které budou v této sekci dále popsány.

### 1.5.1 Lexikon zvířat

Datová sada *Lexikon zvířat* obsahuje několik tabulek. Zásadní jsou tabulky `Lexikon zvířat`, které jsou zde ve třech formátech, a to *CSV* (Comma-separated values, hodnoty oddělené čárkami), *XLS* (formát souboru Microsoft Excel) a *JSON* (javascript object notation, více viz 6.2).

Ostatní tabulky v této datové sadě tvoří jakousi databázovou strukturu, která však není aktuální a hlavní tabulka `Lexikon zvířat` s nimi není provázána. Navíc všechny údaje z těchto tabulek jsou již obsaženy i v tabulkách `Lexikon zvířat`. Tyto tabulky jsou tedy nepoužitelné a dále s nimi nebude počítáno.

Všechny tabulky `Lexikon zvířat` jsou zdánlivě stejné, při bližším prozkoumání však obsahují zásadní rozdíly. Tabulka ve formátu CSV obsahuje

327 zvířat, tabulka ve formátu XLS obsahuje 582 zvířat a tabulka ve formátu JSON obsahuje 605 zvířat (aktuální k 10. 12. 2017). Tabulka ve formátu XLS má některá data u několika zvířat zapsané do špatných sloupců. Tabulka ve formátu JSON má složitější strukturu (vnořené objekty, viz 6.2), některé texty obsahují HTML tagy a URL adresa obrázku zvířete je obsažena ve vlastnosti `desctiption` v HTML tagu.

Podle pozorování se data v datových sadách umístěných na opendatech aktualizují přibližně jednou za půl roku. Zdroj tabulky v JSON formátu však odkazuje na web Zoo Praha a tak se zdá být nejspolehlivější a nejaktuálnější. Také obsahuje nejvíce zvířat a data jsou v pořádku. Nevýhoda je složitější zpracování dat oproti ostatním formátům a nemožnost použít pro dotazování CKAN API.

Protože datová tabulka `Lexikon zvířat` ve formátu JSON obsahuje nejlepší (a pravděpodobně nejspolehlivější) zdroj dat, bude použita pro tuto práci právě tato tabulka.

Jeden záznam v datové sadě ve formátu JSON má tuto strukturu:

```
{
"id":
"title":
"description":
"latin_title":
"class": { "title":  }
"breeding":
"reproduction":
"food_note":
"biotopes_note":
"spread_note":
"projects_note":
"proportions":
"attractions":
"biotop": { "name_b": }
"food": { "name_f": }
"continents": { "name_c": },
"classes": { "title": }
"localities": { "title": , "url": }
}
```

### 1.5.2 Akce v zoo

Tato datová sada obsahuje tři verze tabulky `Akce v zoo`, ve formátech CSV, JSON a DATA. Tabulka ve formátu CSV obsahuje neaktualizovaný kalendář akcí. Tabulka ve formátu JSON obsahuje také kalendář akcí, ale odkazuje

přímo na web Zoo Praha a obsahuje aktualizovaná data. Zároveň však neobsahuje uplynulé akce, pouze plánované.

Poslední tabulka odkazuje na *RSS* (viz [5]) kanál Zoo Praha a data v ní odkazují na články a novinky na webu Zoo Praha.

Jeden záznam v datové sadě Akcí ve formátu JSON má tuto strukturu:

```
{
"start":
"end":
"summary":
"description":
}
```

### 1.5.3 Návštěvnost

Tato datová sada ukazuje návštěvnost Zoologické zahrady za posledních několik let. Každý záznam obsahuje rok, celkový počet návštěvníků v tomto roce a datum návštěvy miliontého návštěvníka v daném roce. Pro tuto práci není tato datová sada dále zajímavá a již se jí nezabývá.

### 1.5.4 Adopce zvířat

Tato datová sada obsahuje informace o možnosti adopce různých zvířat v Zoo Praha. Data obsahují název zvířete, anglický název, zařazení do taxonomické třídy, cenu za adopci a informaci, zda je možné si zvíře prohlédnout.

Kapitola 2

# Návrh

Tato kapitola se zabývá návrhem funkcionality, tvorbou návrhu uživatelského rozhraní (wireframů) a návrhem serveru a mobilní aplikace na základě dat poskytovaných Zoo Praha.

## 2.1 Návrh funkcionality

V této sekci je navržena možná funkcionalita aplikace na základě dostupných dat. Výsledná funkcionalita je výstupem sekce 2.2.

Funkcionalita aplikace se odvíjí z dostupných dat. Základem aplikace bude Lexikon zvířat, tedy možnost procházení, vyhledávání a zobrazování informací o zvířatech nacházejících se v pražské zoo a o kterých jsou dostupná data.

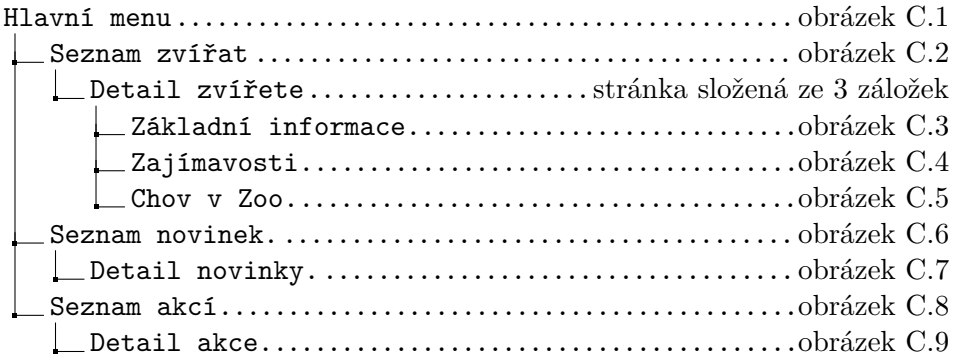
Protože se jedná o zvířata v zoologické zahradě, a vzhledem k tomu, že tuto informaci zdrojová data obsahují, bylo by vhodné umožnit zobrazení zvířat nacházejících se v určeném pavilonu či výběhu. Také by bylo možné vytvořit interaktivní mapu Zoo s vyznačenými výběhy a pavilony. Dostupná data by případně bylo možné zpracovat pro vytvoření nějaké formy hry či kvízu.

Dále je možné využít ostatní datové sady, tedy možnost zobrazení novinek v Zoo z RSS kanálu, možnost zobrazení akcí pořádaných v Zoo nebo zobrazení zvířat pro adopci.

## 2.2 Wireframy

V souladu se zadáním byly vytvořeny *wireframy* (česky tzv. *drátěný model*, dále jen **model**). Na pokyn vedoucího práce bylo vybráno 5 respondentů, kteří tento model testovali a připomínkovali.

Model byl vytvořen v programu *Pencil* (viz [6]). Ten umožňuje vytvoření wireframů a jejich exportování např. do interaktivní webové HTML stránky, souboru ve formátu PDF nebo rastrových obrázků.




Obrázek 2.1: Wireframe: Hierarchie stránek - první iterace

Tvorba a testování modelu probíhala ve třech iteracích popsaných dále. Exportované obrázky jsou obsaženy v příloze C. Každý obrázek obrazovky má v popisu číslo verze (iterace). Verze 2 a 3 neobsahují obrázky pro obrazovky nezměněné oproti předchozím verzím. Kompletní model, včetně zdrojových souborů i exportů do HTML a PDF, lze najít na přiloženém CD.

### 2.2.1 První iterace

První verze modelu obsahuje několik základních obrazovek. Ty vycházejí ze sekce 2.1. Hierarchie obrazovek je zobrazena na obrázku 2.1.

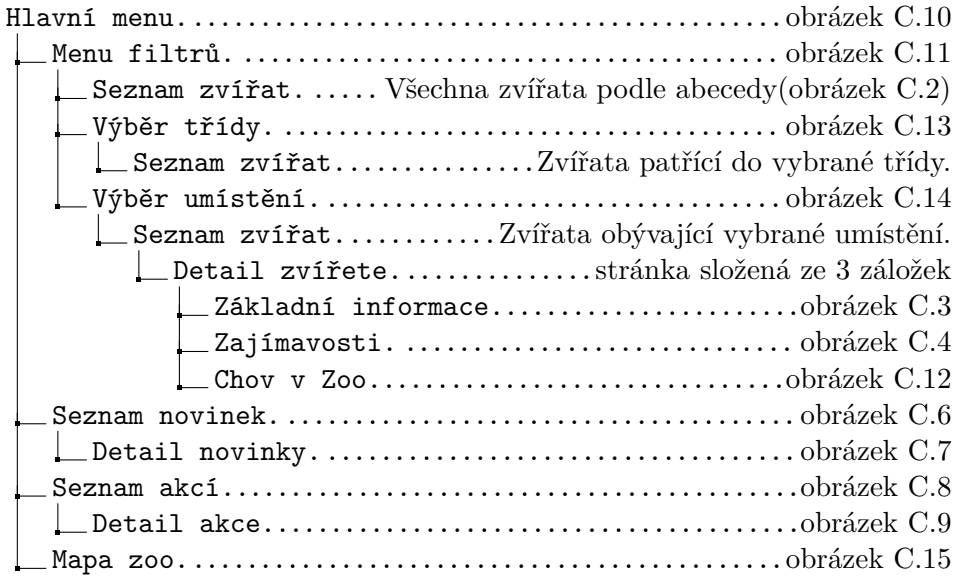
Pro testování modelu byl model exportován do interaktivní HTML stránky. Ta umožňuje simulaci reálné aplikace v prostředí webového prohlížeče. Na jednotlivá tlačítka jsou přidány odkazy na příslušné obrazovky, které umožňují přechody mezi obrazovkami jako v reálné aplikaci.

Tento export byl předložen vybraným respondentům. Ti po vyzkoušení simulace přednesli návrh na vytvoření základního filtrování. Po konzultaci bylo vybráno, že filtrování bude možné zvolit podle taxonomických tříd a podle umístění zvířete v Zoo Praha.

Dále bylo konzultováno možné rozšíření funkcionality, zmíněné v sekci 2.1.

- Interaktivní **mapa Zoo Praha** s vyznačenými lokalitami,
- **kvíz** či vědomostní hra,
- informace o možnosti **adopce** zvířat.

Po krátkém prozkoumání aktuálních řešení byla zvolena varianta implementace mapy. Kvízů a vědomostních her, včetně zaměřených na tématiku zvířat, již existuje mnoho. Pro příklad stačí zadat do vyhledávání na *Google Play* (obchod s aplikacemi pro Android) heslo „animal quiz“ (viz [7]).




Obrázek 2.2: Wireframe: Hierarchie stránek - druhá iterace

Všichni respondenti se shodli, že by se aplikace vytvořená v této práci měla zaměřit na praktické využití, např. při návštěvě pražské zoo. K tomu je interaktivní mapa ideální funkcionalitou.

Informace o adopci zvířat nebyla pro nikoho z respondentů zajímavá.

Výstupem této iterace byly tyto navržené změny: implementace základním filtrů podle taxonomie a umístění a implementace mapy s vyznačenými umístěními zvířat v Zoo Praha.

### 2.2.2 Druhá iterace

V druhé iteraci byly provedeny tyto změny:

- Do hlavního menu přidáno tlačítko **Mapa** (obrázek C.10).
- Přidání obrazovky **Mapa zoo** (obrázek C.15).
- Na obrazovku **Chov v Zoo** přidáno tlačítko **Ukázat na mapě** (obrázek C.12).
- Přidání obrazovky **Menu filtrů** (obrázek C.11).
- Přidání obrazovky pro výběr filtrované třídy (obrázek C.13).
- Přidání obrazovky pro výběr filtrovaného umístění (obrázek C.14).

Výsledná hierarchie obrazovek je na obrázku 2.2.

Opět byl vytvořen export do HTML stránky a předložen respondentům. Z jejich reakcí byly navrženy tyto úpravy:

- Přidání obrázku zvířete ke každé položce v seznamu zvířat.
- Přidání vyhledávání zvířat podle názvu.
- Změna seznamu akcí na kalendář akcí.

### 2.2.3 Třetí iterace

Ve třetí iteraci byly zpracovány návrhy z předchozí iterace. Hierarchie stránek se nezměnila a je stejná jako na obrázku 2.2. Na stránce seznamu akcí byl seznam vyměněn za kalendář (viz obrázek C.18). Na stránku seznamu zvířat byly k jednotlivým zvířatům přidány obrázky (viz obrázek C.16). Na stránku seznamu byla přidána možnost vyhledávání zvířat. Jelikož model neumožňuje interaktivní vkládání textu, byla odpovídající stránka navržena jako výsledek po hledání po vyplnění hledaného textu (viz obrázek C.17).

Model byl znovu exportován do HTML stránky a předán respondentům. Ti již neměli žádné připomínky a model byl označen za finální. Tento model byl použit jako vzor pro implementaci výsledné aplikace.

## 2.3 Návrh serveru

Tato sekce se zabývá návrhem serverové části této práce. Server má stahovat data ze zdrojových dat a nabízet je přes REST API mobilní aplikaci.

Ze zadání je jasné, že server lze rozdělit na dvě části: stahování dat a REST API. Samozřejmě je potřeba stažená data ukládat, proto je nutné přidat další část, a to databázi.

V následujících podsekcích jsou popsány návrhy jednotlivých částí.

### 2.3.1 Databáze

Pro uchování dat je potřeba implementovat databázi. Jelikož zdrojová data nejsou složitá a ani práce s těmito daty nevyžaduje žádné složité konstrukce, lze databázi navrhnout velice jednoduše.

Ve zdrojových datech je pro každý využívaný zdroj použita jedna kolekce (tabulka). Tento přístup lze využít i v návrhu databáze. Struktura databáze tedy bude v základu kopírovat strukturu zdrojových dat.

Aby však bylo možné udržovat informaci o aktuálnosti dat, bude každá kolekce obsahovat navíc několik položek:

Položku `deleted`, která bude označovat, zda byla položka smazána ze zdrojových dat (hodnota `true`), nebo zda je aktuální (hodnota `false`).

Položku `last_updated`, která obsahuje datum poslední aktualizace položky.

Položku `last_visited`, která obsahuje datum posledního stažení položky ze zdrojových dat. Položka nemusí být zároveň aktualizována. Slouží pro rozeznání, zda je položka stále přítomna ve zdrojových datech.

Pro rychlý přístup k informacích o poslední aktualizaci jednotlivých tabulek bude použita tabulka `Versions`. Ta bude obsahovat jeden řádek s daty poslední aktualizace každé z ostatních tabulek.

### 2.3.2 Stahování a aktualizace dat

Ze zdrojových dat je potřeba data stáhnou a uložit do databáze a následně je průběžně aktualizovat.

Vzhledem k frekvenci obnovy zdrojových dat budou data stahována a ukládána do databáze při spuštění serveru a v jednodenních intervalech.

Při stahování se bude kontrolovat, zda již nejsou data v databázi obsažena. V případě dat z lexikonu se tato kontrola bude provádět pomocí id zvířete, v případě ostatních tabulek se tato kontrola provede porovnáním všech vlastností.

Pokud data již v databázi jsou, aktualizuje se jim datum ve vlastnosti `last_visited`. Poté se stažená data porovnají s daty v databázi. Pokud se v nějaké položce liší, pak je tato položka aktualizována a zároveň je aktualizována vlastnosti `last_updated`.

Pokud se data v databázi nenacházejí, jsou do ní uložena. Zároveň jsou jim nastaveny položky `last_visited` a `last_updated` na aktuální datum a čas.

Po stažení každé kolekce se projdou všechny položky dané kolekce v databázi. Pokud mají některé položky hodnotu vlastnosti `last_visited` starší než jeden den, označí se jako smazané nastavením vlastnosti `deleted` na `true`.

Stáří jednoho dne je zvoleno z důvodu zachování konzistence dat. Smazání dat se tedy na serveru provede až po druhém dni, kdy data chybí i ve zdrojových datech. Pokud by například byla na zdrojových datech prováděna údržba a data by byla dočasně smazána ve chvíli jejich stahování, a následně do jednoho dne opět přidána, na serverovém API se to neprojeví. Případné smazané položky, které zůstanou další den v databázi, nejsou, vzhledem k charakteru dat, problém.

S kolekcí zvířat se bude zároveň stahovat obrázek ke každému zvířeti. Ten bude následně zmenšen a uložen do databáze společně s ostatními daty zvířete. Tyto obrázky budou použity mobilní aplikací místo původních obrázků, které by, vzhledem ke své původní velikosti, uživatele stály velké množství dat.

### 2.3.3 REST API

REST API je navrženo pomocí webové aplikace *apiary*[8]. S apiary je možné vytvářet návrhy API pomocí jazyka *API Blueprint*[9]. Ten umožňuje popsat navrhované API, včetně vzorových požadavků a odpovědí.

Apiary dále umožňuje testovat navržené API, jak na vytvořených testovacích datech, tak i na produkčním prostředí, pokud je dodáno jeho URL adresa.

Zdrojový Blueprint lze nalézt na přiloženém CD. Vytvořený návrh API na apiary lze nalézt na [10].

REST API bude obsahovat pouze GET metody, a to tyto:

- `/api/actions` API pro kolekci akcí.
- `/api/rss` API pro kolekci novinek.
- `/api/animals` API pro kolekci zvířat (bez obrázků).
- `/api/animals/images/:id` API pro stahování upravených obrázků, `:id` je ID zvířete v databázi.

## 2.4 Návrh mobilní aplikace

Návrh mobilní aplikace lze rozdělit na dvě části: uživatelské rozhraní (frontend) a vnitřní logiku (backend). Návrh uživatelského rozhraní vychází z finálního modelu ze sekce 2.2 a není zde dále rozebírán.

Vzhledem k tomu, že aplikace nemá provádět žádné výpočty a nemá žádnou složitou vnitřní logiku, lze backendovou část rozdělit pouze na dvě další části. Jendou je databáze pro uchování dat a druhá je logika pro stahování a aktualizaci dat.

Návrh databáze je jednoduchý, databázový model bude kopírovat strukturu databázového modelu serverové části (viz 2.3.1). V mobilní části není nijak potřeba tento model měnit a shodný model je výhodný pro jednodušší zpracování stahovaných dat.

Stahování dat bude probíhat ze serverového REST API. Stahování bude využívat HTTP hlavičky `if-modified-since` a `last-modified`. Při stažení dat se uloží datum z hlavičky `last-modified` a použije se při aktualizaci dat při dotazu na server v hlavičce `if-modified-since`. To umožní stažení jen změněných dat. Při ukládání stažených dat se bude kontrolovat hodnota vlastnosti `deleted` každé položky. Pokud bude nastavená na `true`, pak se daná položka neuloží, ale z mobilní databáze se smaže (pokud tam byla). Tím se zaručí aktuálnost dat v mobilní databázi.

Kapitola 3

# Implementace

Tato kapitola obsahuje popis implementace serveru a mobilní aplikace, volbu technologií a popis použitých knihoven.

## 3.1 Implementace serveru

Tato sekce popisuje implementaci serverové části této práce. Sekce obsahuje volbu technologií použitých pro implementaci, popis implementace rozdělený do tří částí (databáze, stahování a aktualizace dat, REST API) a popis použitých knihoven (resp. *npm* (viz 6.1.2) balíčků).

V příloze D je na obrázku D.12 vyobrazen UML diagram znázorňující strukturu serveru, která je dále popsána v této kapitole.

### 3.1.1 Volba technologií

Server je v souladu se zadáním implementován v technologii *Node.js* (viz 6.1).

Jako databáze byla zvolena *NoSQL* databáze (není založená na SQL a využívá jiné prostředky než tradiční relační databáze) *MongoDB* (viz 6.3). Ta je vhodná hlavně díky jednoduchosti dat. Protože databáze bude obsahovat jen několik jednoduchých tabulek, mezi nimiž nejsou žádné vazby (a bylo by zbytečné je vytvářet) je MogoDB ideální volbou. Navíc integrace s Node.js je díky npm balíčku *Mongoose* (viz 3.1.5.1) jednoduchá. Také není třeba vytvářet databázový model zvlášť, lze jej popsat v Node.js a jednotlivé tabulky se v databázi vytvoří prostým přidáním položek.

### 3.1.2 Databáze

Databáze obsahuje tabulky pro každou datovou sadu, které kopírují datové sady v opendatech. Dále každý dokument obsahuje datum a čas poslední změny (`last_updated`), čas posledního výskytu v opendatech (`last_visited`)

a značku, zda se má dokument smazat (`deleted`, tzn. už se nevyskytuje v opendatech). Tabulky jsou definovány v souborech:

#### 3.1.2.1 AnimalModel.js

Soubor `AnimalModel.js` obsahuje definici modelu pro tabulku lexikonu zvířat. Tato tabulka obsahuje všechna data stažená ze zdrojových opendat. Navíc obsahuje obrázek zvířete, stažený z URL adresy, která se nachází v HTML tagu v položce `description`, a zmenšený, připravený pro mobilní aplikaci.

Model:

```
_id: { type: Number, required: true, unique: true },//ID zvířete
Title: String,  // název zvířete
LatinTitle: String, // latinský název
ImageAlt:String,
ImageUrl: String, // URL obrázku
Continent: String, // kontinent výskytu
Class: String, // taxonomická třída
Order: String, // taxonomický řád
spreadNote: String, // poznámka k~výskytu
Biotop: String, // biotop výskytu
BiotopesNotes: String, // poznámka k~biotopu
Food: String, // potrava
FoodNotes: String, // poznámka k~potravě
Proportions: String, // proporce
Reproduction: String, // rozmnožování
Attractions: String, // zajímavost
ProjectsNote: String, // info o~ohrožení
Breeding: String, // info o~chovu v~pražské zoo
LocalitiesTitle: String, // název pavilonu/výběhu v~pražské zoo
LocalitiesUrl: String, // url pavilonu/výběhu v~pražské zoo
Description: String, // popis zvířete
updated_at: Date, // datum poslední změny zdrojových dat
last_visit: Date, // datum posledního výskytu ve zdrojových
datech
deleted: Boolean, // indikuje, zda se již ve zdrojových
datech více nevyskytuje
image: Buffer  // zmenšený obrázek
```

#### 3.1.2.2 RSSModel.js

Soubor `RSSModel.js` obsahuje definici modelu tabulky pro novinky. Ta obsahuje novinky z RSS kanálu Zoo Praha.

Model:

```
title: String,  // název zprávy
link: String, // URL zprávy
description: String, // popis zprávy
date: Date,   // datum zprávy
updated_at: Date, // datum poslední změny zdrojových dat
last_visit: Date, // datum posledního výskytu ve zdrojových
datech
deleted: Boolean, // indikuje, zda se již ve zdrojových
datech více nevyskytuje
```

#### 3.1.2.3 ActionModel.js

Soubor `ActionModel.js` obsahuje definici modelu pro tabulku akcí. Tabulka obsahuje akce probíhající v Zoo Praha, které jsou vypsané v opendatech. Tabulka navíc obsahuje URL adresu odkazu, která je parsována z popisu akce ve zdrojových datech, tj. z vlastnosti `description`.

Model:

```
start: Date,  // začátek akce
end: Date,  // konec akce
summary: String, // název/shrnutí akce
description: String, // popis akce
url: String,   // url akce
updated_at: Date, // datum poslední změny zdrojových dat
last_visit: Date, // datum posledního výskytu ve zdrojových
datech
deleted: Boolean, // indikuje, zda se již ve zdrojových
datech více nevyskytuje
```

#### 3.1.2.4 VersionsModel.js

Soubor `VersionsModel.js` obsahuje definici modelu pro tabulku s jedním záznamem, který obsahuje datum a čas poslední aktualizace výše popsaných tabulek. Používá se pro rychlé zjištění poslední aktualizace při dotazu na API s hlavičkou `if-modified-since`.

Model:

```
_id: String,  // ID, požívá se jeden záznam s~ID "v1"
animalUpdatedAt: Date, // poslední update tabulky zvířat
actionUpdatedAt: Date, // poslední update tabulky akcí
rssUpdatedAt: Date // poslední update tabulky novinek
```

### 3.1.3 Stahování a aktualizace

Tato část popisuje implementaci stahování a aktualizace dat na serveru ze zdrojových opendat. Sekce je rozdělena podle jednotlivých datových sad. Sa-

motné stahování dat je implementováno v souboru `GetData.js`. Ukládání stažených dat do databáze je implementováno v souboru `DBHandler.js`.

#### 3.1.3.1 Akce

Stahování akcí ze zdrojových dat je implementováno funkcí `getActions` v souboru `GetData.js`.

Datová sada Akce se stahuje ve formátu JSON ze stránek Zoo Praha. Data ovšem nesplňují standardní JSON formát. Formát dat je pole objektů, vnější pole ovšem nemá název. Pro úpravu do správného formátu je nutné chybějící název přidat, tedy:

```
body = "{ \"data\":" + body + "}";
```

kde proměnná `body` jsou stažená data.

Po této úpravě je možné data parsovat pomocí metody `JSON.parse(body)`, která vrací data převedená na javascriptové objekty.

Vytvořené objekty se poté po jednom ukládají do databáze. Před vlastním uložením se zjišťuje, zda již není příslušná akce v databázi. To probíhá voláním funkce `saveAction`, implementované v souboru `DBHandler.js`, na jednotlivé stažené položky. Obojí je popsáno dále.

Popis akce (pole `description`) obsahuje HTML tagy. Ty jsou odebrány pomocí metody `replace` a regulárního výrazu `<[^>]*>|( )`

Před tímto odebráním se nejdříve popis prohledá, zda neobsahuje URL odkaz. To probíhá pomocí regulárního výrazu `href=\"[^\"]*\"`. Výsledek se ještě znovu zkontroluje, zda to není odkaz pro odeslání e-mailu, a to pomocí regulárního výrazu výrazu `mailto`. Poté proběhne kontrola, zda je nalezené URL odkaz na vnější stránku (obsahuje `http://` nebo `https://`) nebo na stránku zoo (bez `http`). Pokud je odkaz na stránku zoo, přidá se před nalezený URL odkaz ještě `https://www.zoopraha.cz/`. Následně se výsledek uloží do vlastnosti `url`.

Akce ve zdrojových datech nemají žádné id, ani jiné unikátní vlastnosti. Z toho důvodu se dotaz, zda již databáze obsahuje akci, skládá ze všech vlastností dané akce. Pokud se akce v databázi nachází, je jí aktualizována vlastnost `last_visited`, která značí, kdy byla daná akce naposledy stažena ze zdrojových dat. Pokud se akce v databázi nenachází, je do ní vložena.

Po kontrole a uložení všech stažených dat se najdou v databázi všechny akce, které mají pole `last_visited` s menším datem, než je čas této aktualizace minus jeden den (vysvětleno v 2.3.2). Tyto akce se označí jako smazané, pomocí zápisu hodnoty `true` do vlastnosti `deleted`. To je implementováno funkcí `markAsDeletedAction` v souboru `DBHandler.js`.

#### 3.1.3.2 Novinky

Stahování novinek ze zdrojových dat je implementováno funkcí `getRss` v souboru `GetData.js`.

Datová sada ve formátu RSS odkazuje na stránky Zoo Praha, kde se nachází ve formátu RSS kanálu. Data z této stránky se stahují a parsují pomocí balíčku *rss-parser* 3.1.5.4 metodou `parseURL`. Ta stáhne data z dané stránky a vytvoří z nich javascriptové objekty. Ty se po jednom ukládají do databáze pomocí funkce `saveRss` implementované v souboru `DBHandler.js`.

Popis se stejně jako u akcí zbaví případných HTML tagů.

Stejně jako u akcí, nemají tyto data žádný identifikátor. Proto se dotaz na přítomnost daných dat v databázi tvoří opět ze všech vlastností.

Kontrola přítomnosti, změna pole `last_visited`, ukládání a označování dat jako smazané probíhá stejně jako u akcí. Pro novinky je toto implementováno funkcí `markAsDeletedRss` v souboru `DBHandler.js`.

#### 3.1.3.3 Lexikon zvířat

Stahování lexikonu ze zdrojových dat je implementováno funkcí `getLexicon` v souboru `GetData.js`.

Zdrojová datová sada Lexikon zvířat je ve stejném formátu jako formát datové sady akcí. Proto je stejně jako u akcí potřeba před parsováním doplnit název pro vnější pole. Stejně tak se poté data parsují do javascriptových objektů a následně postupně ukládají do databáze, což je popsáno dále. Ukládání je implementováno funkcí `saveAnimal` v souboru `DBHandler.js`.

Z příchozích dat je vytvořen objekt typu `AnimalModel`, který představuje řádek tabulky (prozatím nevložený do databáze). Popis zvířete (pole `description`) je zkontrolován, zda neobsahuje URL adresu obrázku. Kontrola se provádí pomocí regulárního výrazu `/images[^\\]*jpg/i`. Případná URL adresa obrázku je uložena do vlastnosti `ImageUrl`.

Poté jsou položky `desription`, `attractions` a `breeding` zbaveny HTML tagů (pomocí metody `replace` a regulárního výrazu `<[^>]*>|( )`).

Před uložením se v databázi hledá záznam s `_id` stejným jako má nově vytvořený model. Pokud se nic nenajde, je nový model uložen. Pokud se v databázi najde záznam se stejným `_id`, pak se porovnávají všechny položky starého a nového záznamu. Pokud jsou stejné, pak se jen zapíše aktuální datum a čas do pole `last_visited`. Pokud stejné nejsou, přiřadí se nové vlastnosti do starého záznamu a změní se i pole `last_updated`.

Po každém zpracování záznamu, pokud má dané zvíře URL adresu obrázku, se stáhne daný obrázek a pomocí npm balíčku *sharp* (viz 3.1.5.3) se zmenší a následně uloží do databáze k danému zvířeti.

Po kontrole a uložení všech stažených dat se v databázi hledají všechny záznamy zvířat, která mají pole `last_visited` s menším datem, než je čas této aktualizace minus jeden den (vysvětleno v 2.3.2). Tyto položky se označí jako

smazané, pomocí zápisu hodnoty `true` do pole `deleted`. To je implementováno funkcí `markAsDeletedAnimals` implementované v souboru `DBHandler.js`.

### 3.1.4 REST API

REST API je implementováno s využitím npm balíčku Express (viz 3.1.5.2) v hlavním souboru `server.js`. API je vytvořeno podle návrhu, viz 2.3.3.

Jednotlivé dotazy na celé kolekce vždy nejdříve zkontrolují, zda je vložena hlavička `if-modified-since`. Pokud ano, pak jsou vráceny jen data, které mají ve vlastnosti větší datum než je ve zmíněné hlavičce. Pokud taková data neexistují, pak je vrácen status kód `304 Not modified`.

Pokud hlavička `if-modified-since` v dotazu nebyla, vrací se všechna data dotazované kolekce.

Pokud se vrací data, je do odpovědi zapsána hlavička `last-modified`, která obsahuje datum a čas poslední změny dat.

Dotaz na obrázek vrací buď obrázek v binární podobě s nastavenou hlavičkou `content-type` na `image/jpg`, nebo, pokud obrázek v databázi není, vrací status kód `404 Not found`.

### 3.1.5 Použité npm balíčky

Node.js používá tzv. npm balíčky (viz 6.1.2). Následující část popisuje balíčky použité při implementaci serverové části této práce.

#### 3.1.5.1 Mongoose

*Mongoose* je balíček pro modelování MongoDB objektů navržený pro asynchronní prostředí. Mongoose poskytuje řešení založené na schématech, které pomohou modelovat data aplikace. Zahrnuje převod typů, validace, tvorbu dotazů, zachytávání business logiky a další. [11]

#### 3.1.5.2 Express

*Express* je rychlý, flexibilní, minimalistický webový framework pro Node.js. Poskytuje velké množství funkcionalit pro rychlou tvorbu webových a mobilních aplikací. Také disponuje mnoha možnostmi pro rychlou a jednoduchou tvorbu API.[12]

V této práci je použit pro vytvoření REST API (viz 3.1.4).

#### 3.1.5.3 Sharp

Balíček *Sharp* slouží ke zpracování a rychlým úpravám obrázků. Typické použití balíčku Sharp je konverze velkých obrázků v běžných formátech na menší, web-friendly formáty. [13]

V této práci je použit pro zmenšení původních obrázků ze zdrojů Zoo Praha. To způsobuje velké ušetření přenesených dat do mobilní aplikace. Více viz 3.1.3.3.

#### 3.1.5.4 rss-parser

*Rss-parser* je balíček pro jednoduché parsování RSS do javascriptových objektů. [14]

V této práci je použit pro parsování RSS ze zdrojových dat Zoo Praha (viz 3.1.3.2).

## 3.2 Implementace mobilní aplikace

Tato sekce popisuje implementaci mobilní aplikace. Implementace aplikace lze rozdělit na frontendovou a backendovou část.

Frontendová část obsahuje:

- `Menu` Úvodní stránka s rozcestím
- `Lexikon` Obsahuje seznam zvířat a informace o nich
- `Novinky` Obsahuje seznam novinek
- `Akce` Obsahuje kalendář akcí
- `Mapa` Obsahuje interaktivní mapu Zoo s vyznačenými pavilony

Backendová část se ze skládá z:

- `Databáze` Implementace mobilní databáze
- `Stahování a aktualizace dat` Logika implementující komunikaci se serverem a aktualizaci dat

### 3.2.1 Volba technologií

Pro operační systém Android lze vyvíjet v několika možných technologiích. Nejběžnější varianta je implementace v programovacím jazyce Java s pomocí vývojového prostředí *Android Studio* (viz 6.8). Další z variant je např. vývoj v jazyce C# s platformou *Xamarin* (viz 6.9) ve vývojovém prostředí Visual Studio.

Pro tuto práci byla zvolena platforma Xamarin a jazyk C# a to z několika, převážně osobních a studijních, důvodů: a

- Větší zkušenost s jazykem C# než Java.
- Seznámení s novou technologií.

- S malou zkušeností s Android studiem z předmětu BI-AND možnost srovnání vývoje v různých technologiích.
- Potenciál rozšíření aplikace pro další platformy (iOS, Windows Phone).

Pro implementaci mobilní databáze je také možné použít různé technologie. Standardem mobilních databází je *SQLite* (viz [15]).

V této práci je však použita databáze *Realm* (viz 6.10). Opět z důvodu poznání nové technologie. A stejně jako u serverové části je vhodná i vzhledem k jednoduchosti dat. Další výhodou je jednoduchost implementace a použití.

### 3.2.2 Databáze

V této sekci je popsána implementace databáze v mobilní aplikaci.

V příloze D je na obrázku D.10 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se implementace databáze.

#### 3.2.2.1 Databázový model

Jako databáze je použita *Realm Database* (viz 6.10). Pro její použití je potřeba vytvořit model. Ten je implementován v souboru `RealmDatabaseModel.cs`. Soubor obsahuje čtyři třídy: `Animal`, `Rss`, `Action` a `Versions`. Všechny tyto třídy dědí ze třídy `RealmObject`.

Třídy `Animal`, `Rss` a `Action` mají totožné vlastnosti stejných typů a názvů jako má serverový databázový model. To zjednodušuje mapování stažených dat.

Třída `Versions` obsahuje id a položky `animalLastDate`, `rssLastDate`, `actionLastDate`. Ty slouží k uchování data z HTTP hlavičky `last-modified` při stahování dat ze serveru. Tabulka obsahuje jen jeden řádek s těmito daty.

Implementace těchto čtyř tříd stačí Realm databázi pro vytvoření příslušných tabulek. Nic víc není potřeba nastavovat ani vytvářet.

#### 3.2.2.2 Přístup k databázi

Databáze je od zbytku aplikace oddělena rozhraním `IDBHandler`. V něm jsou popsány metody, které je možné používat pro přístup k datům. Metody by měly obsluhovat zápis kolekcí typu `Animal`, `Rss` a `Action` do databáze, čtení celých kolekcí i jednotlivých prvků podle id, metody pro vybírání akcí podle měsíců nebo data, zápis a čtení hodnot v tabulce `Versions` apod.

Z aplikace se volá statická metoda `GetHandler` třídy `DBHandlerGetter`. Tato třída nemá jiný účel, než ten, že zmíněnou metodou vrací aktuální implementaci rozhraní `IDBHandler`. Pokud by tedy bylo potřeba změnit databázi, stačí pouze implementovat interface `IDBHandler` a ve třídě `DBHandlerGetter` vracet tuto novou implementaci.

Pro přístup k Realm databázi je implementován `RealmDBHandler`, který implementuje všechny potřebné metody.

### 3.2.3 Stahování a aktualizace dat

Tato sekce popisuje způsob stahování dat ze serverového REST API.

V příloze D je na obrázku D.11 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se implementace stahování a aktualizace dat.

#### 3.2.3.1 Komunikace se serverem

Základem pro stahování dat je třída `RestService`, která obstarává komunikaci se serverem. Třída obsahuje metody `GetAllAnimals`, `GetAllRss` a `GetAllActions`.

V každé metodě se nejdříve zjistí datum poslední úpravy dané tabulky z tabulky `Versions`. To je pak předáno příslušné metodě (`RequestAnimals`, `RequestActions` a `RequestRss`), která provádí příslušný dotaz (request). Do dotazu je datum poslední úpravy vloženo do hlavičky `if-modified-since`.

Pro komunikaci se serverem je použita knihovna *RestSharp* (viz 3.2.9.1). Ta umožňuje poslat dotaz na danou URL adresu a data v odpovědi rovnou zpracovat a uložit (deserializovat) do objektů daného typu, v tomto případě do kolekce `List<Animal>` (resp. `List<Rss>` nebo `List<Action>`).

Po získání odpovědi ze serveru se nejdříve zkontroluje, zda byla přijata HTTP hlavička `last-modified`. Pokud ano, pak se do databáze do tabulky `Versions` do příslušného sloupce vloží datum z této hlavičky.

Pokud je status odpovědi `200 OK`, pak metoda vrací stažená data v kolekci typu `List`.

#### 3.2.3.2 Stahování dat

Po prvním spuštění aplikace, resp. v případě, že v mobilní databázi nejsou žádná data, se zobrazí uživateli dialog, zda chce stáhnout data. Po potvrzení tohoto dialogu se zobrazí dialog s informací o stavu stahování (tzv. progress dialog).

Zobrazení dialogu o stavu stahování a samotné stažení obstarává třída `CompleteDownloader`. Ta dědí ze třídy `Android.OS.AsyncTask`. V metodě `OnPreExecute`, která se spouští před samotným stahováním, zobrazí dialog se stavem stahování. V metodě `OnPostExecute`, která se spouští po ukončení stahování, se tento dialog zavírá.

Metoda `RunInBackground`, běžící v samostatném vlákně, volá privátní metodu `DownloadData`. V této metodě se postupně pouští metoda `DownloadData` na třídách `AnimalDownloader`, `ActionDonwloader` a `RssDownloader`. Mezi voláním těchto metod se aktualizuje dialog se stavem stahování.

Třídy `AnimalDownloader`, `ActionDonwloader` a `RssDownloader` obsahují metodu `DownloadData`, která volá příslušnou metodu ze třídy `RestService` (viz 3.2.3.1). Vrácenou kolekci dat pak dává jako parametr příslušné metodě pro zápis dané kolekce v aktuální implementaci `IDBHandler`.

Aktuálně se používá `RealmDBHandler`. Ten při ukládání dat prochází kolekci a kontroluje vlastnost `deleted`. Pokud je hodnota `deleted` rovna `true`, pak se pokusí najít prvek s daným id v databázi, a pokud se tam nachází, pak ho smaže. Pokud je hodnota `deleted` rovna `false`, pak ho do databáze přidá.

Realm databáze má pro přidání metodu `Add`, která má volitelný parametr `update`. Ten je nastaven na `true`, což má za následek, že pokud daný záznam již v databázi je, provede se jeho aktualizace (update), pokud není, vytvoří se nový (insert).

#### 3.2.3.3 Aktualizace dat

Po spuštění, pokud aplikace obsahuje již stažená data, se nezobrazuje žádný dialog, pouze se na pozadí aplikace data aktualizují.

Aktualizaci obstarává třída `BackgroundUpdateDownloader`, která dědí ze třídy `Android.OS.AsyncTask`. Na rozdíl od třídy `CompleteDownloader` nezobrazuje dialog, jinak se chová stejně. Ve vlastním vlákně, takže uživatel nic nepozná a může aplikaci dál používat, postupně volá metodu `DownloadData` na třídách `AnimalDownloader`, `ActionDonwloader` a `RssDownloader`.

Díky používání HTTP hlaviček `if-modified-since` a `last-modified` se zpravidla, kromě prvního stažení, nemusí stahovat všechna data.

### 3.2.4 Lexikon

Hlavní část, lexikon zvířat, se skládá ze dvou aktivit: `AnimalListActivity` a `AnimalDetailActivity`.

#### 3.2.4.1 AnimalListActivity

`AnimalListActivity` představuje obrazovku se zobrazeným seznamem zvířat. Samotný obsah je implementován ve fragmentu `AnimalListFragment`. Ten obsahuje `ListView`, ve kterém se zobrazují jednotlivá zvířata. `ListView` je objekt, který zobrazuje jednotlivá vnitřní `View` vertikálně nad sebou, a kterým je možno posouvat.

V příloze D je na obrázku D.9 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

Po kliknutí, respektive doteku, na vybrané zvíře se zobrazí detail tohoto zvířete. Detail se zobrazí buď v nové aktivitě `AnimalDetailActivity` nebo při dostatečné šířce obrazovky ve stejné aktivitě v druhém fragmentu (viz 3.2.4.3).

Pro vyplnění jednotlivých `View` v `ListView` je implementován adaptér `AnimalListViewAdapter`. Ten si v konstruktoru načte z databáze zvířata pomocí metody `GetFilteredAnimals` a základní informace si uloží jako položky typu `AnimalListBasics` do kolekce `List`. Třída `AnimalListBasics` obsahuje id zvířete, jeho název a URL adresu obrázku zvířete.

Metoda `GetFilteredAnimals` načítá z databáze zvířata a filtruje je podle filtru nastaveného v aktivitě `AnimalListActivity`. Filtr se v aktivitě nastavuje při jejím vytvoření a je používán pro filtrování podle taxonomických tříd a podle umístění zvířete v zoo. Druhy filtrů jsou `ALL_ANIMALS`, který vrací všechna zvířata seřazená podle abecedy, `TAXONOMY`, který vrací zvířata filtrovaná podle taxonomické třídy, kde hodnota filtru (filtrovaná taxonomická třída) je uložena ve vlastnosti `filterValue` aktivity `AnimalListActivity`, a `LOCALITY`, která vrací zvířata filtrovaná podle umístění v zoo, kde hodnota filtru (filtrované umístění) je uložena ve vlastnosti `filterValue` aktivity `AnimalListActivity`.

Třída `AnimalListViewAdapter` dědí od třídy `BaseAdapter` a přepisuje její metody `GetItem`, `GetItemId` a `GetView`.

Metoda `GetItemId` vrací Id zvířete na parametrem dané pozici. Metoda `GetItem` vrací název zvířete na parametrem dané pozici.

Metoda `GetView` se stará o vytváření a případnou recyklaci `View` pro `ListView`. Díky této metodě se nemusí při posouvání `ListView` při každém posunutí vytvářet nové `View`, ale je použito staré, ve kterém se jen změní text a obrázek, což zásadně šetří využívanou paměť a její správu.

V metodě se z parametru `convertView` vezme vlastnost `Tag`, přetypuje se na `AnimalListViewAdapterViewHolder` a přiřadí do proměnné `holder`. Třída `AnimalListViewAdapterViewHolder` slouží pro uchování zobrazovaných `View`, konkrétně `TextView` pro název a `ImageView` pro obrázek.

Pokud má `holder` hodnotu null, pak se musí vytvořit nový `holder` a vytvořit nové `View`. V aplikaci jsou implementovány dva typy zobrazení seznamu zvířat, podle kterých se vrací konkrétní `View`. Které `View` se má vytvořit se rozhoduje podle proměnné `viewType`, která se nastavuje v konstruktoru. Proměnná `viewType` je výčtového typu `ViewType`, který obsahuje hodnoty `LISTVIEW` a `CARDVIEW`.

Pokud je `viewType` nastavená na LISTVIEW, pak se vytváří `View` popsané v souboru `AnimalListRow.axml` v adresáři `Resources/Layout`. Toto `View` obsahuje `LinearLayout` s horizontální orientací, kde v levé části je `ImageView` pro obrázek a v pravé části `TextView` pro název zvířete. Velikost každého `View` je dána šířkou obrazovky a výška je nastavená na 80 dp. V tomto zobrazení jsou jednotlivé položky relativně malé a na jednu obrazovku se jich vejde více, je tedy jednodušší mezi nimi listovat.

Druhou možností, pokud je proměnná `viewType` nastavená na `CARDVIEW`, pak se vytváří `View` popsané v souboru `AnimalCardView`. Toto `View` obsahuje `CardView`, které obsahuje vertikální `LinearLayout`, ve kterém je v horní části `ImageView` pro obrázek a ve spodní části `TextView` pro název zvířete. Velikost jednotlivých `View` je dána šířkou obrazovky a výška je variabilní podle velikosti daného obrázku. Díky takto zobrazených obrázkům je toto zobrazení vhodné pro jejich prohlížení, nehodí se však moc pro rychlé listování.

Vybrané `View` se přes `inflater.Inflate` přiřadí do vracené proměnné `view`. Vnitřní `TextView` a `ImageView` se uloží do nově vytvořené proměnné

`holder`, která se následně uloží do `view.Tag`.

Pokud `holder` není `null`, pak není nutné jej vytvářet a do `View` uložených v proměnné `holder` se přiřadí nové hodnoty.

V obou případech se nakonec ho `holder.TextView` přiřadí název daného zvířete a do `holder.ImageView` se nahraje obrázek tohoto zvířete.

Pro nahrání obrázku je použita knihovna *Picasso*(viz 3.2.9.2). Pomocí této knihovny se stáhne obrázek ze serveru přes API `api/animals/images/{id}` a vloží do daného `ImageView`. Knihovna Picasso se stará o paměťové i diskové uchování dat v mezipaměti (tzv. kešování), které umožňuje rychlé načítání a šetření dat, protože obrázky se nemusí pořád znovu stahovat. Více o knihovně v sekci 3.2.9.2.

Adaptér také umožňuje rychlé listování v seznamu zvířat (tzv. fast-scroll), které lze většinou najít např. v telefonním seznamu. To je umožněno implementací rozhraní `ISectionIndexer`. V praxi to znamená, že po přejetí prstem po pravém kraji seznamu se zobrazí „bublina“ s prvním písmenem zvířat v dané části seznamu. Po posunu dolů nebo nahoru se seznam posouvá a v „bublině“ se zobrazuje aktuální první písmeno aktuálně zobrazovaných zvířat.

Pro implementaci rozhraní `ISectionIndexer` je nutné implementovat metody `GetPositionForSection`, `GetSectionForPosition` a `GetSections`.

Pro implementaci těchto metod jsou potřeba ještě tři proměnné. První je `alphaIndex` typu `Dictionary<string, int>`, kde klíč je název sekce (tj. první písmeno názvu zvířete) a hodnota je pořadí prvního zvířete s daným prvním písmenem v seznamu. Další je proměnná `sections` typu `string[]`, tedy pole řetězců, které obsahuje všechny sekce, tj. všechna první písmena. Poslední je proměnná `sectionsObjects`, která je typu pole Java objektů, tedy `Java.Lang.Object[]`. Ta obsahuje stejné hodnoty jako proměnná `sections`, jen je typu, který vyžaduje rozhraní `ISectionIndexer`.

Nastavení těchto proměnných probíhá v metodě `createIndex`, která se spouští po každém načtení zvířat do seznamu, tj. v konstruktoru adaptéru a po filtrování zvířat (viz dále). V této metodě se prochází všechna zvířata v listu `animalsBasics`. Zvířata v listu jsou seřazena podle abecedy. Pro každé zvíře se kontroluje první písmeno názvu. Pokud je toto písmeno jiné než předešlé, pak se uloží do proměnné `aplhaIndex` jako klíč a jako hodnota se použije pořadí daného zvířete v seznamu. Pokud je dané písmeno stejné jako předešlé, nic se neukládá a pokračuje se dál. Po projití všech zvířat se všechny klíče uloží do proměnné `sections` a následně i do proměnné `sectionsObjects`. Tím jsou tyto proměnné nastaveny.

Metoda `GetSections` vrací pole sekcí v proměnné `sectionsObjects`.

Metoda `GetPositionForSection` vrací pozici prvního zvířede v sekci na indexu daném parametrem. V proměnné `sections` najde podle parametrem daného indexu název dané sekce (první písmeno názvu zvířete) a podle něj najde v proměnné `alphaIndex` pozici a vrátí ji.

Metoda `GetSectionForPosition` naopak vrací index sekce pro danou pozici.

V seznamu zvířat je možné vyhledávat podle názvu zvířete. To je umožněno implementací rozhraní `IFilterable`. Pro jeho implementaci je potřeba implementovat *getter* na vlastnost `Filter` typu `Android.Widget.Filter`. *Getter* vlastnosti `Filter` vrací novou instanci implementované třídy `MyFilter`, která dědí ze třídy `Filter`.

Třída `MyFilter` přepisuje metody `PerformFiltering` a `PublishResults`. V konstruktoru dostává referenci na adaptér, který ji vytvořil.

Metoda `PerformFiltering` nejdříve načítá všechna zvířata pomocí metody `GetFilteredAnimals` (popsána výše). Následně v názvech zvířat hledá, zda obsahují hledaný řetězec z parametru metody. Název zvířete i hledaný řetězec je před hledáním zbaven diakritiky a převeden na malá písmena, což umožňuje hledání, i když hledaný řetězec nebude obsahovat diakritiku a název zvířete ji obsahuje (např. pokud uživatel zadá slovo „zirafa“, aplikace vyhodnotí správně i název Žirafa). Nalezené výsledky jsou následně převedené na pole objektů typu `Java.Lang.Object` a vloženy do vlastnosti `Values` vracené proměnné, která je typu `FilterResults`. Také se nastaví její vlastnost `Count` na počet nalezených výsledků.

Metoda `PublishResults` dostává v parametru výsledky filtrování z metody `PerformFiltering`. Ty se převádí zpět na `.NET` objekty a vkládají se zpět do kolekce `animalsBasics` adaptéru. Pokud jsou výsledky prázdné, prvky v kolekci `animalBasics` se odstraní. Po provedení změn v `animalBasics` se na adaptér zavolá metoda `NotifyDataSetChanged`. Ta je v používaném adaptéru `AnimalListViewAdapter` přepsána tak, aby nejdříve zavolala metodu `createIndex`, která pro novou kolekci vytvoří nové indexy sekcí pro rychlé posouvání, a následně volá metodu `NotifyDataSetChanged` své nadtřídy. Tato metoda způsobí obnovení `ListView` a načtení aktuálních vyfiltrovaných dat.

#### 3.2.4.2 AnimalDetailActivity

Tato aktivita zobrazuje detailní informace o vybraném zvířeti. Detaily jsou rozděleny do tří stránek mezi kterými lze přepínat přetažením obrazovky do strany. První stránka obsahuje základní informace, druhá zajímavosti a detailní popis a třetí informace o chovu a umístění daného zvířete v Zoo Praha (více popsáno dále).

V příloze D je na obrázku D.8 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

Jednotlivé stránky se zobrazují na `AnimalDetailTabsFragment`, který je z hlavní aktivity vytvářen pomocí metody `NewInstance`, přes kterou mu je předáno id zvířete, které se má zobrazit.

`AnimalDetailTabsFragment` obsahuje `ViewPager`, který se stará o zobrazení jednotlivých stránek. Jako adaptér pro dodání těchto stránek je implementován `AnimalDetailPagerAdapter`.

`AnimalDetailPagerAdapter` dědí z třídy `FragmentStatePagerAdapter` a přepisuje metody `GetPageTitleFormatted`, která vrací název stránky, `Count`,

která vrací počet zobrazovaných stránek, tedy 3, a `GetItem`, která vrací danou stránku, resp. `Fragment`, který stránku obsahuje.

Metoda `GetItem` se podle parametru `position` rozhoduje který fragment má vrátit. Pokud je `position` rovno 0, pak vrací `AnimalBasicInfoFragment`, pokud je `position` rovno 1, pak vrací `AnimalDescFragment` a pokud je rovno 2, pak vrací `AnimalZooFragment`. Pokud fragment ještě neexistuje, vytvoří ho pomocí jeho metody `NewInstance`, které předává id zvířete, a následně jej také uloží do proměnné třídy. Pokud již byl vytvořen, tj. je přiřazen v k tomu určené proměnné, pak se znovu nevytváří a vrací se již dříve vytvořený. To nastává pokud se uživatel vrací na stránku na které již byl. V tom případě se nevytváří nový fragment, ale použije se již dříve vytvořený.

`AnimalDAnimalBasicInfoFragment` obsahuje první zobrazenou stránku se základními informacemi o vybraném zvířeti. Fragment používá `AnimalLayout`, který obsahuje jedno `ImageView` pro zobrazení obrázku zvířete a poté několik `TextView` pro zobrazení základních informací.

Stránka se vytváří v metodě fragmentu `OnCreateView`. Zde se vytvoří nové `ScrollView`, které zajišťuje možnost posouvání obrazovky, do kterého se vloží výše zmíněný `AnimalLayout`. V této metodě se z databáze pomocí třídy `RealmDBHandler` a její metody `GetAnimalById` načtou informace o zvířeti identifikovaném pomocí id, které bylo fragmentu předáno při vytvoření. Z těchto dat se naplní jednotlivá `TextView`. Na této stránce se zobrazuje název zvířete (`Title`), latinský název (`LatinTitle`), taxonomická třída (`Class`), taxonomický řád (`Order`), výskyt (`SpreadNote`), biotop (`BiotopesNotes`), potrava (`FoodNotes`) a proporce zvířete (`Proportions`). Do `ImageView` se pomocí knihovny *Picasso* (viz 3.2.9.2) načte obrázek zvířete. Obrázek se stahuje přes API ze serveru, kde je uložena zmenšená verze původního obrázku. To zmenšuje objem stahovaných dat a výrazně šetří uživatelská data. Nakonec metoda vrací vytvořené `ScrollView` s vložených `AnimalLayout` naplněným daty.

Další stránka je implementována pomocí `AnimalDescFragment`. Stránka obsahuje popis a zajímavosti o zvířeti. Fragment používá `AnimalTextLayout`, který obsahuje několik `CardView`, kde každé obsahuje vertikální `LinearLayout` se dvěma `TextView`, jedno s názvem kategorie popisu a druhé pro konkrétní popis.

V metodě `OnCreateView` se vytvoří `ScrollView` pro možnost posouvání, do které se vloží `AnimalTextLayout`. Z databáze se načtou data pro dané zvíře pomocí třídy `RealmDBHandler` a její metody `GetAnimalById`. Do jednotlivých `TextView` se vloží příslušná data, a to popis zvířete (`Description`), zajímavost (`Attractions`), informace o rozmnožování (`Reproduction`) a případně, pokud taková informace existuje, informace o ohrožení druhu daného zvířete (`ProjectsNote`). Nakonec metoda vrací vytvořené `ScrollView` s vloženým `AnimalTextLayout` naplněným daty.

Poslední stránka je implementována pomocí `AnimalZooFragment`. Ta obsahuje informace o chovu zvířete v Zoo Praha, pavilon či výběh ve kterém

v Zoo Praha žije, případný odkaz na tento pavilon na stránky Zoo Praha a tlačítko pro zobrazení daného pavilonu na mapě. Jako layout je znovu použit `AnimalTextLayout`.

V metodě `OnCreateView` je opět použito jako hlavní view `ScrollView`. Do něho se vloží výše zmíněný `AnimalTextLayout`. Na této stránce je v použitém layoutu potřeba změnit i nadpisy jednotlivých kategorií. Kategorie jsou na této stránce tři: „Chov v Zoo Praha“, „Umístění“ a „Ukázat na mapě“. Čtvrtá `CardView` v použitém layoutu se nepoužije a její viditelnost se nastaví na `Invisible` (neviditelný).

Data se načítají jako v předchozích případech, tj. třídou `RealmDBHandler` a její metodou `GetAnimalById`. Do prvního `CardView` se vloží informace o chovu v Zoo Praha (`Breeding`), do druhé název umístění v zoo (`LocalitiesTitle`) a odkaz na dané umístění (`LocalitiesUrl`), který se pomocí třídy `Linkify` převede na hypertextový odkaz, a do třetí se vloží tlačítko s nápisem „Ukázat na mapě“. Tomuto tlačítku je na akci `Click` přiřazena metoda `mapBtnClick`. Ta způsobí otevření aktivity `MapActivity` (viz 3.2.7) a přiblížení mapy na dané umístění zvířete v Zoo Praha. Metoda opět vrací vytvořené `ScrollView` s vloženým `AnimalTextLayout` s vyplněnými daty.

#### 3.2.4.3 Landscape zobrazení

Pro podporu tabletů a větších obrazovek, je pro zobrazování zvířat implementováno *landscape* zobrazení (zobrazení „na šířku“). Při tomto zobrazení je levé třetině zobrazen seznam zvířat a ve zbylých dvou třetinách je zobrazen detail zvířete. Při kliknutí na položku v seznamu se v levé části načte detail.

To je umožněno za pomoci fragmentů. Celé zobrazení se nachází v aktivitě `AnimalListActivity`. Pro obrazovku, jejíž šířka je alespoň 600 dp, se automaticky použije layout ze složky `Layout-w600dp`. Ten obsahuje, stejně jako obecný layout, `AnimalListFragment` a navíc ještě `FrameLayout`, který je určen na vložení fragmentu `AnimalDetailTabsFragment`. Ten se vytváří a vkládá při každém zvolení položky ze seznamu.

### 3.2.5 Novinky

Novinky jsou implementovány dvěma aktivitami, a to `RssListActivity` a `RssDetailActivity`, které jsou popsány dále.

#### 3.2.5.1 RssListActivity

Aktivita je tvořena jedním fragmentem – `RssListFragment`. Ten dědí od třídy `ListFragment`, která je jako klasický fragment, pouze již obsahuje `ListView` objekt.

V příloze D je na obrázku D.3 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

Po kliknutí, respektive doteku, na jednu novinku se otevře detail této novinky.

O vyplnění `ListView` jednotlivými `View` s daty z databáze se stará adaptér `RssListViewAdapter`. Ten dědí ze třídy `BaseAdapter`. V jeho konstruktoru se z databáze načtou všechny novinky pomocí třídy `RealmDBHandler` a její metody `GetAllRss`. Adaptér obsahuje dále popsané metody.

Metoda `GetItem` vrací název novinky vyskytující se na parametrem dané pozici.

Metoda `GetItemId` vrací stejnou pozici, jaká je dána parametrem. Id v databázi jsou brány ze serverové MongoDB, jsou tedy typu `String` a nelze je v této metodě vracet.

Vlastnost `Count`, která má pouze getter, ve kterém vrací celkový počet prvků, tedy celkový počet novinek.

Metoda `GetView` se stará o vytváření a případnou recyklaci `View` pro `ListView`. Díky této metodě se nemusí při posouvání `ListView` při každém posunutí vytvářet nové `View`, ale je použito staré, ve kterém se jen změní text, což zásadně šetří využívanou paměť a její správu.

V metodě se z parametru `convertView` vezme vlastnost `Tag`, přetypuje na `RssListViewAdapterViewHolder` a přiřadí do proměnné `holder`. Třída `RssListViewAdapterViewHolder` slouží pro uchování zobrazovaných `View`, konkrétně dvou `TextView`, pro název a pro datum.

Pokud má `holder` hodnotu `null`, pak se musí vytvořit nový `holder`. Do vraceného `view` se přes `inflater.Inflate` vloží `View` typu `SimpleListItem2`. Toto je jedno z předdefinovaných `View`, které obsahuje právě dvě `TextView`. Tyto `TextView` se vloží do proměnné `holder` a jako text se jim vloží název a datum z dat odpovídajících parametrem dané pozici. `Holder` se následně uloží do `view.Tag`.

Pokud `holder` není `null`, pak se do jeho jednotlivých `View` přiřadí texty z dat odpovídajících parametrem dané pozici a metoda vrací `convertView` z parametru.

#### 3.2.5.2 RssDetailActivity

Tato aktivita slouží pro zobrazení detailu novinky. Je tvořena fragmentem `RssDetailFragment`. Ten používá layout `RssDetail`, který obsahuje čtyři `TextView` – pro název, datum, popis a případný odkaz.

V příloze D je na obrázku D.4 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

Fragment se vytváří pomocí statické metody `NewInstance`, která jako parametr přijímá ID položky, která se má zobrazit. Ta vytvoří nový fragment a do argumentů mu vloží ID z parametru, následně vytvořený fragment vrací.

Nový fragment pak v metodě `OnCreateView` načte z databáze položku novinky podle id v argumentech. Vlastnosti z načtené novinky se pak vloží do příslušných `TextView`.

#### 3.2.5.3 Landscape zobrazení

Pro podporu tabletů a větších obrazovek, je i pro sekci novinek implementováno landscape zobrazení. Při tomto zobrazení je v levé třetině zobrazen seznam novinek a ve zbylých dvou třetinách je zobrazen detail novinky. Při kliknutí na položku v seznamu se v levé části načte detail.

V příloze D je na obrázku D.5 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se tohoto zobrazení.

Toto zobrazení je umožněno za pomoci fragmentů. Celé zobrazení je v aktivitě `RssListActivity`. Pro obrazovku, jejíž šířka je alespoň 600 dp, se automaticky použije layout ze složky `Layout-w600dp`. Ten obsahuje, stejně jako obecný layout, `RssListFragment` a navíc ještě `FrameLayout`, který je určen na vložení fragmentu `RssDetailFragment`. Ten se vytváří a vkládá při každém zvolení položky ze seznamu.

### 3.2.6 Akce

Akce lze rozdělit na dvě části, a to kalendář akcí a detail akce. V aplikaci jsou tyto části reprezentovány jako aktivity `ActionsCalendarActivity` a `ActionsDetailActivity`.

#### 3.2.6.1 ActionsCalendarActivity

`ActionsCalendarActivity` zobrazuje kalendář ve kterém jsou zvýrazněny dny ve kterých probíhají nějaké akce. Kalendář je implementován jako fragment `CalendarTabFragment`, který obsahuje `ViewPager`. Tažením obrazovky do stran je tedy možno listovat mezi měsíci.

V příloze D je na obrázku D.6 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

`ViewPager` používá adaptér `CalendarAdapter`, který pro každou stránku dodává fragment `CalendarFragment`, který zobrazuje aktuálně prohlížený měsíc. `CalendarAdapter` dědí ze třídy `FragmentPagerAdapter` a implementuje abstraktní metody `Count`, `GetPageTitleFormatted` a `GetItem`.

Metoda `Count` vrací počet zobrazovaných měsíců, nyní 24 (dva roky).

Metoda `GetPageTitleFormatted` vrací název aktuálně prohlíženého měsíce spolu s rokem.

Metoda `GetItem` vrací fragment s měsícem, který se má zobrazit. Všechny fragmenty jsou ukládány do pole, pokud fragment na požadované pozici neexistuje, je vytvořen, uložen do pole a vrácen. Pokud už existuje, je rovnou vracen a nevytváří se znovu. Tím se ušetří paměť i čas, které by byly nutné pro znovu vytváření fragmentů pro stejné měsíce.

Fragment s měsícem, tj. `CalendarFragment`, se vytváří přes statickou metodu `NewInstance`, které jsou jako parametry dány měsíc a rok, které je třeba zobrazit. Metoda vytvoří nový fragment a do argumentů mu vloží tyto parametry, které se využijí při vykreslení fragmentu.

Pro vykreslení fragmentu `CalendarFragment`, tj. měsíce je použit layout `CalendarLayout`, který je tvořen tabulkou se sedmi řádky a sedmi sloupci. V prvním řádku jsou ve sloupcích zkratky dnů v týdnu (PO, ÚT, ST, ČT, PÁ, SO, NE). Zbytek řádků je použit pro jednotlivé dny.

Vykreslení jednotlivých dnů je pak implementováno v privátní metodě `setDays`. Ta si z tabulky dnů načte jednotlivé řádky (kromě prvního, ve kterém jsou názvy dnů), které reprezentují týdny. Pomocí vlastnosti `DayOfWeek` třídy `DateTime` se zjistí, který je první den v požadovaném měsíci den v týdnu. Podle toho se vyplní první řádek, od zjištěného dne se začne vyplňovat čísly od 1, přes ostatní řádky až do počtu dnů v požadovaném měsíci (zjištěno pomocí metody `DaysInMonth` třídy `DateTime`). Zároveň se pro každý den zjišťuje dotazem na databázi, zda je na daný den nějaká akce. Pokud ano, pole pro daný den se zabarví na tmavě tyrkysovou barvu a přidá se mu akce pro událost kliknutí. Tímto se vykreslí celý měsíc a označí všechny dny, ve kterých má být nějaká akce.

Akce řídící událost kliknutí na daný den je `CalendarFragment_Click`. Po kliknutí na danou akci se z databáze načtou akce pro daný den a zobrazí se jejich názvy jako seznam v dialogu. Po kliknutí na vybranou akci se otevře nová aktivita s detailem dané akce – `ActionsDetailActivity`.

#### 3.2.6.2 ActionsDetailActivity

`ActionsDetailActivity` obsahuje `ActionsDetailFragment`. Tento fragment používá stejný layout jako `RssDetailFragment`, a to `RssDetail`. Z databáze se vytáhne příslušná akce a vyplní `title`, `date`, `description` a případně, pokud má akce nějaký odkaz, i link, do příslušných `TextView`.

V příloze D je na obrázku D.7 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

### 3.2.7 Mapa

Aplikace obsahuje mapu, na které jsou vyznačeny výběhy a pavilony zvířat v Zoo Praha. Mapa je implementována v aktivitě `MapActivity`.

V příloze D je na obrázku D.2 vyobrazen UML diagram tříd (class diagram) znázorňující třídy týkající se této aktivity.

`MapActivity` pro vykreslování mapy používá `MapFragment` z knihovny `Xamarin.GooglePlayServicies.Maps`. Na něj se zavolá asynchronní metoda `GetMapAsync`. Té se předá jako parametr aktuální aktivita, která proto implementuje rozhraní `IOnMapReadyCallback`. Proto také aktivita implementuje metodu `OnMapReady`, která je volána po skončení `GetMapAsync` a má jako parametr objekt mapy typu `GoogleMap`.

#### 3.2.7.1 Nastavení mapy

V metodě `OnMapReady` se nastavuje mapa a vykreslují polygony pro označení výběhů a pavilonů.

Použité nastavení mapy:

- **MyLocationEnabled** povoluje zobrazení polohy zařízení na mapě.
- **UiSettings.ZoomControlsEnabled** povoluje zobrazení ovládání přiblížení.
- **UiSettings.CompassEnabled** povoluje zobrazení kompasu.
- **UiSettings.MyLocationButtonEnabled** povoluje zobrazení tlačítka pro přesun na aktuální polohu.

Po úvodním nastavení se nastavují polygony a značky pro jednotlivé výběhy, to probíhá v metodě `setMarkersAndPolygons` (viz 3.2.7.2).

Po vykreslení následují dvě možnosti, pokud byla mapa otevřena z nějakého zvířete, provede se přiblížení k výběhu či pavilonu daného zvířete. Pokud byla mapa otevřena z menu nebo hledaný pavilon/výběh nebyl nalezen, zobrazí se pohled na celou mapu Zoo Praha.

Aktivita implementuje další rozhraní, a to `IOnPolygonClickListener`, `IOnMapClickListener` a `IOnInfoWindowClickListener`. Proto musí implementovat metody `OnPolygonClick`, `OnMapClick` a `OnInfoWindowClick`. Díky tomu může být aktivita předána mapě jako listener pro všechny tyto události.

Metoda `OnPolygonClick` řídí událost, která nastane při kliknutí (resp. dotyku) na vykreslený polygon. Při této události se najde a zobrazí příslušný marker, kamera se vycentruje a přiblíží k tomuto markeru a zobrazí se příslušné `InfoWindow` s názvem pavilonu či výběhu. Příslušný marker se najde pomocí polygon id v kolekci `polygonToMarker` (kolekce popsána v sekci 3.2.7.2).

Metoda `OnMapClick`, tj. kliknutí (dotyk) do mapy mimo vyznačené polygony, provede skrytí případného zobrazeného markeru. Jelikož není možné jednoduše zjistit, zda je nějaký marker zobrazený, provádí se skrytí iterací přes všechny markery (všechny jsou v kolekci `polygonToMarker`) a je jim nastavená hodnota vlastnosti `Visible` na `false`.

Metoda `OnOnInfoWindowClick` tj. kliknutí (dotyk) na zobrazené informační okno s názvem pavilonu či výběhu, provede otevření aktivity (tj. nové obrazovky) se seznamem zvířat chovaných v tomto výběhu/pavilonu.

#### 3.2.7.2 Vytváření polygonů a značek

Metoda `OnMapReady` volá metodu `setMarkersAndPolygons`. Ta volá statickou metodu `CreateLocations` třídy `MapLocationsCreator`. Tato metoda postupně vytvoří objekty třídy `MapLocation` pro každý výběh či pavilon a vrací kolekci `List` s těmito objekty.

Třída `MapLocation` reprezentuje výběh či pavilon v pražské zoo. Třída má vlastnosti `Title`, která obsahuje název výběhu či pavilonu, `FillColor`, která obsahuje hexadecimální kód barvy výplně (včetně průhlednosti) v mapě, `MarkerPoint`, která obsahuje souřadnice značky na mapě a `Points`, která je kolekcí souřadnicových bodů pro ohraničení hranic výběhu či pavilonu.

V samotné mapě je pavilon či výběh tvořen dvěma objekty: objektem třídy `MarkerOptions`, který reprezentuje značku v mapě, a objektem třídy `PolygonOptions`, který reprezentuje vyznačenou plochu v mapě. Tyto třídy jsou třídy z knihovny `Xamarin.GooglePlayServices.Maps`.

Třída `MapLocation` obsahuje metodu `GetMarker`, která z vlastností třídy vytvoří značku, tj. objekt třídy `MarkerOptions`, a metodu `GetPolygon`, která vytvoří polygon pro vykreslení v mapě, tj. objekt třídy `PolygonOptions`. Objektu třídy `GetMarker` se nastavuje poloha z vlastnosti `MarkerPoint` a název, který se objeví v informačním okně značky, z vlastnosti `Title`. Objektu třídy `PolygonOptions` se nastavují hraniční body z vlastnosti `Points` a barva výplně z vlastnosti `FillColor`.

Vykreslování polygonu probíhá tak, že se jednotlivé vložné body, v daném pořadí, spojují úsečkou, s tím, že poslední se automaticky spojí s prvním, čímž vznikne výsledný polygon. Ten se pak vyplní nastavenou barvou. Proto je nutné držet přesné pořadí vkládaných bodů.

O vytváření jednotlivých lokací se stará třída `MapLocationsCreator`. Ta má pro každou lokaci metodu, která vytvoří kolekci souřadnic vytvářeného polygonu, a následně vytváří přes konstruktor objekt třídy `MapLocation`, kterému předá název, kód barvy, souřadnice značky a vytvořenou kolekci souřadnic. V metodě `CreateLocations` volá všechny tyto metody, které vytvoří všechny lokace a vrací je jako kolekci `List`.

Souřadnice lokací bylo nutné ručně vyčíst z map a zadat do kódu aplikace. Byly k tomu použity Google mapy a mapa Zoo, která je k nalezení na webu Zoo Praha (viz [16]). Z mapy zoo byly odhadnuty polohy jednotlivých lokací a v Google mapách pak nalezeny souřadnice jednotlivých bodů, které byly následně zapsány do aplikace.

Nalezení souřadnic lze pomocí kliknutí do prázdně Google mapy. Mapa pak zobrazí informace o místě, včetně geografických souřadnic daného místa, na které bylo kliknuto.

Po vytvoření všech lokací pokračuje metoda `setMarkersAndPolygons` v aktivitě `MapActivity` tím, že do mapy přidá vytvořené polygony a značky. Z lokace se pomocí metod `GetMarker` a `GetPolygon` vytvoří objekty tříd `MarkerOptions` a `PolygonOptions`, které jsou předány mapě pomocí metod `AddMarker` a `AddPolygon`. Tyto metody vrací vytvořené polygony (objekt třídy `Polygon`) a značky (objekt třídy `Marker`), které jsou následně uloženy do kolekce `polygonToMarker` typu `Dictionary`, kde jako klíč je použita vlastnost `Id` polygonu a jako hodnota je vložen objekt značky. Tato kolekce je pak používána pro řízení chování značek při kliknutí na polygon nebo do prázdné mapy (tj. metody `OnPolygonClick` a `OnMapClick`).

### 3.2.8 Design

Po konzultaci s vedoucím práce byl pro grafické pojetí práce zvolen *Material design*.

Material design je vizuální jazyk, který spojuje principy dobrého designu s inovací a možnostmi technologie a vědy.[17]

Material design popisuje doporučené vzory a pravidla pro tvorbu kvalitního designu. Ty jsou popsané na webovém portálu, viz [17]. Tato práce se tyto pravidla a principy snaží dodržovat.

### 3.2.9 Použité knihovny

Tato sekce popisuje použité knihovny třetích stran.

#### 3.2.9.1 RestSharp

*RestSharp* je *opensource* knihovna obsahující jednoduchý *REST* a *HTTP* API klient pro .NET. [18]

Hlavní vlastnosti [18]:

- Jednoduchá instalace pomocí *NuGet*.
- Automatická deserializace XML a JSON formátů.
- Podpora vlastní serializace a deserializace přes rozhraní ISerializer a IDeserializer.
- Fuzzy spojování názvů elementů („produkt_id“ v XML/JSON se spojí s C# vlastností s názvem „ProductId“).
- Automatická detekce typu vráceného obsahu.
- Podpora GET, POST, PUT, PATCH, HEAD, OPTIONS, DELETE, COPY.
- Podpora dalších nestandardních HTTP metod.
- Zahrnuty druhy autentizace OAuth 1, OAuth 2, Basic, NTLM a parametrová.
- Podpora vlastních autentizačních schémat přes rozhraní IAuthenticator.
- Upload vícedílných formulářů či souborů.

Ve zkratce RestSharp umožňuje komunikaci s API pomocí standardních i nestandardních HTTP metod. Automaticky provádí serializaci i deserializaci XML a JSON formátů.

#### 3.2.9.2 Picasso

*Picasso* je knihovna pro Android pro stahování a kešování obrázků. Umožňuje bezproblémové načítání obrázků do aplikace, často pouze jedním řádkem kódu. [19]

Příklad stažení obrázku z URL adresy „http://i.imgur.com/DvpvklR.png“ do widgetu `imageView` typu `ImageView` [19]:

```
Picasso.with(context).load("http://i.imgur.com/DvpvklR.png")
.into(imageView);
```

Mimo jiné poskytuje automatické paměťové i diskové kešování (dočasné uložení pro rychlé znovu načtení). Dále umožňuje transformaci obrázků s využitím minimálního množství paměti. Také automaticky detekuje znovupoužití (re-use) `ImageView` v adaptéru a automaticky zruší předešlé stahování.[19]

Knihovna je implementována pro Javu, ale existují k ní tzv. *bindings* (provázání, „zabalení“ Java knihovny do .NET), které umožňují použití pro Xamarin a C#. Ty jsou dostupné jak jako *NuGet* balíček, (i jako *Xamarin Component*, které však byly sloučeny s NuGet, viz 6.9.3). Knihovna i bindings jsou opensource projekty. Více o knihovně Picasso na [19]. Více o bindings knihovny Picasso pro Xamarin na [20].

Kapitola 4

# Dokumentace a testování

Tato kapitola obsahuje popis dokumentace a testování praktické části této práce.

## 4.1 Dokumentace

Tato sekce popisuje způsob dokumentace této práce, rozdělený na programátorskou a uživatelskou část.

### 4.1.1 Programátorská dokumentace

Hlavní část programátorské dokumentace tvoří text této práce, hlavně kapitola 3. Zde je popsána jak veškerá funkcionalita, tak rozbor vlastní implementace jednotlivých částí práce.

Součástí této dokumentace jsou i UML diagramy tříd v příloze D, ve kterých je názorně vidět struktura obou částí aplikace. Diagramy byly vytvořeny v programu *Umlet* (viz [21]), zdrojové soubory jsou dostupně na přiloženém CD.

Samotný kód byl psát tak, aby byl dostatečně sebe dokumentující. Názvy metod a proměnných byly voleny aby byl co nejvíce jasný jejich účel. Metody jsou dostatečně krátké a plní pouze své funkce a neobsahují nesouvisející kód.

I přesto je kód dokumentován. Dokumentace je tvořena XML dokumentačními komentáři (viz [22]), které popisují třídy a metody, pokud nemusí být úplně jasné co přesně dělají. Tyto komentáře jsou značeny třemi lomítky (`///`) a používají XML tagy.

Příklad dokumentace:

```
/// <summary>
/// Find animal by given id and return it.
/// </summary>
/// <param name="id">Id of animal for search.</param>
```

```
/// <returns>Found animal object.</returns>
Animal GetAnimalById(int id);
```

V příkladu výše je vidět jak vypadá XML dokumentace metody GetAnimalById. V XML tagu *summary* je popis dané metody, v tagu *param* je jméno parametru a jeho popis a v tagu *returns* je popis návratové hodnoty.

Z této dokumentace je možné vygenerovat XML soubory s touto dokumentací. Ty pak lze pomocí různých nástrojů analyzovat a prezentovat. Vzhledem k již dostatečné dokumentaci a jednoduché architektuře aplikace není v této práci žádný takový nástroj použit.

Dále jsou v kódu v některých metodách přítomny komentáře k některým řádkům těchto metod pro větší přiblížení jejich chování.

Komentáře v kódu byly psány, aby dodržovaly zásady a rady z knihy *Code complete*[23], tj. aby popisovaly účel či záměr, než aby znovu popisovaly příkazy kódu.

### 4.1.2 Uživatelská dokumentace

Uživatelská dokumentace k samotné mobilní aplikaci by neměla být potřeba. Mobilní aplikace by samy o sobě měly být dostatečně intuitivní, což je dáno i dodržením vzorů pro Material Design (viz [24]). Potřeba uživatelské dokumentace k mobilní aplikaci značí pravděpodobně špatný návrh takové aplikace.

Vzhledem k uživatelské jednoduchosti vytvořené aplikace, které obsahuje několik obrazovek a přechody mezi nimi jsou jasně definovány buď tlačítkem, případně kliknutím na zvíře, a šipkou zpět, není uživatelská dokumentace potřeba.

Uživatelská dokumentace k serverové části je v podstatě dokumentace k vytvořenému API. Ta je vytvořená pomocí Apiary a popsána v kapitole 2.3.3. Navíc je dokumentace API vytvořena i v aplikaci *Postman* (více v sekci 4.2.1) a je dostupná z [25].

## 4.2 Testování serveru

Testování serverové části této práce je realizováno pomocí nástroje *Postman*.

### 4.2.1 Postman

*Postman* je nástroj pro testování API, který mimo jiné umožňuje v přehledném uživatelském prostředí testovat API, vytvářet kolekce dotazů a implementaci automatických testů. Více o tomto nástroji v [26].

Pro testování vytvořeného API je v této aplikaci vytvořena kolekce Lexicon. Její export lze najít na přiloženém CD. Tento export lze importovat do aplikace a používat.

Testy testují volání metody GET na API `api/animals`, `api/actions` a `api/rss`. Každé z těchto volání testuje, za prvé, zda proběhlo v pořádku, a za druhé, zda data odpovídají zdrojovým datům.

Každý test stáhne data z vytvořeného serveru a zároveň z původního zdroje. Z dat ze serveru jsou následně odstraněna data s hodnotou vlastnosti `deleted` nastavenou na `true`. Poté test vypisuje množství prvků v obou kolekcích. Následně se prochází všechny záznamy v serverové kolekci a hledá se, zda se zde vyskytují jiné, než v původních datech, které následně vypíše.

Automatická kontrola, zda jsou data stejná nelze udělat z důvodu zpoždění stahování dat na implementovaném serveru. Místo toho se vypisují počty záznamů v obou kolekcích a data, která jsou na serveru navíc. To je pak možné zkontrolovat ručně. Testy neprojdou pouze v případě, že kolekce na serveru nemají žádná data.

Kolekce v Postmanu obsahuje i volání metod GET na `api/animals/images/1`, kde testuje pouze, zda volání proběhlo v pořádku, tj. byl vrácen status kód 200.

Postman také umožňuje vytváření pravidelného automatického monitorování kolekce. To spouští v pravidelných intervalech danou kolekci. Interval spouštění lze nastavit od jednou za pět minut až do jednou za týden. Pro rychlé upozornění lze nastavit odesílání informací o nepodařených testech a chybách. Bezplatně lze využívat až 1000 volání za měsíc, pro více je již nutné zakoupit prémiové služby [27].

Pro monitorování stavu serverové části této práce je vytvořen monitor s intervalem opakování každý den o půlnoci.

Postman také umožňuje generování veřejné dokumentace API z vytvořené kolekce. Ta je použita i v této práci a je dostupná z [25].

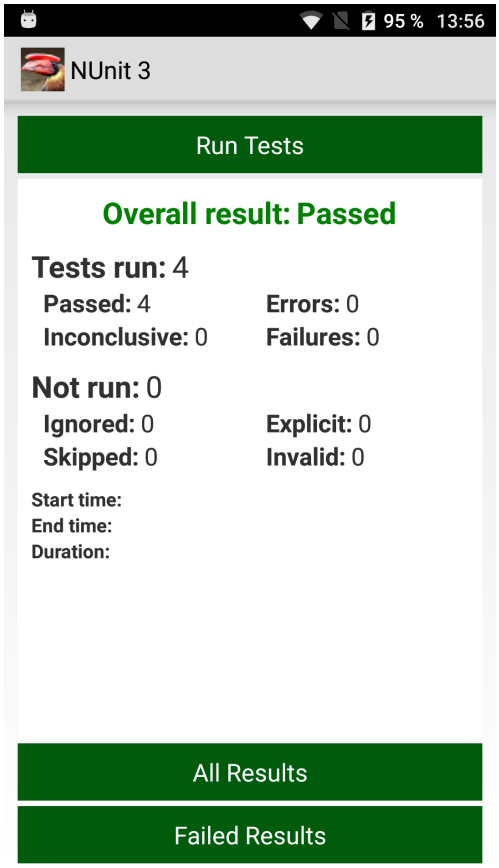
## 4.3 Unit testy mobilní aplikace

Unit testy jsou vytvořeny pomocí *Nunit test runner pro Xamarin*[28].

Testy jsou implementovány jako další část aplikace, která je dostupná, pouze při kompilaci s *DEBUG* konfigurací. V menu se objeví navíc jedno tlačítko, které otevírá aktivitu `TestActivity`. Tato aktivita nastavuje, načítá vytvořené testy a spouští `Nunit.Runner.App`. V té je pak možné pouštět testy a kontrolovat jejich výsledky. Na obrázcích 4.1 a 4.2 je zobrazeno rozhraní těchto testů.

Implementování testů přímo do aplikace bylo zvoleno z důvodu testování databáze, ke které by jinak nebyl možný přístup.

Testy jsou implementovány v souboru `UnitTests.cs` a testují shodu dat v databázi a daty staženými pomocí `RestService`. Dále testují, zda se všechny body, které se zapisují do mapy, nacházejí na území Zoo Praha.




Obrázek 4.1: NUnit runner – hlavní obrazovka

## 4.4 UI testy mobilní aplikace

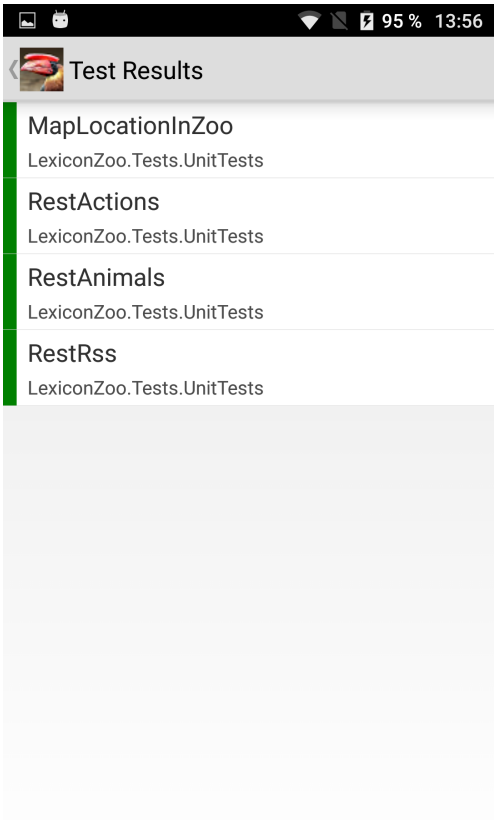
Pro mobilní aplikaci jsou implementovány tzv. UI testy, tedy testy uživatelského rozhraní. Ty simulují akce uživatele a kontrolují chování aplikace.

Pro implementaci UI testů existuje několik frameworků. Protože je pro implementaci mobilní aplikace k této práci použita technologie Xamarin, je pro testování použit *Xamarin.UITest* framework.

### 4.4.1 Xamarin.UITest

*Xamarin.UITest* je automatický akceptační framework pro testování uživatelského rozhraní, založený na frameworku *Calabash*, který umožňuje psaní a spouštění testů v jazyku C# a s pomocí testovacího frameworku *NUnit*, který validuje funkcionalitu Android a iOS aplikací. [29]

Každý test je metoda označená atributem `Test`. Třída, která obsahuje testy, se označuje atributem `TestFixture`. [29]



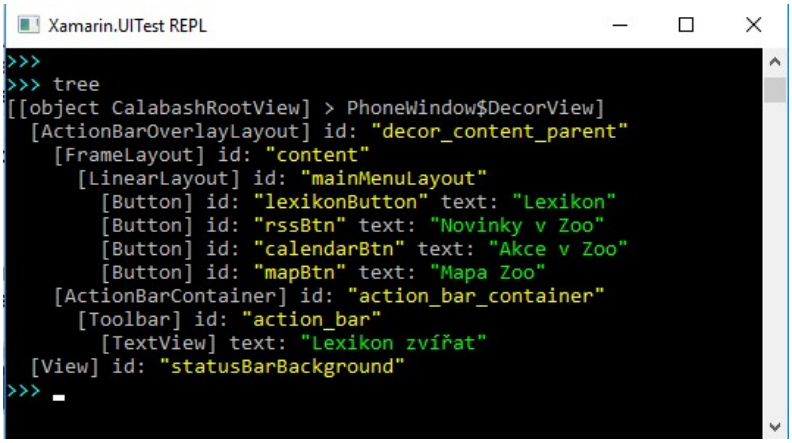

Obrázek 4.2: NUnit runner – výsledky testů

Testy se vytváří v novém projektu, ideálně ale ve stejném Solution jako je testovaný projekt.

Pro vytváření a spouštění testů je nutné mít v projektu referenci na poslední verzi Xamarin.UITest NuGet balíčku. Dále je třeba mít balíček s NUnit frameworkem ve verzi 2.6.x. Xamarin.UITest není kompatibilní s novějšími verzemi 3.x. Pro spouštění je dále nutné mít *NUnit Test Adapter* pro NUnit 2. A pro testování Android aplikací, což je případ této práce, je nutné mít nainstalované *Android SDK* a *Java Developers Kit*. [29]

Interakce testu s mobilní aplikací probíhá pomocí `Xamarin.UITest.IApp`. Toto rozhraní definuje metody zásadní pro spolupráci testu s aplikací a interakci s uživatelským rozhraním. Momentálně existují dvě konkrétní implementace, a to `Xamarin.UITest.iOS.iOSApp` pro testování iOS aplikací a `Xamarin.UITest.Android.AndroidApp` pro testování Android aplikací. Tyto třídy nejsou vytvářeny přímo, ale přes třídu `ConfigureApp`, která zajišťuje že bude `iOSApp` or `AndroidApp` správně vytvořeno. [29]

Tento přístup zajišťuje, že je možné vytvořit jednu skupinu testů, která




Obrázek 4.3: UI test – REPL – příkaz `tree` v úvodním menu

pak lze spustit jak na Android, tak na iOS zařízeních.

Instanci `IApp` je dobré před každým testem vytvořit nově. Tím se vytvoří nová instance testované aplikace a zamezí se přenesení stavu jednoho testu do druhého a testy budou na sobě nezávislé. Pro toto nové vytvoření je vhodné použít metodu s atributem *SetUp*, která se spouští před každým testem. [29]

Pro vytváření testů je vhodné a ve většině případů nezbytné použít *REPL* (Read–eval–print loop), který Xamarin.UITest poskytuje. REPL je interaktivní konzolová aplikace, která umožňuje prozkoumávat hierarchii obrazovky, získávat informace o obsažených prvcích, zkoušet příkazy a dotazy, které umožňují interakci s aplikací. Ty pak lze využít při vlastním psaní testů. REPL lze spustit pouze z vytvořeného testu voláním metody `IApp.Repl()`. [29]

Zásadním příkazem REPLu je `tree`. Ten vypisuje hierarchii aktuální obrazovky (příklad hierarchie úvodního menu viz obrázek 4.3). K jednotlivým prvkům pak lze přistupovat pomocí třídy `AppQuery`. Nejužitečnější metody této třídy jsou `Class`, která vrací `View` specifikované třídy, `Id`, která vrací `View` s daným id, `Text`, která vrací `View` s daným textem a metoda `Marked`, která hledá `View` podle hodnoty ve vlastnosti `Id`, `Text` nebo `ContentDescription`. [29]

Pro samotnou interakci s aplikací slouží metody instance rozhraní `IApp`. Všechny tyto metody přijímají v parametru buď funkci `AppQuery`, která vrací daný prvek, nebo řetězek, který se použije k nalezení prvku metodou `Marked`. Zásadní jsou metody `Tap`, která simuluje dotyk na daný prvek, `EnterText`, která vloží zadaný text do daného prvku, `WaitForElement`, která čeká, až se na obrazovce objeví daný element, `WaitForNoElement`, která čeká, až daný element z obrazovky zmizí, `Screenshot`, která vytvoří a uloží obrázek aktuální obrazovky a další. [29]

### 4.4.2 Implementace UI testů

Součástí této práce je projekt *UITest*, který je součástí hlavního Solution LexiconZoo. Tento projekt obsahuje třídu `Tests`, která má atribut `TestFixture` a obsahuje testy pro testování uživatelského rozhraní.

V metodě `BeforeEachTest`, která je označena atributem `SetUp`, a je tedy spouštěna před každým testem se vytváří nová instance `IApp`, konkrétně `AndroidApp`. Dále se zde volá metoda `DownloadData`. Ta v aplikaci potvrdí stažení dat a počká než se data stáhnou.

Práce obsahuje šest testů pro otestování všech obrazovek a zásadních funkcí.

Test `AnimalBasicTest` testuje základní funkcionalitu části lexikonu zvířat. Testování probíhá podle tohoto testovacího scénáře:

1. Otevření lexikonu zvířat v hlavním menu.
2. Otevření abecedního zobrazení.
3. Ověření, že první zvíře je Achatina žravá. Test se tím stává závislý na datech, ale díky tomu lze odhalit změnu zdrojových dat nebo případnou chybu při stahování dat.
4. Dotyk na první zvíře v seznamu.
5. Kontrola, že se název zvířete rovná názvu ze zvířete seznamu.
6. Dvakrát posun obrazovky doleva. Tím se obraz přesune na obrazovku *V Zoo Praha*.
7. Dotyk na tlačítko *Ukázat na mapě*.
8. Kontrola, že se otevřela obrazovka mapy.
9. Posun zpět na obrazovku se seznamem.
10. Dotyk na tlačítko pro změnu zobrazení.
11. Kontrola, že počet zobrazených zvířat v novém zobrazení (malé položky v seznamu) je větší než v předchozím zobrazení (velké karty).

Test `AnimalSearchTest` testuje funkci vyhledávání v lexikonu zvířat. Testování probíhá podle tohoto testovacího scénáře:

1. Otevření lexikonu zvířat v hlavním menu.
2. Otevření abecedního zobrazení.
3. Dotyk na ikonu hledání (lupa).

4. Vyplnění textu „zirafa“ do vyhledávacího pole.

5. Kontrola, že název prvního zvířete obsahuje slovo „Žirafa“.

Test `AnimalTaxonomyListTest` testuje zobrazení seznamu zvířat podle taxonomických tříd. Testování probíhá podle tohoto testovacího scénáře:

1. Otevření lexikonu zvířat v hlavním menu.

2. Otevření taxonomického zobrazení.

3. Dotyk na první položku v seznamu.

4. Dotyk na první zvíře v seznamu.

5. Kontrola, že třída zvířete odpovídá třídě, která byla otevřena v bodu 3.

Test `AnimalLocalityListTest` testuje zobrazení seznamu zvířat podle umístění v Zoo. Testování probíhá podle tohoto testovacího scénáře:

1. Otevření lexikonu zvířat v hlavním menu.

2. Otevření zobrazení podle umístění.

3. Dotyk na první položku v seznamu.

4. Dotyk na první zvíře v seznamu.

5. Posun obrazovky detailu zvířete na obrazovku *V Zoo Praha*.

6. Kontrola, že umístění zvířete odpovídá umístění, které bylo otevřena v bodu 3.

Test `RssTest` testuje funkcionalitu části novinek. Testování probíhá podle tohoto testovacího scénáře:

1. Otevření části novinek v hlavním menu.

2. Dotyk na první položku v seznamu.

3. Kontrola, že název a datum v detailu novinky je stejné jako název a datum položky, která byla otevřena v předchozím kroku.

Test `ActionsTest` testuje funkcionalitu části kalendáře akcí. Testování probíhá podle tohoto testovacího scénáře:

1. Otevření části kalendáře akcí v hlavním menu.

2. Dotyk na první položku v seznamu.

3. Dotyk na dnešní datum.
4. Kontrola zda se otevřel dialog. Pokud ne, zkusit kliknout na další datum.
5. Opakovat předchozí bod, dokud se neotevře dialog. Pokud se nepodaří nic otevřít tento měsíc, přesunout se na další měsíc. Pokud nevyjde ani další měsíc, test neprošel.
6. Po otevření dialogu s výběrem akcí na dané datum otevřít první položku v dialogu.
7. Kontrola, zda název akce v otevřeném detailu je shodný s názvem akce v dialogu z předchozího kroku.

V tomto testu je část s klikáním na všechna data v kalendáři proto, že z programu není poznat, na které je plánovaná nějaká akce.

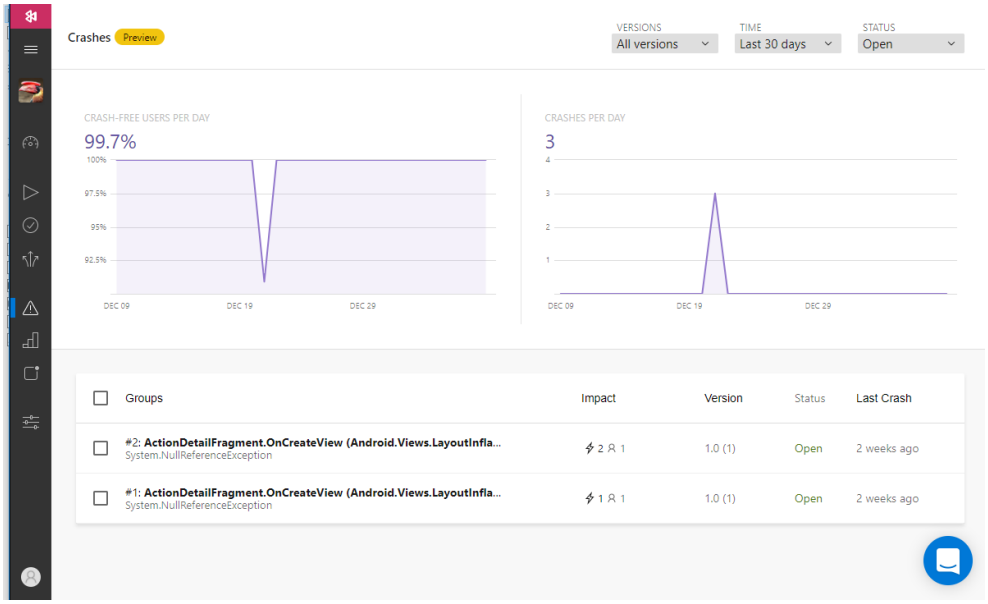
Tyto testy pokrývají všechny obrazovky a jejich možnosti, kromě rychlého posouvání a mapy zoo. Rychlé posouvání nejde z testu nasimulovat a k prvkům v mapě zoo nelze přistupovat, proto jsou tyto funkce z testů vynechány a je potřeba je případně testovat ručně.

## 4.5 Sledování stavu nasazené aplikace

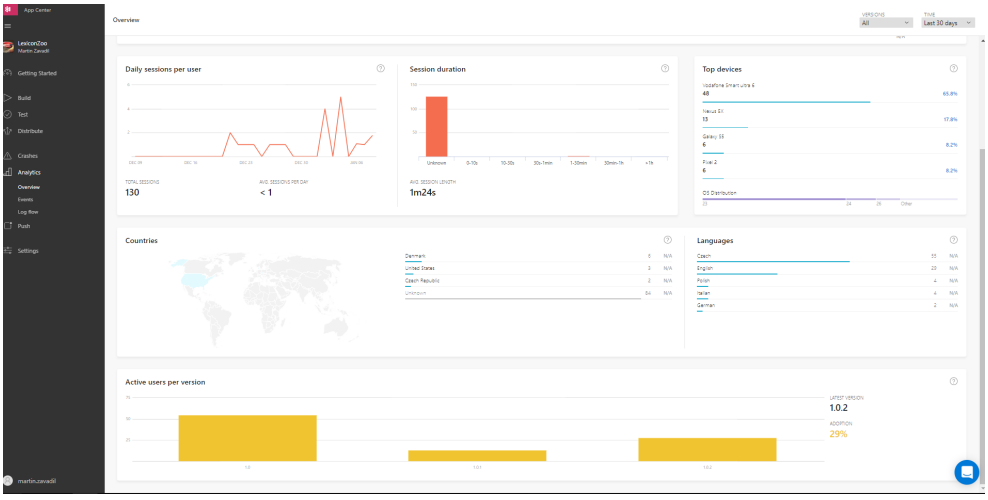
Vzhledem k velkém počtu různorodých zařízení s operačním systémem Android, na kterých může aplikace běžet, není možné mít aplikaci otestovanou pro všechna zařízení. Řešením této situace je sledování stavu a pádů již nasazené aplikace. Pro to existuje několik nástrojů. Jelikož tato práce je založena Microsoft technologiích, je i pro sledování stavu a pádů aplikace použit jejich nástroj, a tím je *Microsoft App Center* (viz [30]).

Pro použití reportování Microsoft App Center stačí přidat do aplikace NuGet balíčky *Microsoft.AppCenter.Analytics* a *Microsoft.AppCenter.Crashes* a následně do hlavní aktivity přidat volání metody `AppCenter.Start` s vygenerovaným id. Po tomto nastavení již lze v App Center sledovat jak pády aplikace, tak analytická data o zařízeních, které používají aplikaci. Na pády aplikace reportované touto cestou je možné průběžně reagovat a aplikaci opravit či upravit pro různá zařízení.

Na obrázcích 4.5 a 4.4 jsou obrázky obrazovek těchto nástrojů.




Obrázek 4.4: MS App center Crashes




Obrázek 4.5: MS App center Analytics

Kapitola 5

# Nasazení

Tato kapitola popisuje způsob nasazení implementovaného serveru na cloudovou službu *Heroku* (sekce 5.1) a nasazení mobilní aplikace do obchodu *Google Play* (sekce 5.2). Dále popisuje použité prostředí pro průběžnou integraci s využitím služby *Microsoft App Center* (sekce 5.3).

## 5.1 Nasazení serveru (Heroku)

Serverová část této práce je nasazena na cloudové službě (služba běžící na internetu, k jejímu využívání není potřeba žádný hardware) *Heroku* (viz [31]). Server běží na URL adrese *https://agile-chamber-32011.herokuapp.com*.

Heroku umožňuje nasazení vlastní aplikace v různých technologiích, mimo jiné i v této práci používaném Node.js. Heroku nabízí různé možnosti financování pro různé potřeby. Pro tuto práci je použita varianta, která je zdarma. Ta má jistá omezení, většina z nich pro tuto práci není podstatná.

Jediné omezení, které se této práce dotýká je uspávání serveru po 30-ti minutách bez aktivity. Takovéto uspání serveru způsobuje, že při dotazu se musí server nejdříve spustit, což prodlužuje čas odezvy. To je nejvíce vidět po instalaci mobilní aplikace, kdy první stažení trvá poměrně dlouho (až 2 minuty). To je způsobeno právě startováním serveru po uspání.

Serverová databáze MongoDB je nasazena v cloudové službě *mLab* (viz [32]). Služba mLab poskytuje je poskytována jako *Database-as-a-Service* (databáze jako služba) pro MongoDB.

Heroku poskytuje doplněk (add-on) pro propojení Heroku projektu a mLab databáze, což zjednodušuje správu.

## 5.2 První nasazení mobilní aplikace (Google Play)

Mobilní aplikace je nasazená v obchodě *Google play* (viz [33]). Pro vydání aplikace v obchodě Google play je nutné založit vývojářský účet, což mimo

jiné zahrnuje jednorázový poplatek 25 $.

První vydání aplikace musí proběhnout přes Google play vývojářskou konzoli.

Aplikace se vydává jako soubor ve formátu *APK*. Před vydáním aplikace je nutné aplikaci podepsat digitálním certifikátem. Každá nová verze aplikace pak musí být podepsána stejným certifikátem. To zvyšuje bezpečnost a snižuje pravděpodobnost podvržení aplikace. Pokud by se někdo nepovolaný dostal k přístupovým údajům k účtu v obchodě Google play, nebude bez daného certifikátu schopen nahrát novou verzi aplikace. [34]

Vytvoření a podepsání souboru APK pro vydání ve Visual Studiu 2015 probíhá dále popsaným způsobem. Nejdříve je potřeba z projektu vytvořit archiv. To lze přes kontextovou nabídku projektu volbou varianty **Archive...**. Tím se spustí archivace projektu a otevře se okno **Archive manager**. Zde jsou k dispozici vytvořené archivy. Po zvolení vytvořeného archivu se pomocí tlačítka **Distribute...** otevře dialogové okno. Zde je možné zvolit, zda se má vytvořit APK soubor tzv. *ad hoc* nebo rovnou distribuovat na Google play. Jak bylo zmíněno výše, první vydání musí být přes Google play vývojářskou konzoli, proto je nutné zvolit možnost ad hoc. Dále už je nutné jen zvolit příslušný certifikát, případně vygenerovat nový, a nový soubor APK uložit.

Takto vytvořený a podepsaný APK soubor se vloží do nového projektu v Google play vývojářské konzoli. Tam je také nutné vyplnit další údaje a vložit obrázky obrazovek aplikace.

Také je možné zvolit, jestli má být aplikace vydána jako *aplha* verze, *beta* verze a nebo ostrá verze. V případě aplha a beta verze je možné zvolit, zda má být otevřená (aplikaci mohou stahovat všichni uživatelé) nebo uzavřená (jsou vybráni a pozváni určití uživatelé pro testování aplikace).

Po vyplnění všech potřebných údajů a potvrzení vydání je aplikace publikována v obchodě Google play a je možné ji stahovat a používat.

Aplikace k této práci je momentálně vydána jako otevřená beta verze. Aplikace je nazvána „Lexikon zvířat Zoo Praha“ a lze ji stáhnout z [35].

## 5.3 Continuous integration (MS App Center)

*Microsoft App center*, které je již použito pro sledování pádů aplikace (viz 4.5), nabízí i další nástroje pro podporu vývoje.

Jedna z možností je propojení s gitovými repositáři, konkrétně *GitHub*[36], *BitBucket*[37] a *Visual Studio Team Services*[38]. Po připojení repositáře je možné nastavit automatické sestavení (Build) aplikace po každém novém přispění do připojeného repositáře. Po sestavení lze nastavit, zda se má provádět test spuštění aplikace na reálném zařízení. Pokud je test nastavený, aplikace se spustí na reálném zařízení a v Microsoft App center se zobrazí, zda spuštění proběhlo úspěšně, včetně nahraného snímku obrazovky. Také je možné vložit

vlastní skript pro sestavení (Build script), který může např. spouštět Unit testy.

K sestavení lze přidat vlastní certifikát aplikace, kterým se následně aplikace podepíše.

Další funkce, kterou Microsoft App center nabízí, je možnost propojení s obchodem Google play. To umožňuje nasazení aplikace přímo z Microsoft App center. Pro propojení je nejdříve nutné vytvořit přístupové API v *Google developer consoli* a nastavit ho jako služební účet v *Google play consoli*. K vygenerovanému API lze stáhnout privátní klíč ve formátu JSON, který se následně vloží do Microsoft App center. Celý návod je k nalezení na [39].

Po propojení k obchodu Google play je možné publikovat aplikaci přímo z Microsoft App center. Podmínkou je, že aplikace již byla nasazena přímo v Google play. Aplikaci lze publikovat jak jako vlastní APK soubor, tak také jako výsledek sestavení z propojeného repositáře. To je podmíněné vložením správného certifikátu pro podpis aplikace při sestavení.

Tato práce využívá jako repositář Bitbucket, který je připojen na Microsoft App center. Je zde nastavené průběžné sestavování po každém přispění do repositáře. Je nastavený test po spuštění a vložen certifikát pro podepsání. Také byla z Microsoft App center publikována nová verze aplikace do obchodu Google play.

Microsoft App center je v této práci použito jako prostředí pro průběžnou integraci (*Continuous integration*, viz [40]). Výhodou využití tohoto nástroje je odstranění nutnosti vytváření, podepisování a nahrávání APK souboru. Toto vše se provede po každé změně kódu v repositáři. Aplikaci je pak možné několika jednoduchými kroky publikovat do obchodu Google play.

Jediné co zatím nelze je navázání vlastních testů uživatelského rozhraní na sestavení a je nutné je spouštět ručně.

Kapitola 6

# Použité technologie

Tato kapitola popisuje technologie použité pro implementaci praktické části této práce.

## 6.1 Node.js

*Node.js* je javascriptový runtime systém postavený na *Chrome's V8 JavaScript engine* (viz [41]). Node.js používá událostmi řízený (tzv. event driven), neblokující (tzv. non-blocking) I/O model. Je navržený pro vývoj škálovatelných síťových aplikací (serverů). Pro správu balíčků používá systém *npm*. [42]

### 6.1.1 Blokující vs. neblokující volání [43]

Termíny *blokující* a *neblokují* volání (blocking and non-blocking calls) jsou užívány pokud kód, v tomto případě javascriptový, volá nějakou nejavascriptovou operaci. Typicky se jedná o I/O operace (např. čtení či ukládání na disk). Tyto operace zpravidla trvají delší dobu.

Blokující volání je synchronní, tzn. pozastaví program a čeká se na dokončení operace. Pokud při této operaci nastane chyba, je "vyhozena"a musí být zachycena, jinak se program zhroutí. Příklad v Node.Js:

```
const fs = require('fs');  // načtení balíčku pro čtení souboru
const data = fs.readFileSync('/file.md');
// čtení souboru - blokuje (čeká) dokud není soubor přečten
```

Neblokující volání je asynchronní, tzn. program nečeká a provádí další kód. Po skončení dané operace se zavolá callback, který zpracuje případný výsledek. Pokud nastane chyba, autor se může rozhodnout zda ji vyhodí, zpracuje nějak jinak nebo bude ignorovat. Příklad neblokujícího volání v Node.js:

```
const fs = require('fs'); // načtení balíčku pro čtení souboru
fs.readFile('/file.md', (err, data) => { //callback
```

```
if (err) throw err; // zpracování a vyhození chyby
});
// callback se zavolá po dokončení čtení
```

Node.js používá neblokující model, tedy většina volání je asynchronní. Díky tomu lze ušetřit mnoho času. Např. uvažujme požadavek na server, který trvá 50 ms, z toho 45 ms je databázová I/O operace. Pokud by bylo použito blokující volání, pak musíme čekat 45 ms. Při použití neblokujícího volání je těchto 45 ms možno využít pro obsluhu dalších požadavků.

Nebezpečím při používání asynchronních volání je možnost zpřeházení pořadí volání závislých operací. Např. pokud chceme číst ze souboru a poté ho smazat, při nesprávném použití ho můžeme smazat dříve než bude přečten. Příklad špatného použití:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {//asynchroní čtení
if (err) throw err;
console.log(data);
});
fs.unlinkSync('/file.md');  //smazání
//smazání může proběhnout dříve než se dokončí asynchronní čtení
```

Správný přístup:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {//asynchroní čtení
if (err) throw err;
console.log(data);
fs.unlink('/file.md', (err) => { //asynchronní smazání
if (err) throw err;
});
});
//smazání proběhne v~callbacku, tedy po dokončení čtení
```

### 6.1.2 npm

*Npm* je balíčkový (package) manažer pro JavaScript a jeho klient je instalován spolu s Node.js. Obsahuje stovky tisíc balíčků a knihoven, které je možné použít v vlastních projektech. Umožňuje také sdílet a aktualizovat vlastní balíčky, včetně závislostí na jiných.[44]

Npm je založeno na sdílení balíčků (neboli modulů) a znovupoužití hotového kódu. Balíčky jsou většinou malé a úzce zaměřené, případně složené z dalších balíčků. To umožňuje stavět komplexní modulární systémy prostým skládáním již hotových modulů.[44]

Npm začínalo jako balíčkový systém pro Node.js, proto je velké množství balíčků určeno pro servery. Obsahuje ale i balíčky, které přidávají příkazy do příkazové řádky nebo i balíčky použitelné pro frontend v prohlížečích.[44]

## 6.2 JSON

*JSON* (JavaScript Object Notation) je jednoduchý formát pro výměnu dat. Je založený na podmnožině JavaScriptu. JSON je bez problému čitelný i zapisovatelný člověkem a stejně tak jednoduše zpracovatelný strojem. Tyto vlastnosti a nezávislost na žádném jazyku jej dělají ideálním formátem pro výměnu dat. [45]

JSON je založen na dvou strukturách [45]:

- Kolekce párů jméno/hodnota - v mnoha jazycích reprezentován jako objekt, hash tabulka, slovník, struktura apod.
- Seřazený seznam hodnot - typicky reprezentován jako pole nebo list apod.

### 6.2.1 Formát

Tato sekce popisuje formát objektů JSON.

#### 6.2.1.1 Objekt

Objekt je ohraničen složenými závorkami `{}`. V závorkách jsou obsaženy páry jméno:hodnota, každý pár je oddělen čárkou. Jméno je vždy string (řetězec znaků), hodnota může být různá (viz Hodnota dále). [45]

Příklad:

```
{name: "Martin", age: 25}
```

#### 6.2.1.2 Pole

Pole je ohraničeno hranatými závorkami`[]`. Uvnitř závorek jsou hodnoty (viz Hodnota dále) oddělené čárkou. [45]

Příklad:

```
["Matin", "Jatoslav", "Tomáš"]
```

#### 6.2.1.3 Hodnota

Hodnota může být string (řetězec znaků v uvozovkách), číslo (celé, desetinné, s exponentem), objekt (viz výše), pole (viz výše), a speciální hodnoty *true*, *false* a *null*. Hodnoty mohou být vnořovány (tzn. objekt uvnitř objetu, pole uvnitř pole apod.).[45]

Příklad:

```
{ name: "Martin",
age: 25,
Adress: {
Street: "My street",
City: "Prague"
},
image: null,
licence: true,
groups: ["school", "football"]
}
```

## 6.3 MongoDB

*MongoDB* je multiplatformní opensourcová dokumentově orientovaná NoSQL databáze.

### 6.3.1 Dokumentová databáze [46]

Záznamy v MongoDB jsou reprezentovány jako dokumenty - datové struktury složené z párů jméno/hodnota. Tyto dokumenty mají podobný formát jako JSON (viz sekce 6.2).

Výhody dokumentů:

- dokumenty korespondují s datovými type v mnoha programovacích jazycích
- vnořené dokumenty a pole redukují potřebu drahých spojení („joinů“).
- dynamické schéma plně podporuje polymorfismus

#### 6.3.1.1 BSON

MongoDB ukládá dokumenty ve formátu *BSON* (zkratka pro binary JSON). BSON rozšiřuje typy hodnot o několik dalších, např. datum (Date).

Příklad dokumentu:

```
{
_id: ObjectId("5099803df3f4948bd2f98391"),
name: { first: "Alan", last: "Turing" },
birth: new Date('Jun 23, 1912'),
death: new Date('Jun 07, 1954'),
contribs: [ "Turing machine", "Turing test", "Turingery" ],
views : NumberLong(1250000)
}
```

#### 6.3.1.2 _id

Pole `_id` je rezervované jako primární klíč, je vždy na začátku dokumentu, je unikátní v celé kolekci (viz sekce 6.3.2) a může být jakéhokoli typu kromě pole. Pokud není zadané při vložení dokumentu, MongoDB jej vygeneruje automaticky pomocí funkce `ObjectId()`. Ta generuje unikátní 12 bytový objekt, který se skládá z [47]:

- 4 bytové hodnoty reprezentující počet sekund od Unix epochy,
- 3 bytového identifikátoru stroje,
- 2 bytového id procesu,
- 3 bytové náhodné hodnoty.

Výsledné id je pak reprezentováno jako hexadecimální zápis bytů tohoto objektu.

#### 6.3.1.3 Vnořené dokumenty a pole

Pro přístup k vnořeným dokumentům se využívá tečková notace. Například `name.first` vrátí křestní jméno.

Pro přístup k prvkům pole je také používána tečková notace. Například pro přístup k druhému prvku pole se přistupuje takto: `array.2`.

### 6.3.2 Kolekce [46]

V MongoDB se dokumenty ukládají do kolekcí. Ty se dají přirovnat tabulkám v relačních databázích.

Na rozdíl od relačních databází není potřeba kolekce explicitně vytvářet. Explicitní vytvoření je ale možné a lze pomocí něj nastavit parametry kolekce, například maximální velikost.

Kolekce se vytvoří automaticky (pokud neexistuje) vložením dokumentu. Dokumenty v kolekci nemusí mít pevnou strukturu a různé dokumenty mohu mít různé položky.

## 6.4 REST

*REST* je zkratka pro Representational state transfer. REST je styl architektury popsaný šesti podmínkami [48]:

- Klient/server architektura.
- Bezstavovost – stejný dotaz má vždy stejnou odpověď.
- Kešování (průběžné ukládání) odpovědí pro rychlejší odezvu.

- Vrstvený systém – klient neví a nepotřebuje vědět, zda je připojen přímo k serveru nebo k prostředníkovi (např. loadbalancer).
- Jednotné rozhraní.
- (Volitelné) Kód na požádání (Code on demand) – rozšíření funkcionality klienta zasláním spustitelné logiky (např. Java applet, JavaScript).

## 6.5 Android

Následující sekce obsahují popis technologií pro vývoj mobilní aplikace pro operační systém Android. Popisuje nejdůležitější komponenty vývoje, a to Android SDK 6.6 (vývojový kit), základní třídy Android aplikace 6.7 (activity a fragmenty).

Dále obsahují popis možností vývoje v Android Studiu 6.8 a Xamarinu 6.9.

## 6.6 Android SDK

*Android SDK* je sada všech knihoven a souborů potřených pro vývoj Android aplikací. Obsahuje Android SDK manager (viz 6.6.1) pro stahování potřebných knihoven, AVD manager (viz 6.6.2) pro správu emulátorů a ADM pro monitorování a debugging. [49]

### 6.6.1 Android SDK manager

*Android SDK manager* slouží pro správu balíčků a knihoven pro vývoj Android aplikací.

SDK manager umožňuje stahovat různé druhy balíčků popsaných dále.

#### 6.6.1.1 Android SDK Build-Tools

*Android SDK Build-Tools* obsahuje nástroje potřebné k sestavení aplikace. Tato položka je nutně potřebná pro vývoj.[50]

#### 6.6.1.2 Android SDK Platform-Tools

*Android SDK Platform-Tools* obsahuje nástroje potřebné Android platformou. Tato položka je nutně potřebná pro vývoj.[50]

#### 6.6.1.3 Android SDK Tools

*Android SDK Tools* obsahuje základní nástroje potřebné pro vývoj. Tato položka je nutně potřebná pro vývoj.[50]

Součástí jsou například *build-tools*, *debugging-tools* a *image-tools*. [49]

#### 6.6.1.4 Android SDK Platform

*Android SDK Platform* se stahuje pro danou verzi systému, pro kterou se bude vyvíjet. Je možné mít staženo více verzí, ale je potřeba vždy minimálně jedna.[50]

#### 6.6.1.5 Android System Images

*Android System Images* slouží pro emulaci Android zařízení v AVD emulátoru. Image (obraz) systému lze stáhnout pro různé verze systému Android i pro různá zařízení (mobil, hodinky, televize atd.). Existují ve verzích Intel a ARM, kde verze pro Intel nabízí hardwarovou akceleraci se kterou běží mnohem rychleji a plynuleji.[50]

### 6.6.2 AVD manager

*AVD (Android virtual device) manager* slouží pro správu emulátorů. Emulovat lze různá Android zařízení (mobily, tablety, hodinky, televize atd.). Image pro emulaci lze stáhnout např. pomocí Android SKD manageru.[50]

## 6.7 Základní třídy

Základem Android aplikace jsou Activity, které představují jednotlivé obrazovky. Dalším prvkem jsou fragmenty, které se dají skládat a používat v různých aktivitách.

### 6.7.1 Activity

*Aktivita* představuje jednu věc, kterou uživatel může dělat. Většina aktivit komunikuje s uživatelem, proto se třída Activity stará o vytvoření obrazovky. V praxi to znamená, že jedna Activita rovná se jedna obrazovka.[51]

### 6.7.2 Životní cyklus activity

Na obrázku 6.1 je vidět životní cyklus aktivity. Třída Activity poskytuje šest callbacků, které jsou volány při přechodech mezi stavy.

Callbacky:

- `onCreate()` se volá při prvním vytvoření aktivity. Pomocí volání metody `setContentView()` se nastavuje požadované View (UI).
- `onStart()` se volá při přechodu do stavu Start. Aktivita se dostává do popředí a začíná být viditelná uživateli.
- `onResume()` se volá při přechodu do stavu *Resume*. V tuto chvíli uživatel vidí a využívá aktivitu.

- `onPause()` se volá při přechodu do stavu *Paused*. Aktivita je pozastavena na pozadí, může být vidět, ale uživatel s ní nereaguje. Tento stav nastává například při přerušení aktivity např. telefonním hovorem, při otevření dialogového okna apod.
- `onStop()` se volá při přechodu do stavu *Stopped*. Aktivita je překryta jinou aktivitou a není dále viditelná.
- `onDestroy()` se volá před konečným ukončením aktivity. Po proběhnutí je aktivita ukončena a pro její zobrazení je ji potřeba znovu kompletně vytvořit.

### 6.7.3 Fragment

*Fragmenty* se používají pro tvorbu vícepanelových a dynamických uživatelských rozhraní. Umožňují zabalit UI komponenty do modulů, které lze vkládat a odebírat z aktivit. Fragment je chová jako vnitřní aktivita uvnitř jiné aktvivity. Má vlastní životní cyklus a lze jej použít ve více různých aktivitách. [52]

### 6.7.4 View

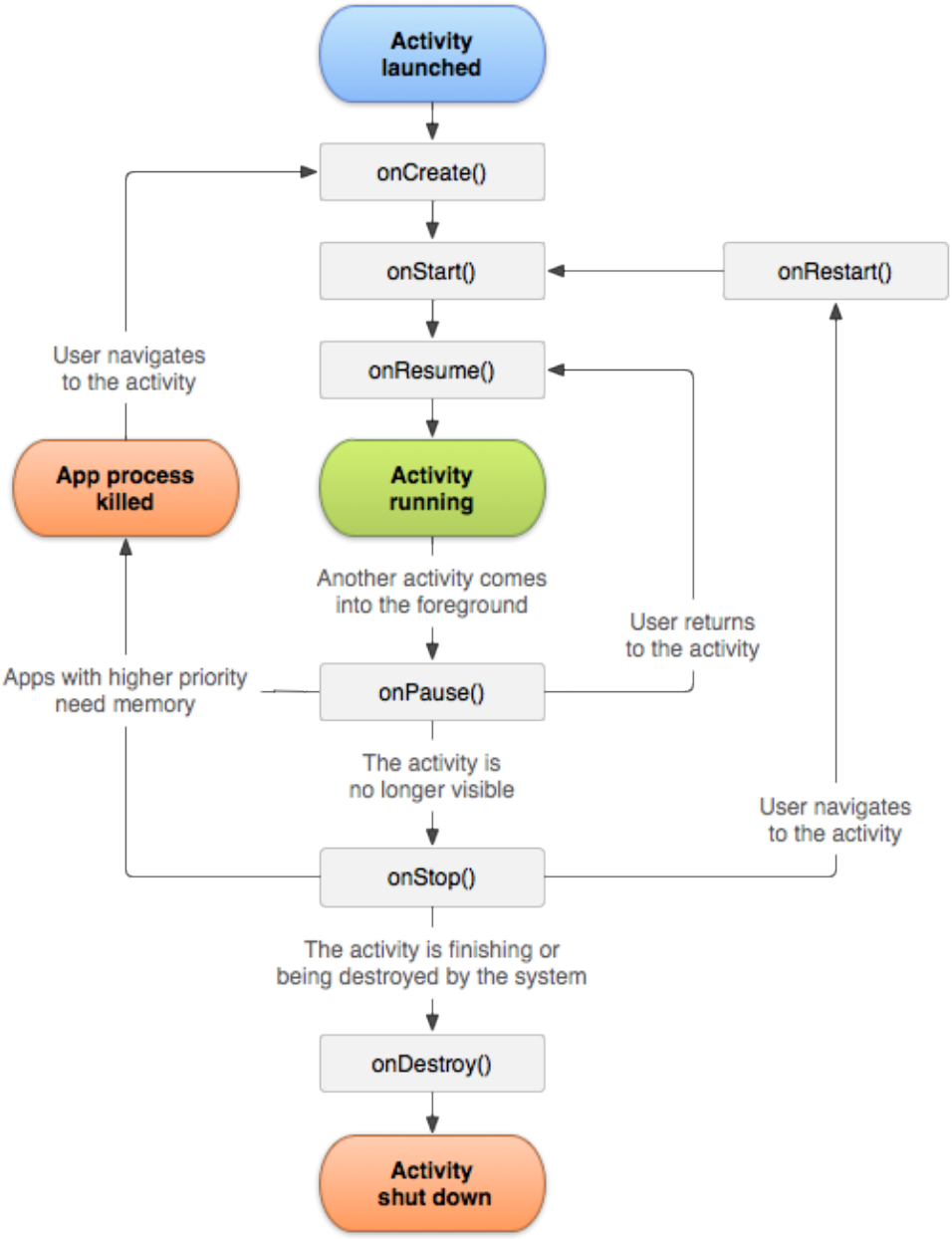
*View* je základní stavební blok komponent uživatelského rozhraní. View zabírá obdélníkovou oblast na obrazovce a je zodpovědné pro vykreslování a řízení událostí. View je hlavní třídou pro *widgety*, které se používají pro vytvoření interaktivních komponent (tlačítka, textové pole, atd.). *ViewGroup* podtřída je základní třída pro *layouty*, které představují neviditelné kontejnery, které jiné View (nebo jiné ViewGroup) a definují vlastnosti layoutu. [53]

## 6.8 Android studio a Java

Oficiálním nástrojem pro tvorbu aplikací pro Android je *Android Studio* a aplikace se píší v jazyku Java. Android studio je postavené na vývojovém IDE IntelliJ IDEA. Tato práce tuto technologii nepoužívá, pro více viz [54].

## 6.9 Xamarin

*Xamarin* je platforma pro vývoj mobilních aplikací, pro vytváření nativních aplikací pro iOS, Android a Windows, ze společného základu v C# / .NET kódu, dosahujících 75 % až téměř 100 % opětovného použití kódu mezi platformami. Aplikace, které jsou napsané pomocí Xamarinu a C# mají plný přístup k nativnímu API dané platformy a mohou vytvářet nativní uživatelská rozhraní a vytvářet balíčky specifické pro každou platformu, takže dopad na výkon je malý.[55]




Obrázek 6.1: Životní cyklus activity [51]

V Xamarinu lze vyvíjet aplikace dvěma způsoby, buď s nativním uživatelským rozhraním (*Xamarin.Android*, *Xamarin.iOS*, také nazývané Xamarin Native), nebo s pomocí *Xamarin.Forms*, které jsou společné a následně se převádí do UI pro konkrétní platformu.

### 6.9.1 Xamarin.Forms

S pomocí Xamarin.Forms lze mezi platformami sdílet více než 96 % kódu. Pro definici uživatelského rozhraní se používá *XAML*. [56]

*XAML* je značkovací jazyk založený na XML a vyvíjený společností Microsoft. XAML je mimo jiné používán ve WPF (Winddows presentation foundation). Díky tomu je použití Xamarinu pro .NET vývojáře jednodušší. Pro více o XAML viz [57].

Mezi nevýhody patří horší výkon a menší množství UI prvků, neobsahuje prvky specifické pro různé platformy.

Platforma Xamarin.Forms je vhodná pro [56]:

- Aplikace, které mají málo funkcionality specifické pro různé platformy.
- Aplikace, kde sdílení kódu je důležitější než vlastní UI.
- Pro vývojáře, kteří jsou spokojeni s XAML.

### 6.9.2 Xamarin.Android

Xamarin.Android umožňuje vytváření nativních Android aplikací s použitím stejného uživatelského rozhraní jako v jazyce Java, ale s flexibilitou a elegancí moderního jazyka C#, síly .NET Base class library (BCL) a prvotřídního IDE (Visual Studio).[58]

Lze tedy používat Android Layouty a obecně prvky uživatelského rozhraní stejné jako pro Javu, s tím že kód je psán v C# místo Android studia je použito Visual Studio.

Platforma Xamarin.Ansroid je vhodná pro [56]:

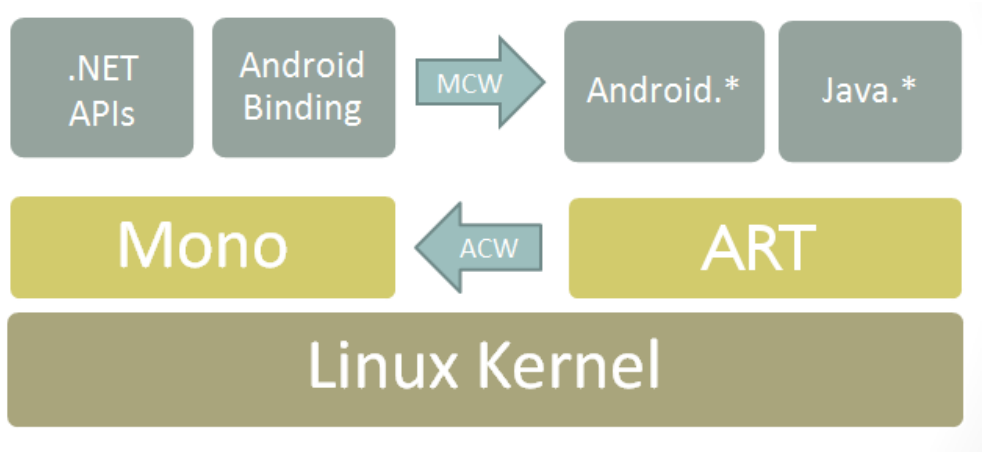
- Aplikace, které vyžadují nativní chování.
- Aplikace, které využívají mnoho pro platformu specifických API.
- Aplikace, kde UI je důležitější než sdílené kódu.

Dále je popsána architektura, tedy jak Xamarin.Android funguje.

#### 6.9.2.1 Architektura

Tato část popisuje, jak Xamarin.Android funguje.

Xamarin.Android aplikace běží na *Mono execution environment* (Mono spouštěcí prostředí). Toto prostředí běží souběžně s *Android Runtime (ART)*




Obrázek 6.2: Xamarin.Android architektura] [59]


Obrázek 6.3: Velikost Xamarin aplikace [60]

*virtual machine*. Obě prostředí běží nad linuxovým jádrem a vystavují různá API pro uživatelský kód, pomocí kterých vývojáři mohou přistupovat k systému. Prostředí Mono je napsáno v jazyku C.[59]

Vývojář může používat `System`, `System.IO`, `System.Net` a ostatní .NET knihovny pro přístup k linuxovému jádru.[59]

Na Androidu není většina systémových prvků, jako audio, grafika, OpenGL a telefon, přístupná přímo z aplikace, jsou vystaveny skrz *Android Runtime Java APIs* ve jmenných prostorech `Java.*` nebo `Android.*`. Architektura vypadá jako na obrázku 6.2. [59]

Xamarin.Android vývojáři tedy mohou přistupovat k operačnímu systému přes .NET API nebo použít třídy z jmenného prostoru `Android`, které vytváření můstek k Java API, které vystavuje Android Runtime.[59]

#### 6.9.2.2 Velikost výsledné aplikace

Velikost výsledné aplikace je o něco větší než u nativní aplikace. To je dáno tím, že aplikace musí obsahovat Mono runtime, Base Class Library (BCL) a případné další knihovny. Struktura výsledného APK balíčku aplikace `HelloWorld` je zobrazena na obrázku 6.3.

Velikost však lze zmenšit pomocí nástroje Linker, který staticky prochází kód a nepoužívané knihovny z Base Class Library (BCL) odstraňuje. Tím je

možné výsledný balíček o něco zmenšit. [60]

### 6.9.3 NuGet a Xamarin Components

Pro sdílení knihoven pro Xamarin existuje balíčkový repositář *Xamarin Components* (viz [61]). Ten se však přesouvá a spojuje k ostatním .NET knihovnám do NuGet galerie (viz [62]).

NuGet je balíčkový manažer pro .NET. NuGet galerie je centrální repositář, využívaný jak autory balíčků, tak jejich uživateli.[63]

Knihovny pro Xamarin je tedy možné stahovat pomocí NuGet přímo z Visual Studia.

### 6.9.4 Xamarin – výhody/nevýhody

Tato sekce pojednává o výhodách a nevýhodách použití Xamarinu.

Výhody (podle [64]):

- Jedna technologie pro všechny platformy.
- Výkon blízký nativním aplikacím.
- Využití nativního UI.
- Plná hardware podpora.
- Opensource technologie s podporou silného korporátu.
- Zjednodušená údržba, díky sdílení kódu.
- Kompletní vývojový ekosystém.
- Xamarin.Forms pro jednoduché aplikace a prototypy.
- Vytváření aplikací pro Mac s Xamarin.Mac

Nevýhody (podle [64]):

- Zpožděná podpora pro nejnovější verze platforem.
- Menší množství opensource knihoven než má Java.
- Problémy Xamarin ekosystému, dané menší komunitou vývojářů.
- Stále je potřeba znát alespoň základy nativního vývoje aplikací.
- Nevhodné pro aplikace zaměřené na grafiku, kde nejde sdílet kód mezi platformami.
- Větší velikost aplikace.

- Nemusí být kompatibilní všechny knihovny třetích stran.

Pro představu zpoždění vydání nové Xamarin verze – Android 8.0 byl vydán 21. 8. 2017 (viz [65]), Xamarin 8.0 byl vydán 9. 10. 2017 (viz [66]).

Zajímavé srovnání výkonu lze nalézt na [67].

Ve shrnutí, Xamarin.Android má u většiny operací srovnatelný výkon jako nativní aplikace pro Android.

Ve srovnání Xamarin.iOS s nativní iOS aplikací je Xamarin.iOS horší.

Xamarin.Forms má ve výkonu výrazně zaostává za všemi ostatními řešeními.

## 6.10 Realm databáze

*Realm databáze* je opensourcová, vestavěná databázová knihovna pro mobilní použití. Na první pohled se může jevit jako objektově-relační mapovač (object-relational mapper – ORM) s lehkou relační databází na pozadí. Ale není tomu tak. Realm používá „data container“ model. Datové objekty jsou v databázi uloženy jako objekty. To má několik výhod [68]:

- Realms ukládá nativní objekty: Realm databáze má navázání na mnoho populárních jazyků pro vývoj mobilních aplikací, včetně Swift, Java, Objective-C, C#, JavaScript (používající React Native). Objekty jsou uloženy jako stejné objekty, které jsou používány ve zbytku kódu.
- Realms je nekopírující: data nejsou kopírování z a do databáze, pracuje se s nimi přímo.
- Realms implementuje vzor živých objektů (live objects pattern): pokud se změní instance objektu uložená v Realm databázi, změna se projeví v ostatních aktuálně používaných instancích.
- Realms je multiplatformní: dokud se neukládají data specifická pro jednu platformu, lze tyto data synchronizovat přes různé operační systémy. (V podstatě stačí zkopírovat soubory obsahující Realm data.)
- Realms je v souladu s ACID (vlastnosti databázové transakce: atomicita, konzistence, isolace, trvanlivost).

Více o Realm databázi na [68].

Kapitola 7

# Řešené problémy a výhledy do budoucna

Tato kapitola popisuje problémy, které byly řešeny při tvorbě této práce. Dále jsou zde také popsány výhledy do budoucna. Tyto dvě témata byly spojeny do této kapitoly, protože spolu často souvisejí.

## 7.1 Obrázky

V původním návrhu aplikace bylo počítáno s tím, že se obrázky zvířat budou stahovat do mobilní aplikace přímo z odkazů dostupných v datech. Po bližším zkoumání bylo zjištěno, že jeden obrázek má v průměru velikost kolem 0,4 MB. To v současném počtu zvířat (přibližně 600, s tím, že všechna nemají obrázky) dává dohromady okolo 240 MB. To je pro mobilní připojení pro většinu uživatelů velké množství dat, které je možné velmi rychle zkonzumovat (např. rychlím posouváním v seznamu zvířat).

Tento problém byl vyřešen tak, že při stahování a aktualizaci dat na serveru se zároveň stahují obrázky zvířat, zmenšují se a ukládají do databáze. Do aplikace jsou stahovány přes vytvořené REST API. To je občas pomalé, z důvodu startu serveru (viz 5.1). Ušetření dat je však značné, naměřený přenos dat pro většinu obrázků byl přibližně okolo 5 MB.

Další variantou by bylo stahování obrázků na serveru ihned po dotazu na API, zmenšení a odeslání, což je ovšem o dost pomalejší. Proto by zde musela být implementována ještě keš (cache) pro rychlou odezvu na další dotazy na stejný obrázek. Tato možnost byla otestována a výsledná doba stažení obrázků byla horší než u předchozí varianty.

Výhled do budoucna je buď vylepšení této funkce na serveru nebo využití nějakého hotového nástroje. Vylepšení by mohlo spočívat nejen ve zrychlení a ušetření dat, ale také např. v možnosti uživatelského nastavení kvality obrázků v mobilní aplikaci.

## 7.2 Kalendář

Překvapivým, i když menším problémem byla implementace kalendáře pro zobrazení kalendáře akcí. Nebylo nalezeno žádné vyhovující řešení pro Xamarin a bylo nutné vytvořit kalendář vlastní.

## 7.3 Překlad

Námětem do budoucna je překlad aplikace do dalších jazyků. Samotný překlad aplikace není problém, vzhledem k tomu, že všechny řetězce jsou v souboru `Strings.xml` a Android nabízí možnost překladu pouze pomocí vytvoření tohoto souboru v jiném jazyce.

Větší problém je se zdrojovými daty, které jsou v této chvíli dostupné pouze v českém jazyce. Překlad do jiných jazyků proto postrádá smysl.

## 7.4 Harmonogram krmení a setkání

Zajímavou funkcí, po konzultaci s několika uživateli, by byl harmonogram krmení a setkání u zvířat v Zoo Praha. Bohužel tato data nejsou dostupná v používaných otevřených datech, ani v žádném přívětivém formátu. Bylo by nutné je buď získat přímo ze zdrojového kódy příslušné stránky webu Zoo Praha, nebo je udržovat ručně. Další variantou vy mohla být domluva se Zoo Praha o poskytnutí a udržování těchto dat v domluveném formátu, např. opět JSON.

## 7.5 Mapa

Výhledem do budoucna s velkým potenciálem je interaktivní mapa Zoo Praha. Nabízí se jak přidání více zájmových bodů, např. občerstvení, případně podrobnější rozmístění zvířat v Zoo Praha. Další možností je vytvoření navigace pro nejlepší trasu v závislosti na zvolených zvířatech apod.

Problémem je nedostatek otevřených dat pro podklady mapy. Bylo by nutné mapu udržovat ručně, případně se domluvit se Zoo Praha o případném dodání podkladů. Případnou další variantou by bylo zapojení komunity uživatelů, kteří by mapu mohli průběžně aktualizovat a vylepšovat.

## 7.6 Vylepšení filtrování

Dalším výhledem do budoucna je vylepšení filtrování, resp. vytvoření možnosti detailnějšího filtrování zvířat. Zajímavé by mohlo být filtrování např. podle biotopů, potravy nebo velikosti a váhy. K tomu je potřeba detailnější rozbor dostupných dat a zkvalitnění některých částí dat (např. proporce zvířete).

Zkvalitnění dat by bylo možné dosáhnout vytvořením nových vlastností (např. délka, výška, váha) a úpravou stávajících dat. Úprava by mohla být realizována buď pomocí vlastní syntaktické analýzy a následné úpravy, nebo případnou domluvou se Zoo Praha.

# Závěr

Závěr této práce obsahuje shrnutí výsledku práce a zhodnocení zvolené technologie Xamarin.

## Výsledek práce

V rámci této práce byla analyzována dostupná data z portálu pražských otevřených dat. Nakonec byla pro výslednou aplikaci zvolena data nacházející se přímo na webu Zoo Praha, na něž se některé datové sady z výše zmíněného portálu odkazují. Kromě samotného lexikonu byla použita datová sada pro aktuální akce v Zoo Praha a RSS kanál novinek Zoo Praha.

Dále byl navržen a vytvořen server v technologii Node.js, který výše zmíněná data stahuje, zpracovává a následně je nabízí přes vytvořené REST API. Pro potřeby serveru byla vytvořena MongoDB databáze. Server byl otestován pomocí aplikace Postman a byly vytvořeny průběžné automatické testy. Ty průběžně kontrolují funkčnost serveru a aktuálnost dat.

Server je nasazený na cloudové službě Heroku, MongoDB databáze běží na cloudové službě mLab. Server běží na URL adrese *https://agile-chamber-32011.herokuapp.com*.

Na základě analyzovaných dat byly vytvořeny wireframy, které byly na pokyn vedoucího třikrát iterovány s vybranými respondenty. Na základě jejich připomínek byly provedeny požadované úpravy. Kromě zobrazení využívaných dat byla do návrhu zapracována i mapa Zoo Praha.

Na základě těchto wireframů byla vytvořena aplikace pro mobilní operační systém Android za využití technologie Xamarin. Aplikace stahuje ze serveru vystavená data přes REST API. Tyto data následně zobrazuje na obrazovkách vytvořených podle navržených wireframů. Po konzultaci s vedoucím bylo grafické pojetí práce zpracované jako jednoduchý Material Design.

Vytvořená aplikace byla otestována jak uživatelsky, tak Unit testy i vlastními automatickými testy uživatelského rozhraní. Stav aplikace a případné pády jsou monitorovány pomocí nástrojů Microsoft App Center.

Microsoft App Center je také použito jako prostředí pro průběžnou integraci. Je napojeno na používaný repositář Bitbucket.org, z kterého stahuje a sestavuje zdrojový kód po každém přispění. Výsledný build pak testuje, zda se spustí na reálném zařízení. Navíc ho podepisuje vloženým certifikátem, díky čemuž je možné aplikaci publikovat do obchodu Google Play rovnou z Microsoft App Center.

Aplikace je nasazena v obchodě Google Play jako otevřená beta verze. Aplikace je nazvána „Lexikon zvířat Zoo Praha“ a lze ji stáhnout z [35].

## Zhodnocení zvolené technologie Xamarin

Pro vývoj aplikace byla zvolena technologie Xamarin s využitím nativního Android uživatelského rozhraní. Následuje osobní zhodnocení práce s touto technologií.

Výsledná zkušenost s Xamarin technologií je velmi pozitivní. Většina věcí týkající se implementace uživatelského rozhraní, aktivit, fragmentů apod. působí stejným nebo podobným dojmem jako vývoj v Android studiu. Jediným rozdílem je jazyk, tedy C# místo Javy.

Objevily se i problémy. Např. při implementaci filtru, je nutné výsledky přetypovávat na Java objekty a následně zpět na .NET objekty. Na takovéto problémy je však na internetu mnoho návodů.

Celkově je počet i kvalita manuálů, návodů, dokumentace a ukázkových projektů velmi dobrá, a to i přes rychlý vývoj a změny Android platformy.

Výhodou je dostatečně velká komunita kolem Xamarin technologie (samozřejmě pořád mnohem menší než u Javy a Android Studia). Díky tomu má většina nástrojů či knihoven používaných pro vývoj v Javě svojí verzi či alternativu i pro Xamarin. A samozřejmě lze použít i spoustu knihoven určených primárně pro vývoj v .NET.

Osobně bych Xamarin doporučil začínajícím mobilním vývojářům zkušenějším více s jazykem C# než s jazykem Java. V obou případech je však nutné znát architekturu Android aplikace. Nevýhody Xamarinu pro běžné aplikace, tedy větší velikost výsledného APK souboru, pomalejší start aplikace a o málo horší výkon nejsou tak kritické.

Z toho plyne, že Xamarin není vhodný pro aplikace, pro které je kritický výkon.

Přechod z Android Studia a Javy na Xamarin má smysl, pokud má aplikace zásadně větší část logiky v backendu, než v uživatelském rozhraní. Pak lze tuto logiku sdílet mezi platformami s vytvořením pouze nativního uživatelského rozhraní pro každou platformu.

Případným dalším důvodem přechodu na Xamarin je rychlé vytvoření jednoduché aplikace s jednoduchým uživatelským rozhraním pro všechny platformy za využití Xamarin.Forms. Vzhledem k nevýhodám Xamarin.Forms

(pomalejší, pouze jednoduché uživatelské rozhraní) je vhodné opravdu jen pro jednoduché aplikace.

V jiných případech nemá pro zaběhlé mobilní vývojáře přechod z Android Studia na Xamarin smysl.

Velkou výhodou pro mě byla předchozí znalost architektury Android aplikací z předchozí zkušenosti s Android Studiem kombinovaná s větší praxí s jazykem C#. Tím se Xamarin stal ideální volbou.

Celkově volbu této technologie hodnotím pozitivně, ačkoli důvody jsou spíše subjektivní. Oproti nativní aplikaci jsou viditelné nevýhody pomalejší start aplikace a větší velikost APK souboru. Výhody jsou subjektivní a to již zmíněný jazyk C# a vývojové prostředí Visual Studio.

# Literatura


[1] Opendata hlavního města Prahy: Zoo Praha. [online].[cit. 2017-04-12]. Dostupné z: http:/opendata.praha.eu/organization/zoo

[2] Opendata hlavního města Prahy: O nás. [online].[cit. 2017-04-12]. Dostupné z: http://opendata.praha.eu/about/

[3] Opendata hlavního města Prahy. [online].[cit. 2017-04-12]. Dostupné z: http://opendata.praha.eu/

[4] CKAN, the open source data portal software. [online].[cit. 2017-04-12]. Dostupné z: https://ckan.org/developers/about-ckan/

[5] XML.com: what is RSS. [online].[cit. 2018-1-6]. Dostupné z: https://www.xml.com/pub/a/2002/12/18/dive-into-xml.html

[6] Pencil project. [online].[cit. 2017-11-25]. Dostupné z: https://pencil.evolus.vn/

[7] Google Play (vyhledané aplikace s kvízy se zvířaty). [online].[cit. 2017-12-29]. Dostupné z: https://play.google.com/store/search?q=animal%20quiz

[8] Apiary. [online].[cit. 2017-12-30]. Dostupné z: https://apiary.io/

[9] API Blueprint. [online].[cit. 2017-12-30]. Dostupné z: https://apiblueprint.org/

[10] Apiary: Animal lexicon Prague Zoo. [online].[cit. 2017-12-30]. Dostupné z: https://lexiconzooprague.docs.apiary.io

[11] Mongoose. [online].[cit. 2017-11-11]. Dostupné z: http://mongoosejs.com/

[12] Express. [online].[cit. 2017-11-25]. Dostupné z: http://expressjs.com/

[13] Sharp. [online].[cit. 2017-11-25]. Dostupné z: http://sharp.dimens.io/en/stable/

[14] rss-parser. [online].[cit. 2017-11-25]. Dostupné z: https://www.npmjs.com/package/rss-parser

[15] SQLite. [online].[cit. 2018-1-6]. Dostupné z: https://www.sqlite.org/

[16] Zoo Praha. [online].[cit. 2017-11-25]. Dostupné z: https://www.zoopraha.cz/

[17] Material design: Guidelines. [online].[cit. 2018-1-5]. Dostupné z: https://material.io/guidelines/

[18] RestSharp. [online].[cit. 2018-1-3]. Dostupné z: http://restsharp.org/

[19] Picasso. [online].[cit. 2018-1-3]. Dostupné z: http://square.github.io/picasso/

[20] GitHub: square-bindings. [online].[cit. 2018-1-5]. Dostupné z: https://github.com/mattleibow/square-bindings

[21] Umlet. [online].[cit. 2018-1-6]. Dostupné z: http://www.umlet.com/

[22] XML Documentation Comments (C Programming Guide). [online].[cit. 2017-12-30]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/xml-documentation-comments

[23] McConnell, S.: *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004, ISBN 0735619670, 9780735619678.

[24] Material design: Navigation. [online].[cit. 2018-1-5]. Dostupné z: https://material.io/guidelines/patterns/navigation.html

[25] Dokumentace API v aplikaci Postman. [online].[cit. 2017-12-25]. Dostupné z: https://documenter.getpostman.com/view/1754868/lexicon/7Lof2g9

[26] Postman. [online].[cit. 2017-12-25]. Dostupné z: https://www.getpostman.com/

[27] Postman: Pricing for monitors. [online].[cit. 2017-12-25]. Dostupné z: https://www.getpostman.com/docs/postman/monitors/pricing_monitors

[28] GitHub: nunit.xamarin. [online].[cit. 2018-1-6]. Dostupné z: https://github.com/nunit/nunit.xamarin

[29] Xamarin.UITest. [online].[cit. 2017-11-25]. Dostupné z: https://developer.xamarin.com/guides/testcloud/uitest/

[30] Microsoft App center. [online].[cit. 2017-12-26]. Dostupné z: https://appcenter.ms/

[31] Heroku. [online].[cit. 2017-12-29]. Dostupné z: https://www.heroku.com/

[32] mLab. [online].[cit. 2017-12-29]. Dostupné z: https://www.mlab.com/

[33] Google Play. [online].[cit. 2017-12-29]. Dostupné z: https://play.google.com/

[34] Android studio: Sign Your App. [online].[cit. 2017-12-29]. Dostupné z: https://developer.android.com/studio/publish/app-signing.html

[35] Google play: Lexikon zvířat Zoo Praha. [online].[cit. 2017-12-29]. Dostupné z: https://play.google.com/store/apps/details?id=com.zavadil.lexiconzoo

[36] GitHub. [online].[cit. 2017-12-30]. Dostupné z: https://github.com/

[37] Bitbucket. [online].[cit. 2017-12-30]. Dostupné z: https://bitbucket.org/

[38] Visual Studio Team Services. [online].[cit. 2017-12-30]. Dostupné z: https://www.visualstudio.com/cs/team-services/

[39] Microsoft: Google Play Store Distribution. [online].[cit. 2017-12-29]. Dostupné z: https://docs.microsoft.com/en-us/appcenter/distribution/stores/googleplay

[40] Fowler, M.; Foemmel, M.: Continuous integration. *Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf*, ročník 122, 2006.

[41] Chrome V8, Google's high performance, open source, JavaScript engine. [online].[cit. 2017-04-14]. Dostupné z: https://developers.google.com/v8/

[42] Node.js: About Node.js. [online].[cit. 2017-04-14]. Dostupné z: https://nodejs.org/en/about/

[43] Node.js: Overview of Blocking vs Non-Blocking. [online].[cit. 2017-04-14]. Dostupné z: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

[44] npm: what is npm. [online].[cit. 2017-04-14]. Dostupné z: https://docs.npmjs.com/getting-started/what-is-npm

[45] Introducing JSON. [online].[cit. 2017-04-14]. Dostupné z: http://www.json.org/

[46] mongoDB: documentation. [online].[cit. 2017-04-14]. Dostupné z: https://docs.mongodb.com/manual

[47] MongoDB Documentation: ObjectId. [online].[cit. 2018-1-2]. Dostupné z: https://docs.mongodb.com/manual/reference/method/ObjectId/

[48] What Is REST? [online].[cit. 2018-1-6]. Dostupné z: http://www.restapitutorial.com/lessons/whatisrest.html

[49] Mullis, A.: Android SDK tutorial for beginners. [online].[cit. 2017-04-14]. Dostupné z: http://www.androidauthority.com/android-sdk-tutorial-beginners-634376/

[50] Update the IDE and SDK Tools. [online].[cit. 2017-04-14]. Dostupné z: https://developer.android.com/studio/intro/update.html#sdk-manager

[51] Android developers: Activity. [online].[cit. 2018-1-6]. Dostupné z: https://developer.android.com/reference/android/app/Activity.html

[52] Fragments. [online].[cit. 2017-11-25]. Dostupné z: https://developer.android.com/guide/components/fragments.html

[53] Android developers: View. [online].[cit. 2018-1-6]. Dostupné z: https://developer.android.com/reference/android/view/View.html

[54] Android studio. [online].[cit. 2018-1-6]. Dostupné z: https://developer.android.com/studio/index.html

[55] Microsoft: Visual Studio a Xamarin. [online].[cit. 2018-1-6]. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/cross-platform/visual-studio-and-xamarin

[56] Xamarin.Forms. [online].[cit. 2018-1-6]. Dostupné z: https://www.xamarin.com/forms

[57] Microsoft: What is XAML? [online].[cit. 2018-1-6]. Dostupné z: https://msdn.microsoft.com/en-us/library/cc295302.aspx

[58] Xamarin: Getting Started with Android. [online].[cit. 2018-1-6]. Dostupné z: https://developer.xamarin.com/guides/android/getting_started/

[59] Xamarin: Architecture. [online].[cit. 2018-1-6]. Dostupné z: https://developer.xamarin.com/guides/android/under_the_hood/architecture/

[60] Xamarin: Application Package Sizes. [online].[cit. 2018-1-6]. Dostupné z: https://developer.xamarin.com/guides/android/advanced_topics/application_package_sizes/

[61] Xamarin Components. [online].[cit. 2018-1-6]. Dostupné z: https://components.xamarin.com/

[62] Xamarin: Hello NuGet! A New Home for Xamarin Components. [online].[cit. 2018-1-6]. Dostupné z: https://blog.xamarin.com/hello-nuget-new-home-xamarin-components/

[63] nuget. [online].[cit. 2018-1-6]. Dostupné z: https://www.nuget.org/

[64] Altexsoft: The Good and The Bad of Xamarin Mobile Development. [online].[cit. 2018-1-6]. Dostupné z: https://www.altexsoft.com/blog/mobile/the-good-and-the-bad-of-xamarin-mobile-development/

[65] Android Developers Blog: Introducing Android 8.0 Oreo. [online].[cit. 2018-1-6]. Dostupné z: https://android-developers.googleblog.com/search/label/official%20launch

[66] Xamarin.Android 8.0. [online].[cit. 2018-1-6]. Dostupné z: https://developer.xamarin.com/releases/android/xamarin.android_8/xamarin.android_8.0/

[67] Altexsoft: Performance Comparison: Xamarin.Forms, Xamarin.iOS, Xamarin.Android vs Android and iOS Native Applications. [online].[cit. 2018-1-6]. Dostupné z: https://www.altexsoft.com/blog/engineering/performance-comparison-xamarin-forms-xamarin-ios-xamarin-android-vs-android-and-ios-native-applications/

[68] Realm: The Realm Database. [online].[cit. 2018-1-6]. Dostupné z: https://realm.io/docs/get-started/overview/

Příloha **A**

# Seznam použitých zkratek

**I/O** Input/output = vstupně/výstupní

**XML** Extensible markup language

**IDE** Integrated Development Environment = vývojové prostředí

**HTML** Hypertext markup language = hypertextový značkovací jazyk

**PDF** Portable document format = formát přenosného dokumentu

**SDK** software development kit

**UI** user interface = uživatelské rozhraní

**API** Application Programming Interface

**UML** Unified Modeling Language

**SQL** Structured Query Language

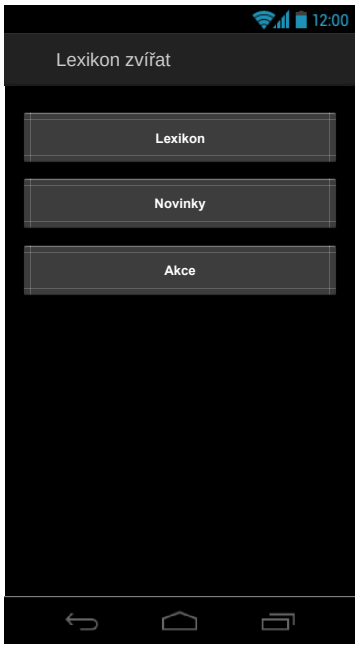
Příloha **B**

# Obsah přiloženého CD

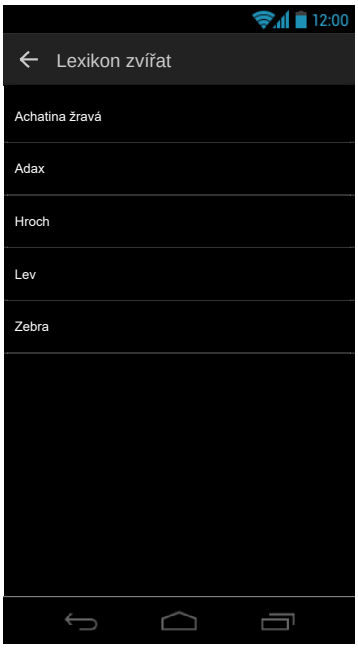
- `readme.txt` ..... stručný popis obsahu CD
- `apk` ..... adresář s APK balíčkem pro instalaci
- `documentation`
  - `Apiary blueprint`
    - `apiary.blueprint.txt` ..... zdrojový blueprint pro apiary
  - `Postman collection`
    - `Lexicon.postman_collection.json` .... export kolekce z Postman
  - `UML` adresář se zdrojovými soubory UML diagramů a exporty do EPS
  - `Wireframes`
    - `html` ..... adresář se exportem wireframů do html
    - `pdf` ..... adresář se exportem wireframů do pdf
    - `src` ..... adresář se zdrojovými soubory wireframů
- `screenshots` ..... adresář se screenshoty aplikace
- `src`
  - `mobile src` ..... zdrojové kódy implementace mobilní aplikace
  - `server src` ..... zdrojová kódy implementace serveru
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF

Příloha C

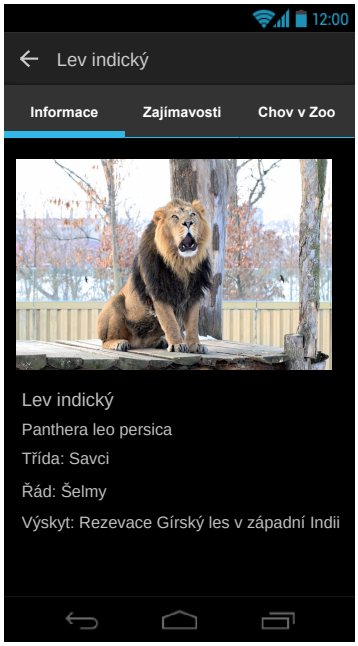
# Wireframy



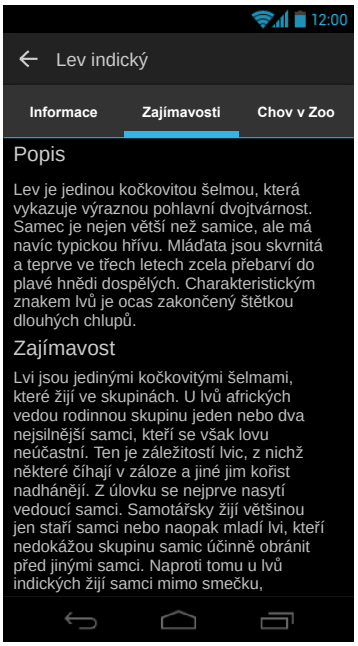

Obrázek C.1: Hlavní menu v.1



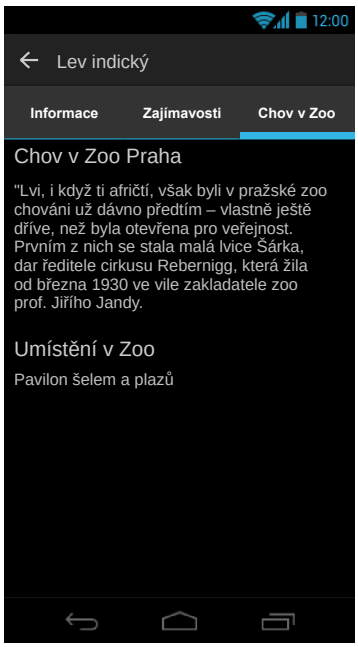

Obrázek C.2: Seznam zvířat v.1



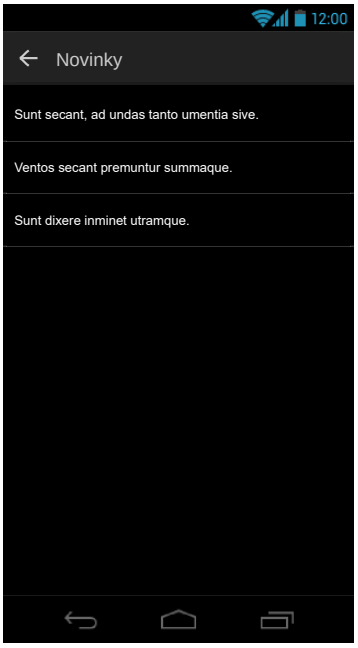

Obrázek C.3: Detail – informace v.1




Obrázek C.4: Detail – zajímavosti v.1

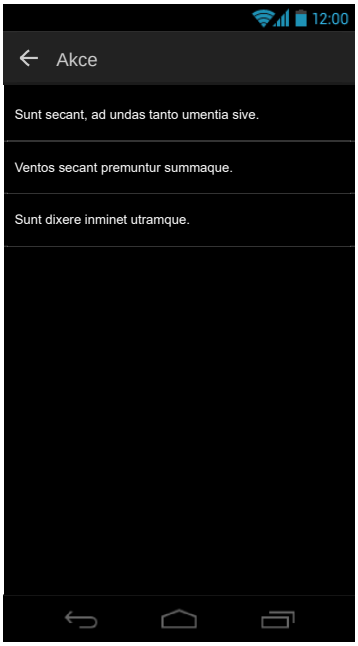



Obrázek C.5: Detail – chov v Zoo v.1



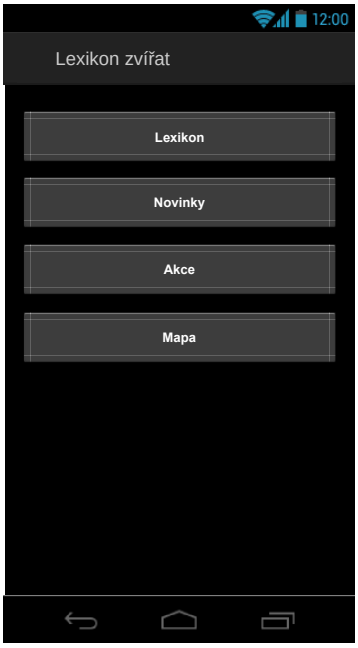

Obrázek C.6: Novinky – seznam v.1


Obrázek C.7: Novinky – Detail v.1




Obrázek C.8: Akce – seznam v.1

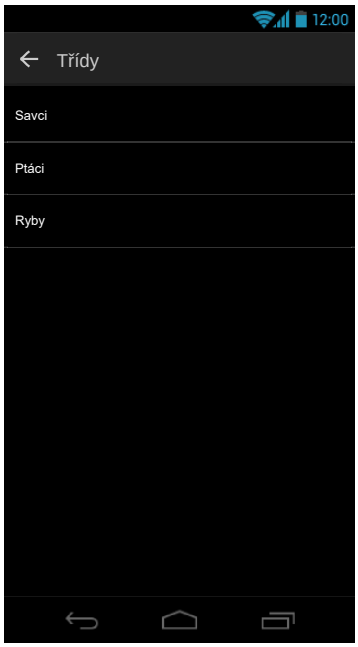

Obrázek C.9: Akce – Detail v.1




Obrázek C.10: Hlavní menu v.2


Obrázek C.11: Menu filtrů v.2

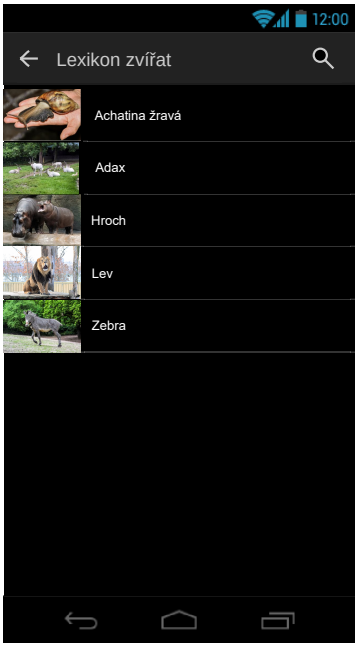

Obrázek C.12: Detail – chov v Zoo v.2



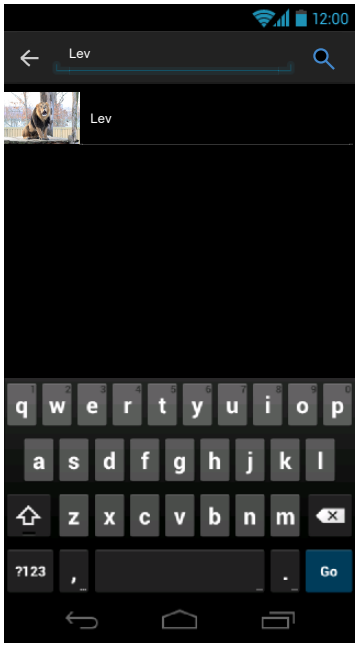

Obrázek C.13: Filtr taxonomie v.2


Obrázek C.14: Filtr umístění v.2

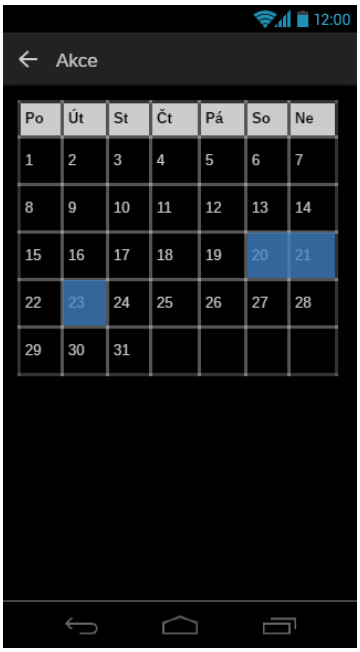

Obrázek C.15: Mapa Zoo v.2




Obrázek C.16: Seznam zvířat v.3



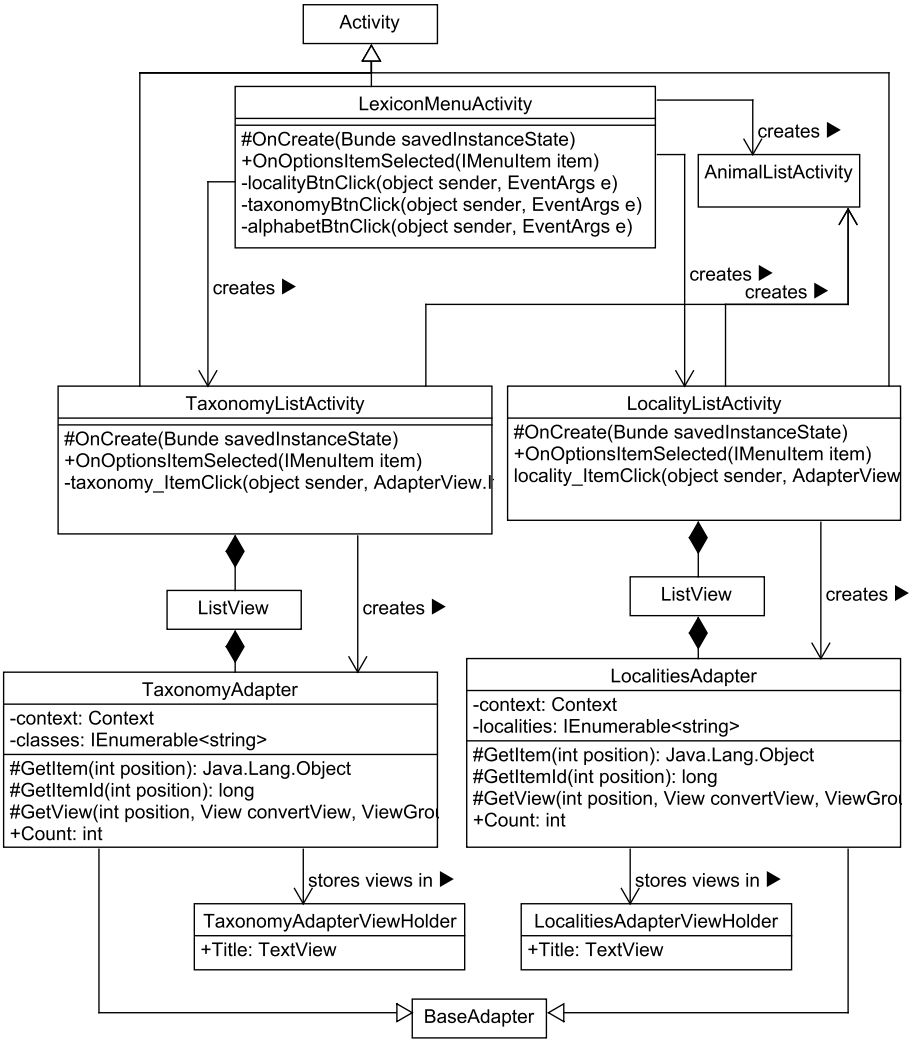

Obrázek C.17: Vyhledávání zvířat v.3



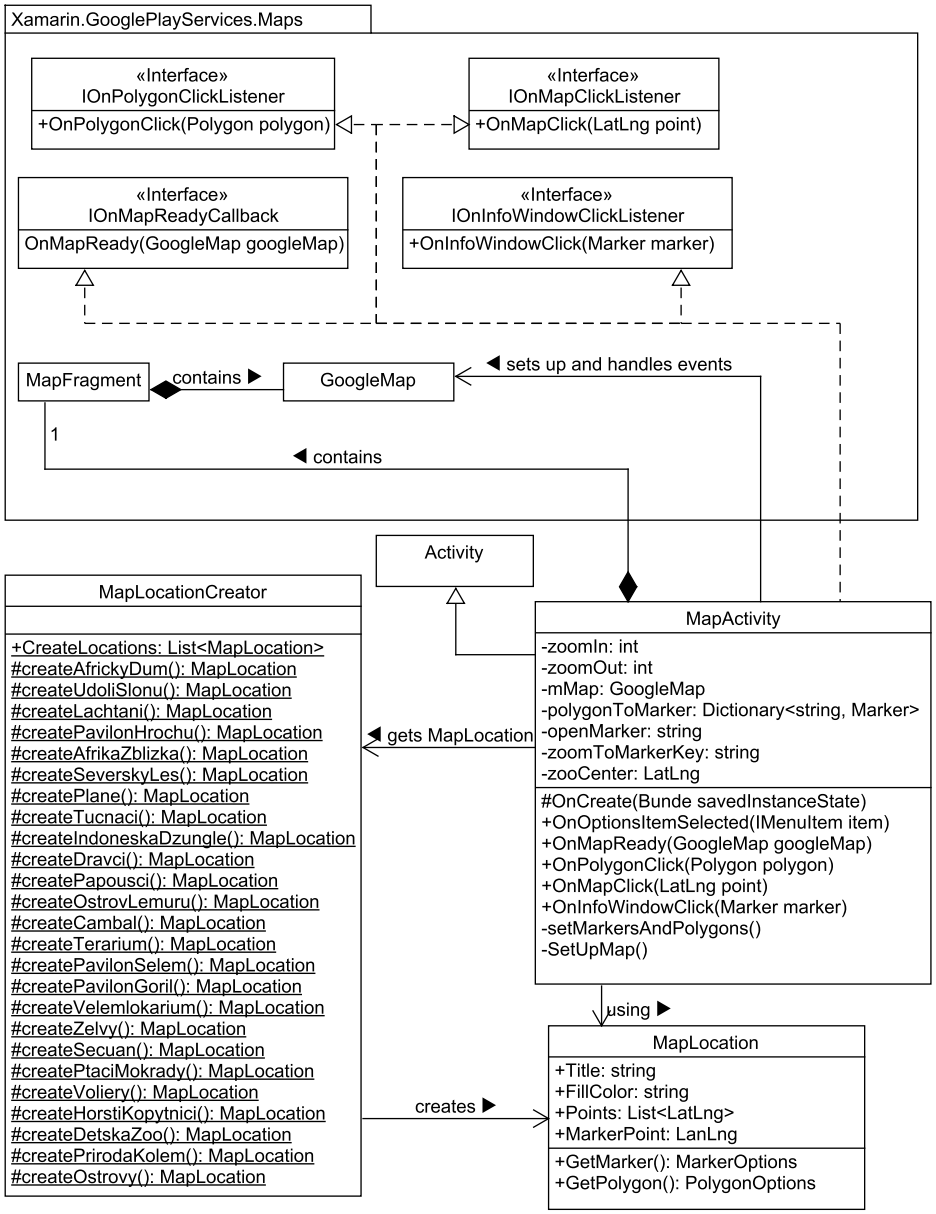

Obrázek C.18: Kalendář akcí v.3

Příloha D

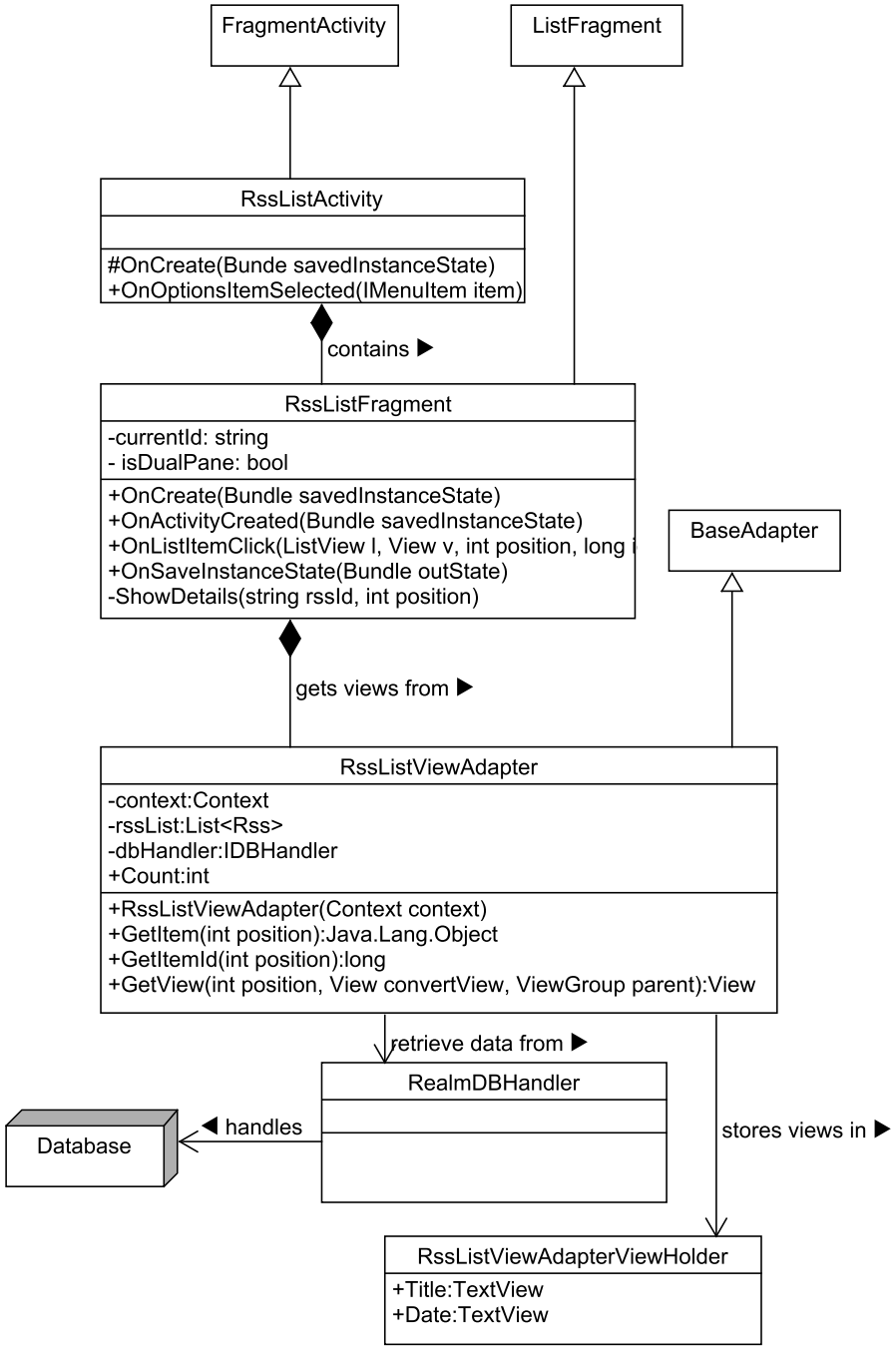
# UML diagamy



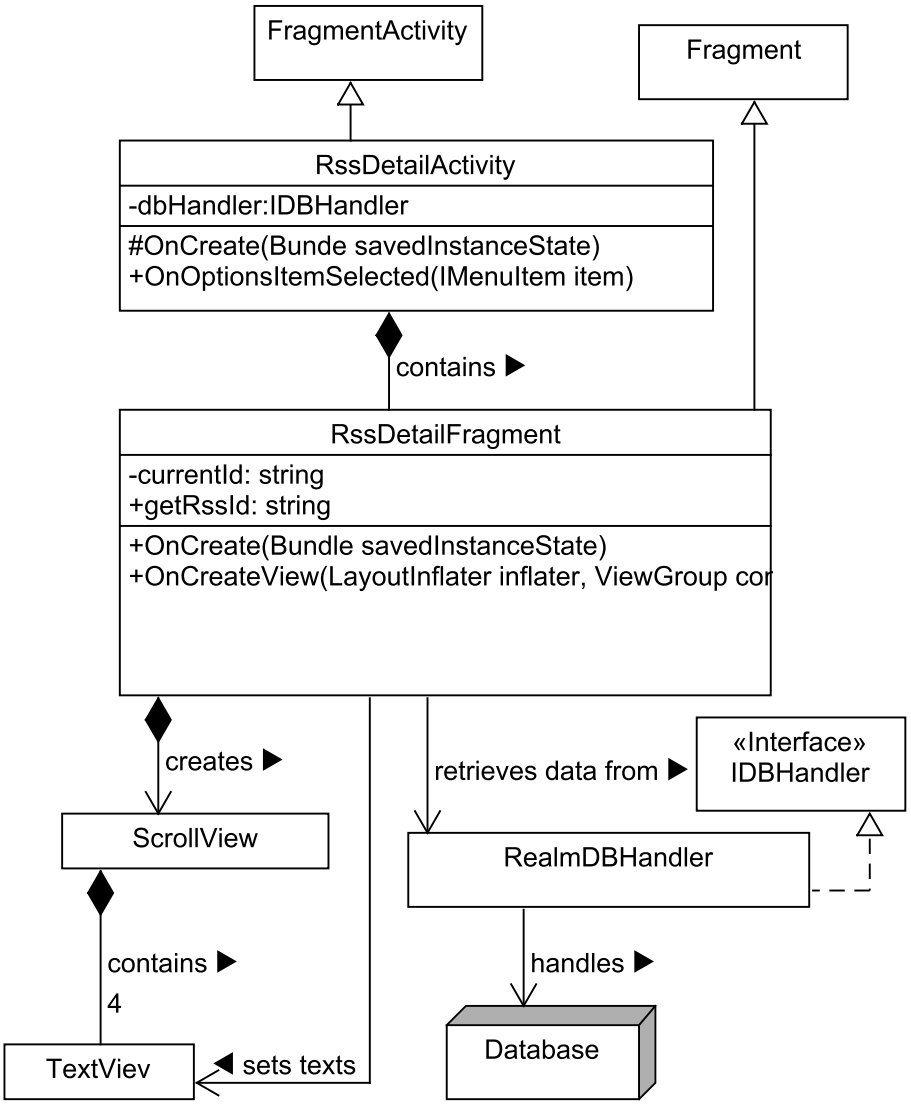

Obrázek D.1: LexiconMenuActivity



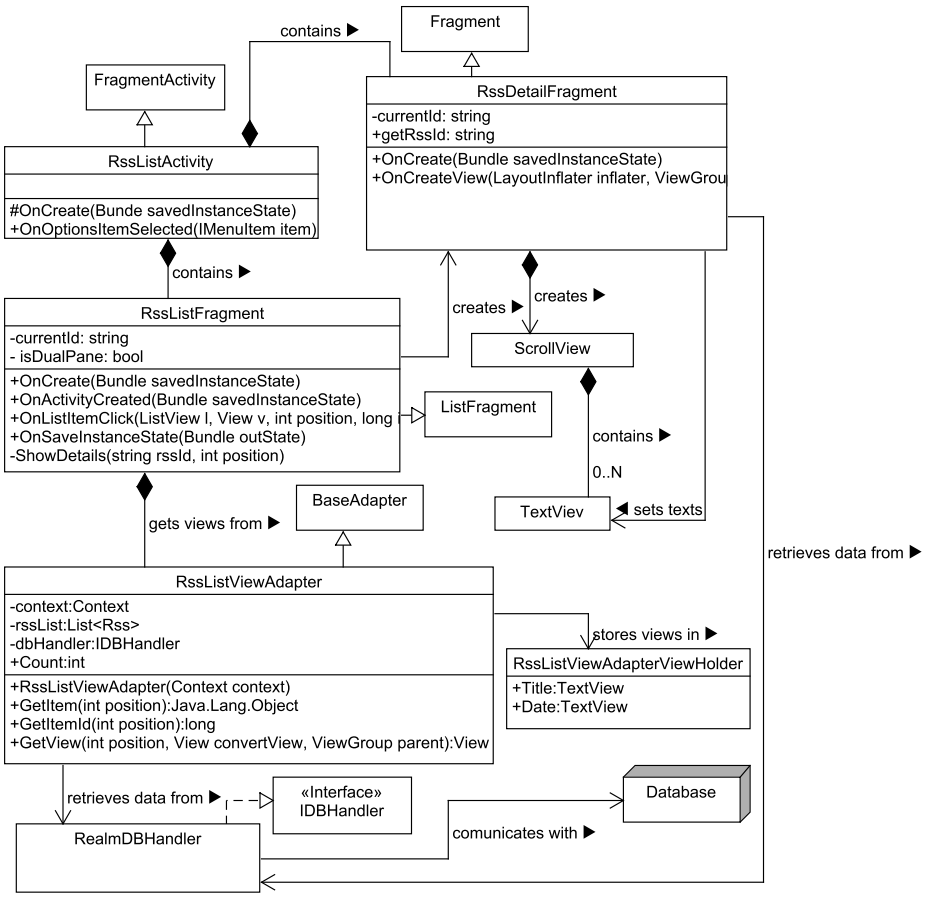

Obrázek D.2: MapActivity



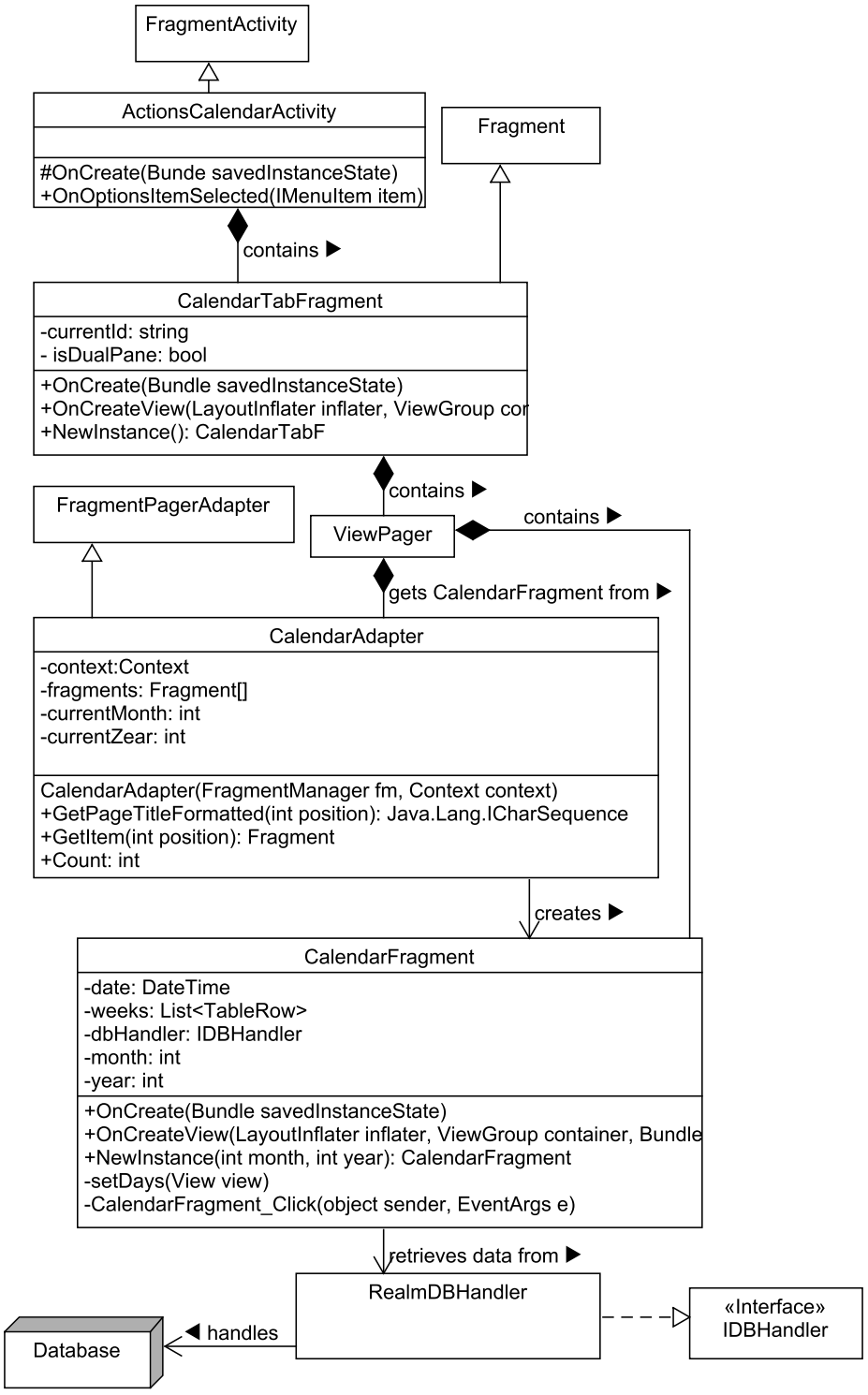

Obrázek D.3: RssListActivity



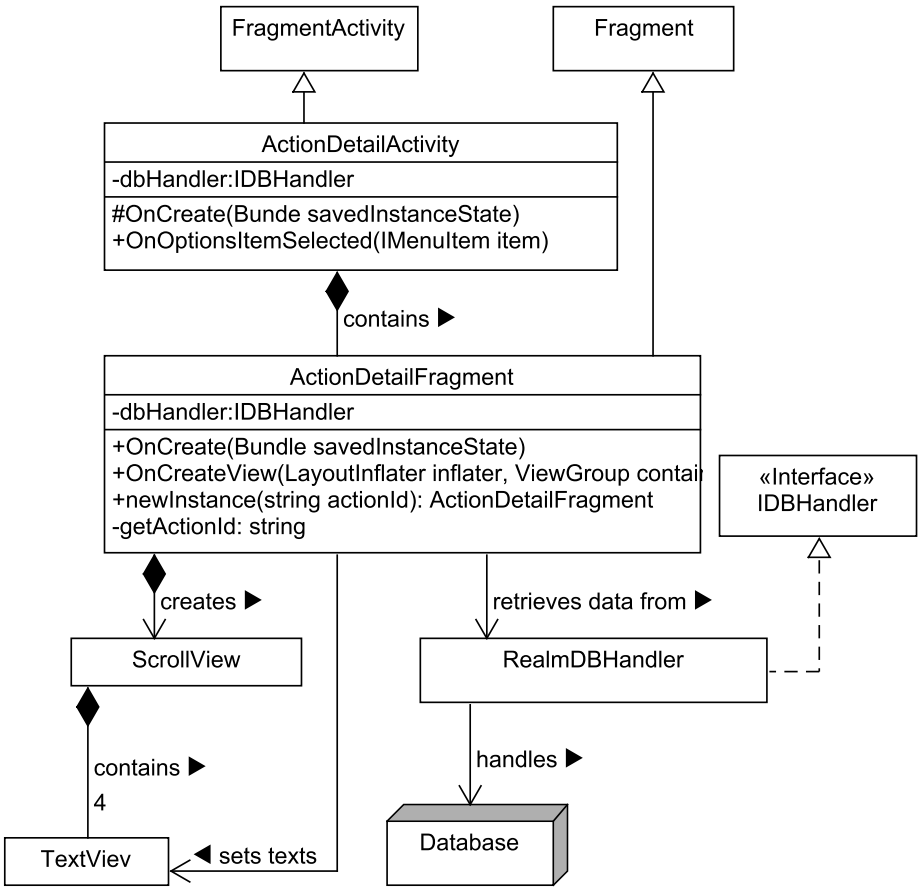

Obrázek D.4: RssDetailActivity



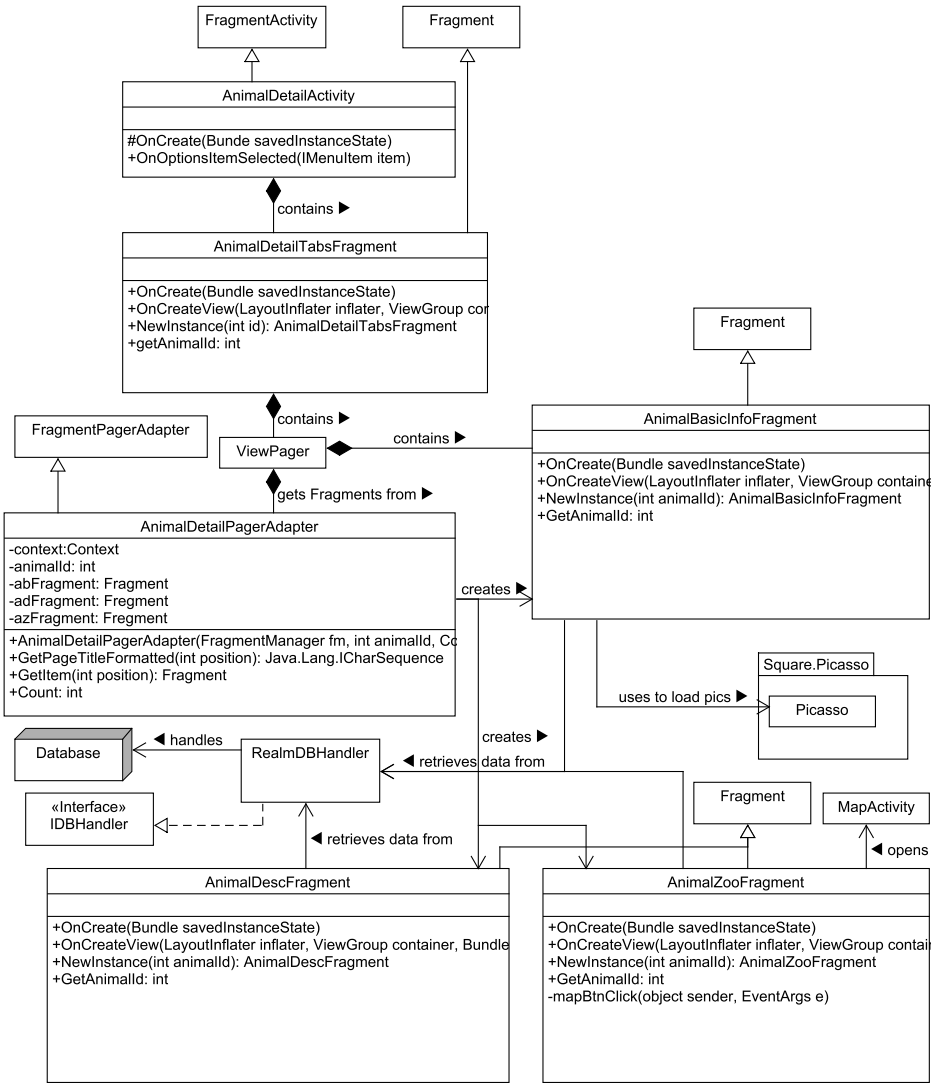

Obrázek D.5: Rss Dual Pane



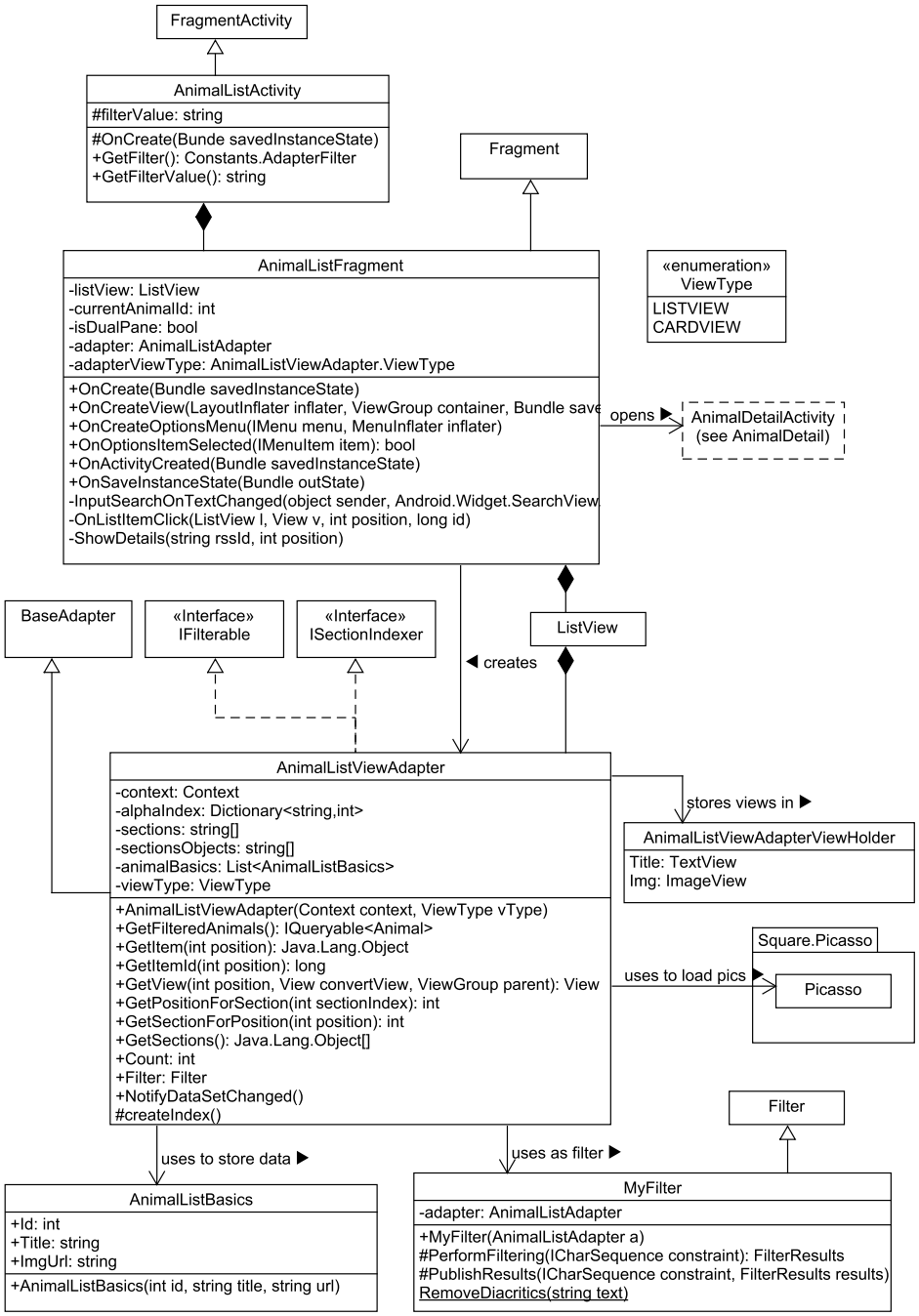

Obrázek D.6: ActionsCalendarActivity



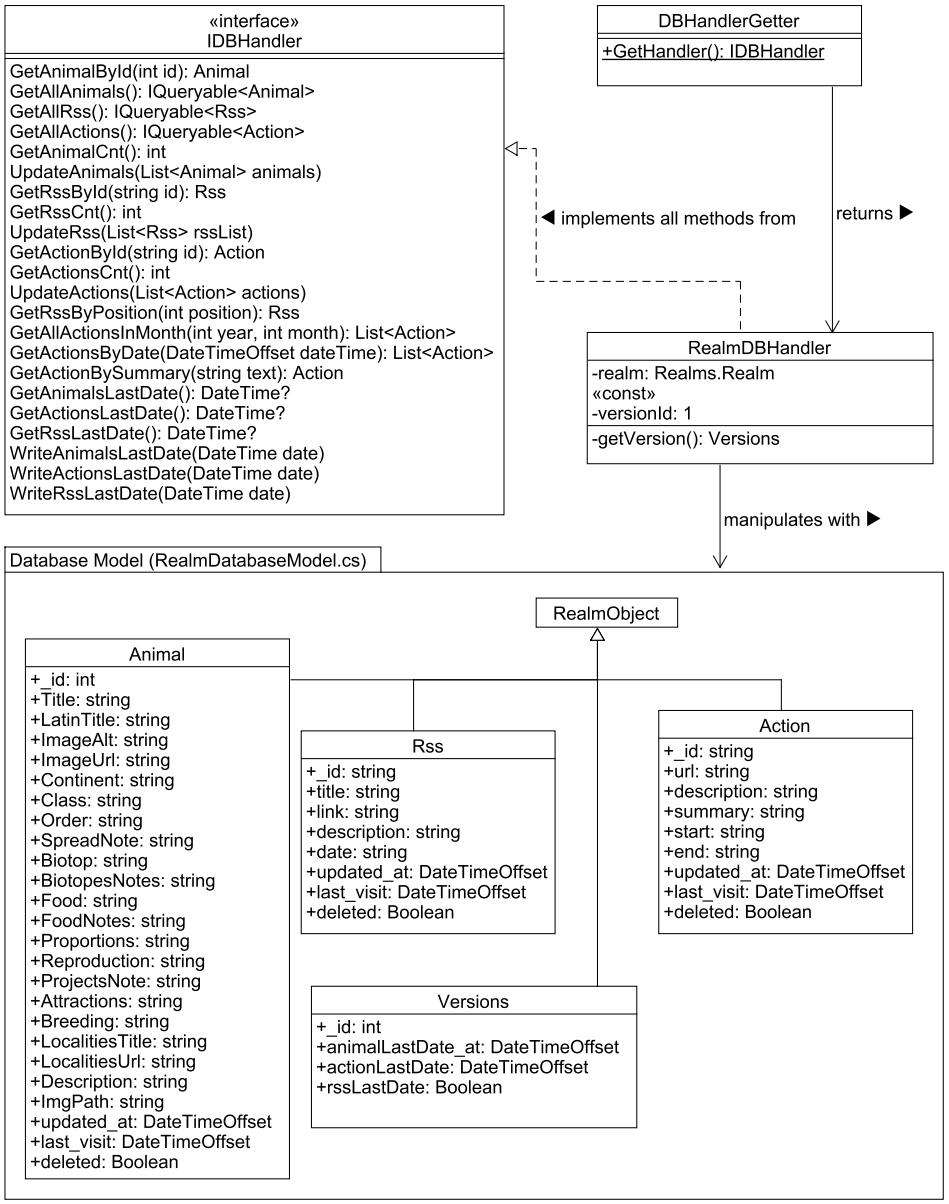

Obrázek D.7: ActionDetailActivity



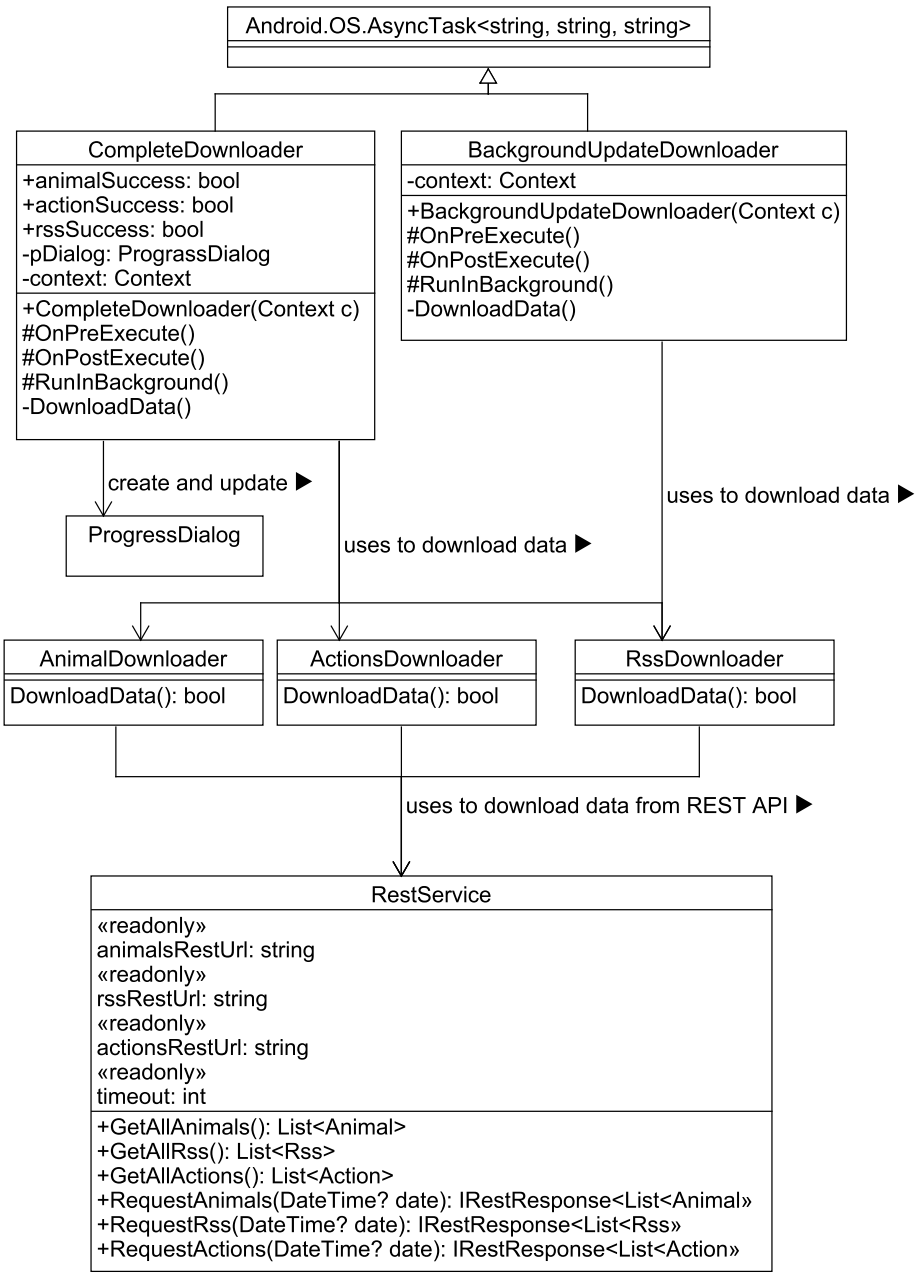

Obrázek D.8: AnimalDetailActivity



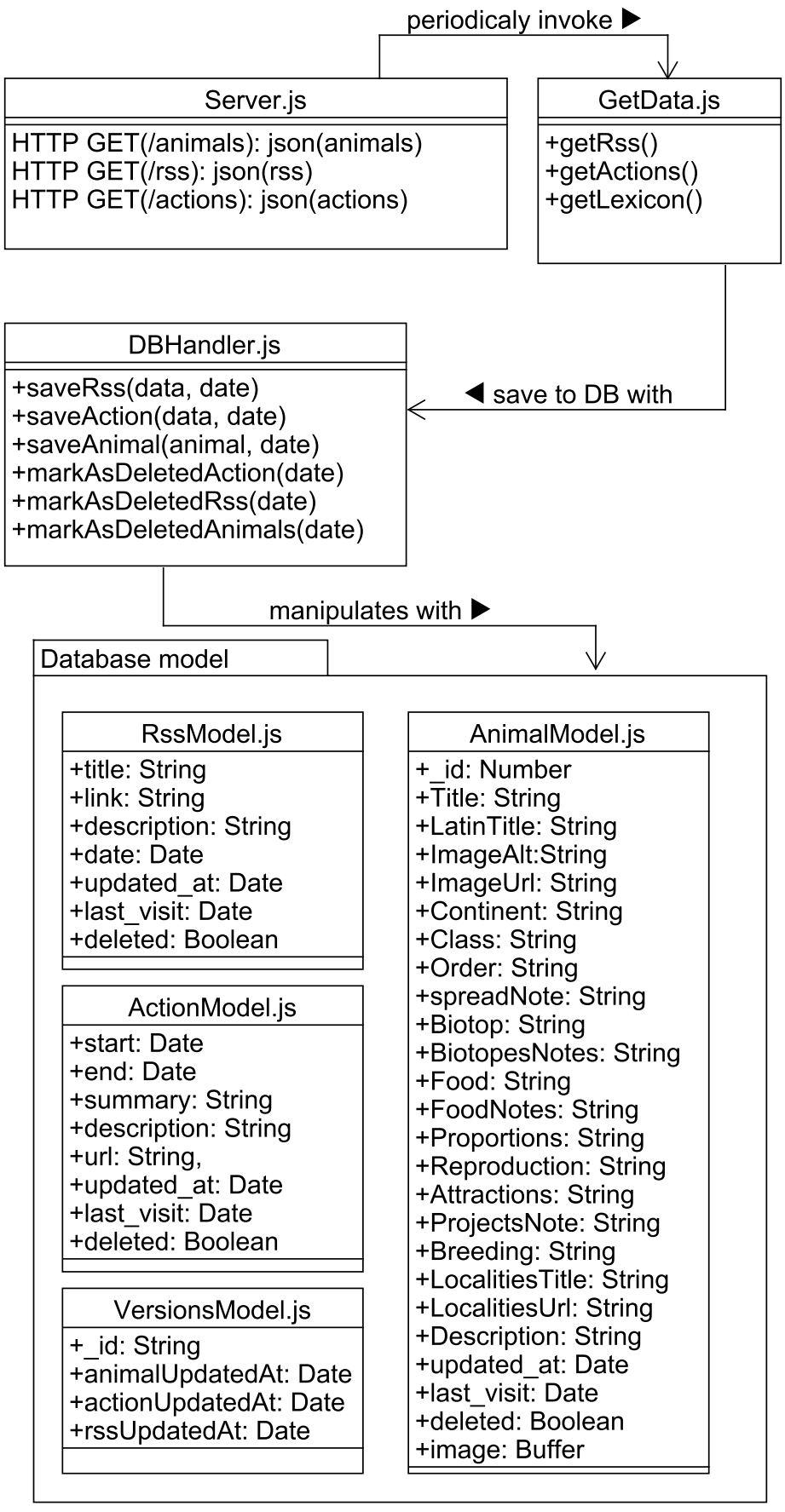

Obrázek D.9: AnimalListActivity




Obrázek D.10: Databázový model




Obrázek D.11: Stahování dat




Obrázek D.12: Server