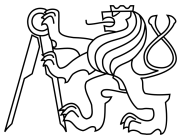
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Mobilní lexikon zvířat ZOO Praha
Student:	Bc. Vít Zdrubecký
Vedoucí:	Ing. Josef Gattermayer
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Pražská ZOO uveřejnila v rámci portálu opendata.praha.eu množství zajímavých informací o zde žijící zvěři. Cílem práce je vytvořit mobilní aplikaci pro telefony a tablety, která tato data atraktivní formou přiblíží návštěvníkům a to včetně serveru, který bude data pravidelně aktualizovat.

1. Navrhněte vhodnou funkcionalitu pro mobilní aplikaci na základě dostupných dat.
2. Navrhněte wireframy aplikace.
3. Implementujte interaktivní model wireframů, tento model otestujte na vybrané skupině uživatelů a zapracujte jejich připomínky.
4. Konzultujte s vedoucím práce grafické pojetí aplikace.
5. Navrhněte a implementujte server v Node.js, který bude data stahovat z portálu opendata.praha.eu a přes REST API nabízet mobilní aplikaci.
6. 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 1. ledna 2017

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

Diplomová práce

Mobilní lexikon zvířat ZOO Praha

Bc. Vít Zdrubecký

Vedoucí práce: Ing. Josef Gattermayer

30. června 2017

Poděkování

Hlavní dík patří mým nejbližším, kteří i v dobách temna, jež často zastiňovaly průběh mých studií, pevně věřili mé schopnosti postavit se sám sobě a stojatá mračna rozehnat. Dále vedoucímu této práce, panu inženýru Gattermayerovi, který její téma zprostředkoval a dále ji vedl i zastřešoval.

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 30. června 2017 .....

Odkaz na tuto práci

Zdrubecký, Vít. *Mobilní lexikon zvířat ZOO Praha.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce obsahuje návrh a implementaci mobilní aplikace pro Android platformu, využívající data poskytnutá Pražskou zoologickou zahradou k zobrazení lexikonu tamějších zvířat a dalších sekcí, jež jsou na ně navázána. Tato data stahuje, zpracovává a transformuje Node.js server, který je následně nabízí prostřednictvím svého aplikačního rozhraní a jehož tvorba je popsána v první části práce.

Klíčová slova ZOO, zvířata, lexikon, fauna, Android, mobilní aplikace

Abstract

This thesis contains the design and implementation of a mobile application for the Android platform, utilizing the data supplied by the Prague ZOO to display the lexicon of local animals and other sections that are linked to them. Those data are downloaded, processed and transformed by a Node.js server, which then exposes them through its application interface and whose creation is described in the first part of the thesis.

Keywords ZOO, animals, lexicon, fauna, Android, mobile applications

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vzhledem k rozmachu snah o zpřístupňování co největšího množství elektronických dat široké veřejnosti, který je v posledních letech čím dál intenzivnější, je důležité se vzniklými informacemi manipulovat a využívat je. Takovou cestou se vydala i tato práce, která využívá možnosti zpracovávat data Pražské zoologické zahrady (dále jen „ZOO Praha“) a prezentovat je zajímavou formou prostřednictvím mobilní aplikace. Propojení těchto dvou světů má smysl, jelikož se dnes chytré telefony i tablety považují za přirozenou součást našich životů a s jejich pomocí jsme zvyklí řešit valnou část denních úkonů. Kromě pracovních a zábavních možností, které jsou aplikacemi na těchto zařízeních poskytovány, vznikají i varianty vzdělávací a čistě informační, což je kategorie, do které spadá výsledek této práce.

Nepochybně by měl splňovat i určitý stupeň zábavnosti, ale jeho hlavním cílem je zprostředkovat uživatelům aplikaci pro platformu Android, která bude výuková a naučná. O dosažení tohoto cíle se primárně snaží lexikon, který se honosí vysokou komplexitou a objemem informací o zvířatech, nacházejících se v ZOO Praha. Z něj vychází i myšlenka vědomostního kvízu, jež atributy zvířat přenáší do světa her a zužitkovává tak kompetitivní faktor, který uživatele nechává překonávat své vlastní výsledky a v rámci toho se i vzdělávat. Dále bude poskytována možnost zobrazovat seznam zvířat k adopci, což je činnost velice prospěšná a vypomáhající při chodu ZOO Praha. Posledním příspěvkem je přehled událostí, které se v ZOO konají, jež slouží k vyšší informovanosti potenciálních návštěvníků.

Podporu mobilní aplikaci poskytuje *backend* server, který bude na pozadí zpracovávat a nabízet předpřipravená data. Tuto činnost bude provádět skrze *API*, aplikační rozhraní popsané množinou operací, které je možné zvenku volat a jež vracejí reprezentaci požadovaných zdrojů. Zdroje jsou definovány svými *URL* (jednoznačným umístěním v síti) a formátem, odpovídajícím standardu *JSON* (notaci vycházející z popisu objektů v programovacím jazyce JavaScript).

Kapitola 1

Open data

1.1 Princip

Open (čili „otevřená“) data si zakládají na myšlence, že některá data by měla být volně dostupná k využívání i opětovnému publikování komukoliv, bez restrikcí a kontrolních mechanismů[1]. V rámci tohoto jejich svobodného šíření mohou být následně prezentována v rámci mnoha zajímavých projektů. Svoboda, s jakou jsou Open data definována, lze definovat s pomocí několika kroků vázaných na jejich publikování:

1. Data jsou vystavena na Internetu pod hlavičkou otevřené licence.
2. K tomu jsou navíc poskytnuta ve strojově čitelném formátu (tzn. nejedná se např. o obrázky).
3. Onen formát není proprietární (což splňuje např. JSON, CSV nebo XML).

Hlavní město Praha se této iniciativy účastní a na jejím webovém portálu lze Open data nalézt na jejich vlastní subdoméně `http://opendata.praha.eu/`, kde se shromažďují datové sady od jejích jednotlivých městských částí a organizací jako je Dopravní podnik, Městská knihovna, Technická správa komunikací nebo právě ZOO Praha. [1]

1.2 Struktura

Organizace ZOO Praha poskytuje v Open datech čtyři datové sady: *Adopce zvířat*, *Akce v ZOO*, *Lexikon zvířat* a *Návštěvnost*, všechny pod hlavičkou licence *CC0*, dopřávající absolutní volnost při jejich využívání. Poslední z nich nabízí celkovou návštěvnost ZOO po letech, počínaje rokem 2005, což je sice

[1] `http://opendata.praha.eu/organization/zoo`

zajímavá informace, ale do celkového konceptu této práce se nehodí. Pozornost tedy byla soustředěna na první tři sady.

1.2.1 Adopce zvířat

Sestává z 528 záznamů, které jsou dostupné v jediném formátu - CSV. Jsou zde k dispozici informace o zvířatech, která mohou případní zájemci sponzorovat - poskytnout finanční příspěvek na krmivo, chov a péči o zvíře po celý rok. Tato částka se liší podle druhu a je samozřejmě zahrnuta i v Open datech. Dále je zde příznak, zda lze zvíře navštěvovat, jeho český i anglický název a taxonomická třída, do které spadá. Bohužel ale chybí seznam benefitů, které sponzor za svůj příspěvek obdrží (může se jednat o certifikát, roční vstupenku do ZOO apod.), který snad bude časem přidán, poněvadž na webu pražské ZOO je dostupný.[2]

1.2.2 Akce v ZOO

Jedná se o kalendář událostí, které se v ZOO pořádají. Data jsou dostupná ve dvou odlišných formátech, jejichž zdroje jsou téměř identické jak obsahem tak velikostí - 71 záznamů za rok 2017 je přístupných buď v JSON nebo v CSV. U každého z nich je datum začátku a konce, spolu s názvem akce a jejím detailním popisem (např. `Den hmyzožravců`: *Ochutnejte hmyz na všechny způsoby a podívejte se, jak chutná hmyzožravcům v zoo.*), kde se také nachází jediný rozdíl mezi zdroji - v případě JSON je popis obohacen HTML značkami.

1.2.3 Lexikon zvířat

Obsahuje z vybraných sad největší množství informací a tvoří jádro této práce. Jeho základem je trojice stejnojmenných zdrojů, nabízejících přehled o všech zvířatech, jež se v ZOO nachází. Zdroje se opět liší ve svém formátu i obsahu - první dva jsou reprezentovány pomocí CSV (328 záznamů) a XLSX (581 záznamů), přičemž jejich data mají stejné parametry a liší se právě jen jejich počtem. Třetí zdroj je v JSON a je v něm také celých 581 zvířat, bohužel ale některá pole obsahují text obohacený o HTML značky, čímž by se jejich zpracování ztížilo. Navíc objekty nemají odkazy na své obrázky ve zvláštním atributu, ale přímo zakomponované jakožto součást popisu (opět v HTML), což dává další výhodu zdroji ve formátu XLSX, který oba zmíněné neduhy postrádá a obsahuje aktuálně maximální počet zvířat, čili na něj bude soustředěna pozornost.

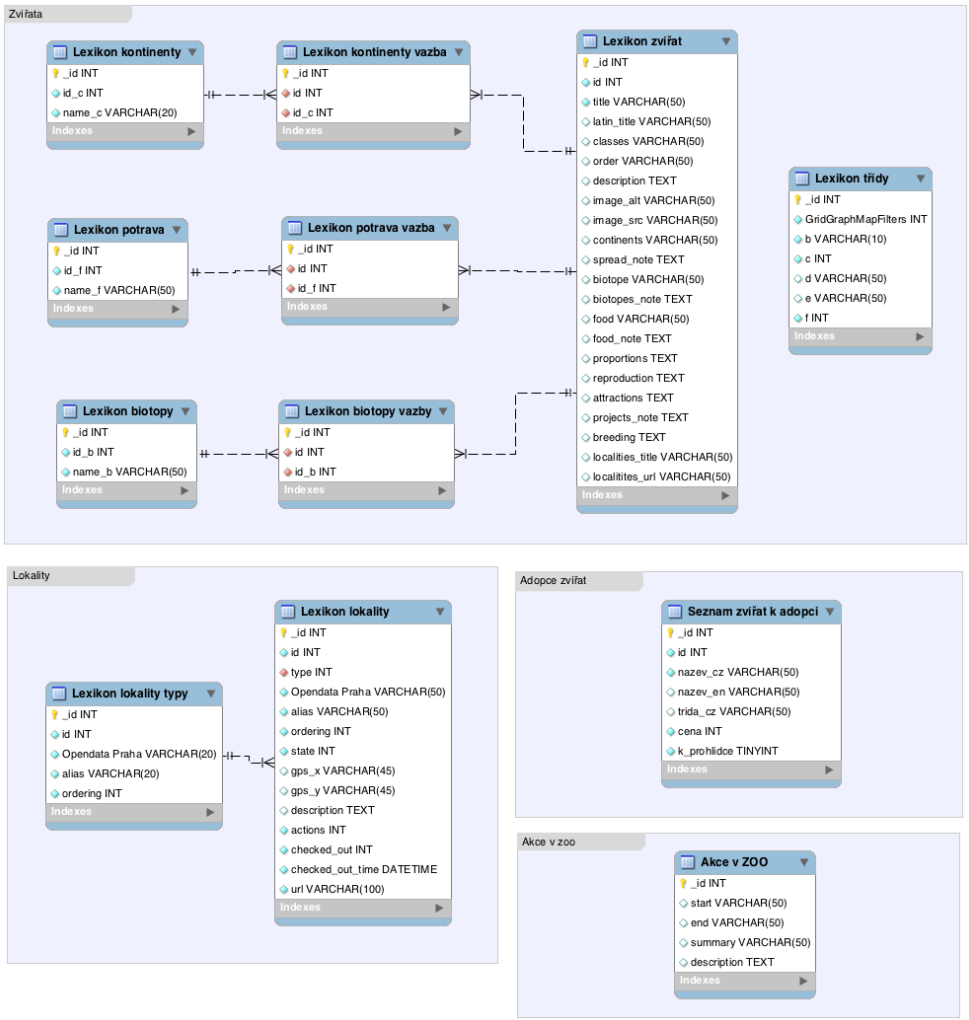
Lexikon dále představuje množství textových informací o zvířatech, jako je jejich detailní popis, rozměry, umístění v areálu ZOO, typy potravy a samozřejmě český i latinský název. Dále jsou zde samostatná pole pro taxonomickou třídu a řád, jejichž odpovídající záznamy jsou v následujícím zdroji - *Lexikon*

[2] `https://www.zoopraha.cz/jak-pomoci/adopce/seznam-zvirat-pro-adopci`

třídy. Ten obsahuje taxonomii včetně hierarchie, tudíž jsou zde řády navázány na své rodičovské třídy. Na tyto hodnoty je možné zvířata z lexikonu mapovat pouze pomocí názvu taxonomie, jelikož propojení pomocí cizího klíče bylo aktualizací sady v Open datech ztraceno. Spolu s ní i navázání na čeleď, která je v `Lexikonu třídy` stále přítomna včetně umístění v hierarchii, ale bohužel nepoužitelná. Dalšími přítomnými zdroji, které jsou aktuálně zbytečné, jelikož jsou informace v nich obsažené přesunuty přímo ke zvířatům do jejich vlastního lexikonu, jsou *Lexikon biotopy*, *Lexikon potrava* a *Lexikon kontinenty* včetně vazebních tabulek, poněvadž byly dříve napojeny na zvířata skrze relace.

Zbývají *Lexikon lokality* a *Lexikon lokality typy*, kde jsou zanesena místa v areálu ZOO. V nich je nyní hodně informací nevyplněných, ale v budoucnu by se mohly doplnit a využít. Jedná se především o GPS souřadnice a informace o otevírací době.

Diagram na obrázku 1.1 reprezentuje skladbu i obsah všech dostupných sad a zdrojů, které budou využity pro import. Data jsou rozdělena do logických celků, aby byly vzájemné relace zjevnější.



Obrázek 1.1: Schéma datových sad a zdrojů v Open datech

1.3 Specifika

Ve zdrojových datech se nalézá množství případů, kdy je nutné měnit jejich strukturu, doplnit potřebné informace nebo transformovat dostupné hodnoty. Je potřeba vytvořit spojení mezi adopcemi se zvířaty z lexikonu, kterých se týkají, což půjde pouze na základě jejich názvů. Na více místech se bude hodit rozdělení názvů taxonomických tříd a řádů, které jsou nyní pokaždé v jednom řetězci, na jejich českou a latinskou verzi, aby mohly být využity pro lepší zobrazení uživateli i dotazování na `backend API`.

Dále je třeba řešit případy, kdy je obrázek zvířete nevyplněný a může být

obsažený v rámci popisu, jak to bylo před aktualizací sady v Open datech. V takové situaci by se odkaz na něj z popisu vypreparoval a samotný popis by byl zbaven všech HTML značek. Také je potřeba omezit pole, které se budou při importu získávat, aby se zbytečně nepřenášela ta, která nebudou potřeba. To je spojeno s názvy polí, která ve většině zdrojů nejsou vůbec popisná a jejich smysl je třeba hádat z obsahu. Ty je po importu nutné smysluplně přejmenovat.

1.4 Způsob získávání dat

Každý datový zdroj je v rámci Open dat jednoznačně identifikován řetězcem alfanumerických znaků, pomocí kterého lze ke zdroji přistupovat a buď v něm provádět změny nebo se na něj dotazovat. Příslušné `endpointy` jsou tyto:

- `http://opendata.praha.eu/api/action/datastore_create` - vytváření dat
- `http://opendata.praha.eu/api/action/datastore_upsert` - aktualizace nebo vložení dat
- `http://opendata.praha.eu/api/action/datastore_search` - dotazování pomocí `query` parametrů
- `http://opendata.praha.eu/api/action/datastore_search_sql` - dotazování pomocí *SQL*

Za každým z nich následuje `query` parametr *resource_id*, který reprezentuje patřičný zdroj. První dva přirozeně vyžadují autentizaci a autorizaci, ale dotazování je volně dostupné. Další parametry, které je možné aplikovat, slouží k limitování počtu vrácených záznamů, jejich řazení, výběru atributů každého záznamu nebo jejich filtrování podle požadovaného řetězce. V případě dotazu s `SQL` je připojen databázový dotaz, který může obsahovat i složité databázové operace a přenechat tak veškerou práci serveru. Dokumentace k `API` Open dat je dostupná v anglickém jazyce.[3]

[3] `http://docs.ckan.org/en/latest/maintaining/datastore.html`

Kapitola 2

Návrh

Každá sekce návrhu obsahuje svůj vlastní seznam funkčních požadavků. Seznam těch nefunkčních je společný pro všechny, jelikož definuje vlastnosti produktu jako celku:

- Mobilní aplikace bude cílená na operační systém Android
- Mobilní aplikace bude mít k dispozici `backend` server, se kterým bude synchronizovat data
- Mobilní aplikace bude mít schopnost fungovat offline pomocí persistence importovaných dat
- Mobilní aplikace bude svým návrhem pokrývat co největší množství různých zařízení

2.1 Lexikon zvířat

Jádrem finální mobilní aplikace bude prezentace lexikonu. V něm budou obsažena všechna zvířata, která jsou v Open datech dostupná a mají korektně vyplněné hodnoty, jež je popisují. Díky této funkci budou mít uživatelé k dispozici rozsáhlou databázi informací, ve které si budou moci okamžitě zobrazit cokoliv je právě zajímá o tom kterém zvířeti. Data obsažená v lexikonu budou taktéž sloužit jako základ ostatním dostupným částem aplikace, které z něj budou čerpat a snažit se na něj napojit, kdykoliv bude potřeba.

Seznam funkčních požadavků:

- Uživatel mobilní aplikace si bude moci zobrazit seznam zvířat z lexikonu
- Uživatel mobilní aplikace si bude moci zobrazovaná zvířata filtrovat podle následujících kritérií:
  - Klasifikace (třídy a řády zvířat)

  - Základní druhy potravy, kterou se zvířata živí
  - Světové kontinenty jakožto místa výskytu zvířat
  - Přírodní biotopy, jež zvířata ve volné přírodě obývají
  - Umístění zvířat v objektu ZOO Praha

- Uživatel mobilní aplikace bude moci v zobrazeném seznamu zvířat vyhledávat podle jejich názvu
- Uživatel mobilní aplikace si bude moci zobrazit detail vybraného zvířete ze seznamu, obsahující rozsáhlejší informace o něm
- Uživatel mobilní aplikace si bude moci na detailu vybraného zvířete zobrazit zvětšenou verzi jeho obrázku, pokud je k dispozici
- Uživatel mobilní aplikace se bude moci z detailu vybraného zvířete dotykem dostávat na detaily sousedních zvířat ze seznamu

2.2 Seznam zvířat k adopci

Stejně jako lexikon bude i tento seznam uživatelům plně k dispozici. Vzhledem k tomu, že sám o sobě neposkytuje detailní informace o zvířeti, kterého se adopce týká, bude s lexikonem propojen kdekoliv to bude možné. Charakter zobrazování bude zatím pouze informační - samotnou adopci nepůjde zařídit přímo z aplikace.

Seznam funkčních požadavků:

- Uživatel mobilní aplikace si bude moci zobrazit seznam zvířat určených k adopci
- Uživatel mobilní aplikace bude moci v zobrazeném seznamu zvířat vyhledávat podle jejich názvu
- Uživatel mobilní aplikace si bude moci zobrazit detail vybraného zvířete ze seznamu, obsahující o něm rozsáhlejší informace získané z lexikonu
- Uživatel mobilní aplikace si bude moci na detailu vybraného zvířete zobrazit zvětšenou verzi jeho obrázku, pokud je k dispozici

2.3 Přehled událostí

Další seznam, obsahující akce konané pod záštitou ZOO Praha. Zprostředkuje uživatelům seřazený přehled událostí, které právě probíhají nebo se budou konat v budoucnu.

Seznam funkčních požadavků:

- Uživatel mobilní aplikace si bude moci zobrazit seznam událostí pořádaných v ZOO Praha
- Uživatel mobilní aplikace si bude moci zobrazit detail vybrané události ze seznamu, obsahující rozsáhlejší informace o ní

2.4 Kvíz

Vědomostní kvíz bude sloužit jakožto zábavná prověrka znalostí, které uživatel nasbíral o zvířatech vyskytujících se v lexikonu. Stejně tak bude testovat pohotovost a schopnost dávat informace do souvislostí.

Seznam funkčních požadavků:

- Uživatel mobilní aplikace bude moci spustit instanci vědomostního kvízu
- Odpověď na kvízové otázky bude časově limitována a po vypršení viditelného intervalu bude uživateli zobrazena odezva
- Uživatel mobilní aplikace bude moci na každou otázku kvízu odpovědět, na což mu bude okamžitě zobrazena odezva
- Uživatel mobilní aplikace si bude po zodpovězení otázky nebo vypršení časového limitu moci zobrazit detail zvířete, kterého se otázka týká, s informacemi získanými z lexikonu
- Uživatel mobilní aplikace si bude moci na detailu zvířete ukázaném v průběhu kvízu zobrazit zvětšenou verzi jeho obrázku, pokud je k dispozici
- Uživatel mobilní aplikace bude moci po zodpovězení otázky nebo vypršení časového limitu označit otázku jako chybnou (pokud nebude v pořádku její znění nebo odpovědi)
- Uživatel mobilní aplikace bude v průběhu kvízu zobrazována informace o aktuálním skóre a pořadí otázky
- Uživateli mobilní aplikace bude po skončení kvízu zobrazen jeho výsledek
- Uživatel mobilní aplikace si bude moci zobrazit pořadí nejlepších výsledků kvízu
- Uživatel mobilní aplikace si bude moci nastavit výchozí parametry kvízu, jako je počet otázek, časový limit a uživatelské jméno, pod kterým bude vystupovat
- Uživatel mobilní aplikace si bude moci zobrazit nápovědu ke kvízu

Kapitola 3

Serverová část

3.1 Použité nástroje

3.1.1 Platforma

Mobilní aplikace bude ke svému chodu využívat tzv. *backend* server, který má za úkol nabízet předzpracovaná, bezpečně uložená a konzistentní data, týkající se požadované funkcionality. Platformou, která bude tuto roli zaujímat, je *Node.js*[4] - poměrně mladý nástroj, jehož kořeny sahají do roku 2009 a který využívá ke svému chodu programovací jazyk JavaScript.

Vzhledem ke zkušenostem autora práce by bylo možnou alternativou *PHP*[5], které se na podobný typ úlohy také hodí a zhostilo by se ho adekvátně[2]. `Node.js` se ale honosí následujícími výhodami, které z něj dělají ideálního adepta: využívá ke správě svých závislostí (tedy knihoven třetích stran, využívaných při vývoji) nástroj *npm*, který je dostupný spolu s instalací samotné platformy. Skrze něj je udržování aplikace snadné, rychlé, přehledné a efektivní, jelikož jsou všechny závislosti koncentrované na jednom místě, ze kterého jsou z kódu dostupné. Kromě toho se spolu s jádrem aplikace nemusí přenášet mezi servery, poněvadž jsou informace o knihovnách uložené v souboru `./package.json`, na základě kterého je následně `npm` stáhne z centrálního repozitáře[6] do adresářové struktury. V `PHP` plní podobnou úlohu *composer*, který z `npm` v mnohém čerpá, který ale nemá tak rozsáhlou uživatelskou komunitu a nabídku knihoven. Externí knihovny se nazývají *modules* a při jejich výběru je třeba brát v úvahu několik faktorů - jak jsou staré, zda se na nich vývojáři ještě aktivně podílí nebo je projekt ukončen a autoři nereagují na nové požadavky, jak velké množství požadovaných funkcí poskytují, rozsah uživatelské komunity a především - jestli je knihovna v aplikaci potřeba. Přílišná závislost na cizím kódu může snadno vést k tzv. *dependency hell*[3], kdy se aplikace

[4] `https://nodejs.org/en/`
[5] `https://secure.php.net/`
[6] `https://www.npmjs.com/`

stává zbytečně velkou, aniž by potřebovala všechny své `moduly`. Náhlá absence jednoho z nich (např. kvůli smazání autorem z centrálního repozitáře) poté ústí v nutnost její místo okamžitě zaplnit, aby se udržela funkčnost.

`Node.js` je dále ideální ke stavbě `API`, které lze vyrobit velice rychle a kvalitně za pomoci nevelkého objemu kódu. Stejně tak převyšuje `PHP` ve výrobě samostatných skriptů, které pracují nad databází a provádí dávkové operace. Mírně ale ztrácí v možnostech hostingu (míst, kde lze aplikaci spolu s databází spustit) a integrace s dalšími nástroji, jako jsou například databáze (v tomto `PHP` těží ze svého stáří a značné zaběhnutosti, jelikož má oproti `Node.js` o patnáct let více).

`Node.js` má další specifickou vlastnost, kterou je jednovláknovost. Jakmile je skript spuštěn, běží v jednom procesorovém vlákně, které sice zpracovává kód sekvenčně, ale snaží se valnou část operací řešit asynchronně - neblokujícím způsobem. V případě `API` serveru se tak veškeré závislosti zavedou pouze jednou a spuštěný proces je *event-driven*[4], což znamená, že je každý příchozí požadavek brán jako událost, která je obsloužena tzv. „handlery“ - funkcemi sloužícími jako obsluha. Není tak třeba vytvářet pro každý požadavek nový proces. Server permanentně běží a očekává vstupy.

Dalším faktorem je rychlost, ve které `Node.js` exceluje a vítězí nad konkurencí. Jeho jádro tvoří open-source JavaScriptový engine V8, který je vyvíjen projektem The Chromium Project[7] a sponzorovaný společností Google, který jej také využívá ve svém webovém prohlížeči Chrome. V8 používá tzv. *JIT* kompilaci, kdy JavaScript za běhu překládá přímo do strojového kódu. Mezitím probíhá dynamická optimalizace, která vede k vysoké rychlosti i na ní stavějícího `Node.js`.

3.1.2 Persistence dat

Co se týče výběru databáze, `Node.js` má vskutku široký záběr. Ohledně integrace do vývojového prostředí je většinou třeba pouze nalézt adekvátní *driver*, což je mezivrstva propojující platformu s databázovým serverem. K dispozici jsou `drivery` k databázím relačním (MySQL, PostreSQL, SQL Server), dokumentovým (MongoDB, CouchDB, ElasticSearch), key-value (Cassandra, Redis, LevelDB) i grafovým (Neo4j). Vzhledem k tomu, že je tato volba nezávislá jak na datové struktuře v Open datech, tak na uchovávání informací v mobilní aplikaci, která se bude na `backend` dotazovat, byla z čistě praktických důvodů zvolena `MongoDB`.[8]

Tento systém se vyznačuje enormní výhodou, kterou je vnitřní reprezentace dat prostřednictvím dokumentů založených na formátu `JSON` (resp. *BSON*, jeho binárně zakódovaná varianta ideální k vyhledávání a skladování dokumentů), díky čemuž je kompatibilní jak s `API` Open dat, tak s jazykem

[7] `https://www.chromium.org/`
[8] `https://www.mongodb.com/`

`Node.js` platformy - JavaScriptem, včetně `API` které bude `backend` server nabízet navenek. Odpadne tím tak kód a výpočetní čas nutný k tomu data mezi jednotlivými místy transformovat. Další výhodou je snadná škálovatelnost a replikovatelnost databáze, což se výrazně projeví při zvýšené zátěži. `MongoDB` se také prezentuje absencí schématu a z toho plynoucí volnosti při návrhu struktury. Data nejsou jasně definována, kolekce obsahují dokumenty s různými poli a samotná pole mohou obsahovat různé typy dat. To může na jednu stranu mást, ale při správném návrhu je tato funkcionalita velkou výhodou. V neposlední řadě nabízí `MongoDB` rychlé a efektivní *indexy* na všechny typy polí. Jedná se o struktury, které za cenu dalšího místa na disku poskytují výrazně pohotovější získávání dat, které se při vysokých nárocích na pohotovost odpovědi z `API` výrazně vyplatí. Kromě standardního typu `indexu`, který hodnoty v poli pouze seřadí podle daného klíče buď vzestupně nebo sestupně, existuje i speciální textový `index`. Ten je určen pro fulltextové vyhledávání, jež proti všem nadefinovaným polím porovnává dotazovaný řetězec a hledá alespoň částečnou shodu, tj. souhrnný text v polích musí onen řetězec obsahovat ale nemusí se mu rovnat. Takový `index` může existovat maximálně jeden na celou kolekci a jeho součástí je libovolný počet polí.

Do databáze se lze dotazovat za pomoci silného jazyka, založeného opět na `JSON` a honosícího se komplexitou rovnou *SQL*, dotazovacímu jazyku relačních databází. Pro složitější a rozsáhlejší požadavky je možné využívat samotný JavaScript, který je procesorem v `MongoDB` zpracováván a vyhodnocován. To lze aplikovat pro efektivní dotazování na velký objem dat skrze *MapReduce* model, určený k práci v paralelním, distribuovaném systému. Takové schopnosti stávající `backend` aplikace zatím nemá, ale v budoucnosti se na tuto možnost lze spolehnout.

Struktura `MongoDB` databáze je tvořena jednotlivými kolekcemi, které v sobě sdružují podobné dokumenty a svým účelem i využitím odpovídají tabulkám v relačních databázích. Ekvivalentem sloupců těchto tabulek jsou pole, reprezentované skrze atributy uložených dokumentů. Jedním ze základních znaků `MongoDB` je absence schématu, který by strukturu databáze jednoznačně definoval. Dokumenty v kolekci tak mohou mít různý počet polí, přičemž dvě identicky nazvané pole nemusí ani obsahovat hodnoty stejného datového typu. Díky této vlastnosti je databáze daleko flexibilnější a umožňuje tak větší volnost v návrhu i následné práci se svým obsahem.

Vyjma zmiňovaných výhod má `MongoDB` dlouho a pečlivě rozvíjený nativní `driver`[9], který nabízí téměř všechny možnosti, které v sobě databáze ukrývá. Tento `driver` se ukrývá v modulu *mongodb* a vyznačuje se kvalitní, rozsáhlou a detailní dokumentací. Jednou z možných alternativ je modul *mongoose*[10], který nativní `driver` dále rozšiřuje o možnost mapovat databázové dokumenty na modely a funguje tím pádem jako *ORM* vrstva. Tato funkcionalita ale není

[9] `https://mongodb.github.io/node-mongodb-native/`
[10] `http://mongoosejs.com/`

v aplikaci potřeba a byla tedy vynechána.

3.1.3 Aplikační rozhraní

3.1.3.1 Architektura

Aplikace bude fungovat jako webová služba, čili by měla odpovídat určitým standardům, aby byla její struktura schopná obsluhovat všechny potřebné požadavky, které se od ní očekávají. Proto byla její architektura založena na REST (Representational state transfer) přístupu[5], který slouží pro popis a stavbu podobných služeb. S jeho pomocí lze za pomoci uniformních a bezestavových operací přistupovat a manipulovat s daty, které se za rozhraním ukrývají. K těmto operacím jsou využívány standardní slovesa protokolu HTTP: GET (získání dat), POST (vytvoření entity), PUT (nahrazení nebo vytvoření entity), DELETE (mazání), PATCH (částečná modifikace entity), HEAD (vrácení hlavičky v odpovědi za absence jejího těla), OPTIONS (zjištění, jaká slovesa server poskytuje na dané `URL`), TRACE (sledování trasy požadavku) a CONNECT (vytvoření tunelu pro šifrovanou komunikaci).

Každý datový *zdroj* je definovaný svou `URL`, přes kterou je k němu možné přistupovat. Kolekce jsou reprezentované ve tvaru `www.zoo.cz/lexicon` a jejich jednotlivé elementy např. `www.zoo.cz/lexicon/zvire_18`. Další informace je možné v `URL` předávat díky *query* parametrům, které slouží ke specifikaci požadavku a nachází se v `URL` za otazníkem, tedy:
`www.zoo.cz/lexicon?trida=Savci`.

`RESTová` služba, jak se po správné implementaci této architektury nazývá, musí splňovat následující šestici požadavků:

- Klient-server model: obě části je potřeba oddělit, aby mohly fungovat nezávisle.
- Bezestavovost: Mezi jednotlivými požadavky se na serveru neukládá žádný stav, veškerý kontext si uchovává klient a každý požadavek tak obsahuje všechny potřebné informace pro jeho obsluhu.
- Cache: kdokoliv na cestě mezi klientem a serverem musí mít možnost odpověď uložit a předejít tak zbytečným požadavkům.
- Systém vrstev: klient nemůže na první pohled vědět kolik a jakých prostředníků se nachází na cestě od něj k serveru.
- Kód na vyžádání (nepovinné): server může klientovi předat spustitelný kód, aby tak přizpůsobil funkcionalitu.
- Uniformní rozhraní: základní prvek, oddělující a zjednodušující architekturu a sestávající z těchto částí:

- `Zdroje` musí být jednoznačně identifikovány (např. pomocí *URI*, jejíž je `URL` podtřídou) a jejich reprezentace se může lišit od serverové implementace.
- Reprezentace `zdroje` klientovi stačí k tomu, aby za její pomoci provedl se `zdrojem` jakoukoliv operaci.
- Každá zpráva předaná skrze rozhraní musí obsahovat informace, jak má být zpracována.
- *HATEOAS* (Hypermedia as the engine of application state) - po získání zprávy by z ní měl být klient schopen vyčíst, kam dále pokračovat (tento přístup reprezentuje provázanost operací).

3.1.3.2 Specifikace pro komunikaci

Formát dat, které má `API` přijímat i nabízet musí být co nejvíce upřesněn a pokrývat nejen aktuální požadavky, ale i možné budoucí. Měl by být snadno čitelný, pochopitelný a v neposlední řadě jednoduše zpracovatelný. Všechny tyto požadavky splňuje specifikace *JSON-API*.[11] Ta definuje jasná pravidla, jaká musí producent i konzument dat respektovat a splňovat, pokud chtějí komunikovat přes `API`, které je tímto standardem popsáno. Specifikace používá přehledné konvence[6], které vyhovují návrhu a je již natolik rozšířená, že ji implementuje množství knihoven pro různé jazyky jak na klientské tak na serverové straně, `Node.js` nevyjímaje.

`JSON-API` používá jakožto formát dat `JSON` a jejím *MIME typem* (identifikátorem pro přenos dat přes Internet) je `application/vnd.api+json`, což musí implementovat obě strany komunikující přes `API`. Tento typ se tedy musí vyskytovat v HTTP hlavičkách `Content-type` a `Accept`. Specifikace je schopná reprezentovat provázanost mezi jednotlivými `zdroji` dat, definovat příbuzná data, meta informace i chyby, které na serveru v rámci požadavku vznikly. V následující ukázce požadavku na server a odpovědi na ni lze popsat jednotlivé části, které `JSON-API` definuje.

Požadavek cílí na takové klasifikace, jejichž název zní „Savci“ a kritérium splňuje jedna třída, která je vrácena v těle odpovědi:

```
GET /classifications?name=Savci HTTP/1.1


HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "http://localhost:3000/classifications"
  },
  "meta": {
```

[11] `http://jsonapi.org/`

```
    "count": 1
  },
  "data": [
    {
      "type": "classifications",
      "id": "5920544433e2c025e665c54b",
      "attributes": {
        "opendata_id": "1",
        "type": "class",
        "parent_id": 0,
        "name": "Savci",
        "latin_name": "Mammalia",
        "slug": "savci",
        "orders": [
          {
            "_id": "5920544433e2c025e665c54c",
            "opendata_id": "2",
            "type": "order",
            "parent_id": "1",
            "name": "Malozubi",
            "latin_name": "Diprotodontia",
            "slug": "malozubi"
          },
          {
            "_id": "5920544433e2c025e665c554",
            "opendata_id": "10",
            "type": "order",
            "parent_id": "1",
            "name": "Sudokopytnici",
            "latin_name": "Artiodactyla",
            "slug": "sudokopytnici"
          }
        ]
      }
    }
  ]
}
```


Kořenový objekt obsahuje tři hlavní prvky: *links* s odkazy na další zdroje, *meta*, kde se nalézají meta informace buď o `zdroji` dat nebo o samotném požadavku a *data*, v němž je pole objektů splňujících dotaz. `data` dále zahrnují *type* (typ `zdroje`, v tomto případě se jedná o klasifikace), *id* (jednoznačný identifikátor prvku, zde databázové ID) a *attributes* (specifické vlastnosti `zdroje`) obsahující i pole „orders“, kde jsou objekty navázané na rodičovskou třídu, čili taxonomické řády.

3.1.3.3 Dokumentace

`API` je nutné přesně popsat a zdokumentovat, aby konzument dat neměl s jejich získáváním ani zpracováním problém. K tomuto účelu je ideální webový

nástroj *Apiary*.[12] Tato platforma je de facto standardem pro návrh, prototypování, dokumentaci a testování aplikačních rozhraní. Její jádro tvoří jazyk *API Blueprint* postavený na standardu *MSON*[13], s pomocí kterého je rozhraní navrhováno. Jedná se o způsob popisu struktur takovou syntaxí, která je čitelná jak pro člověka tak pro stroj. S jeho pomocí lze definovat `endpoint URL` včetně všech jejích náležitostí, jako jsou povolené `query` parametry se svými hodnotami a příklady, popis i ukázkové verze požadavků a odpovědí ze serveru. Prakticky navržený webový editor umožňuje dokumentaci v reálném čase generovat a kontrolovat, k čemuž nabízí i syntaktickou validaci textu.

Po vytvoření získává dokumentace vlastní subdoménu, ze které je dostupná veřejnosti. Její návštěvnost je možné sledovat prostřednictvím pokročilého monitoringu.

3.2 Návrh

3.2.1 Aplikační vrstva

3.2.1.1 Promises

Rozsah aplikace vyžaduje zvolit některé zásadní programátorské techniky co nejdříve, aby v průběhu vývoje nedošlo ke zbytečným prostojům z důvodu špatného návrhu. Následná refaktorizace, úpravy kódu a přizpůsobování se aktuálně vzniklým požadavkům je zdlouhavý proces, kterému lze mnohdy snadno předejít. Jednou z moderních technik, kterou bylo možno do vývoje zanést, jsou *Promises*.[14] Tento přístup prostupuje většinu aplikace a byl proto zásadním rozhodnutím v rámci návrhu. K jeho odůvodnění je potřeba definovat několik vlastností vybraného programovacího jazyka a platformy.

Vzhledem k základním stavebním blokům, na kterých stojí `Node.js`, jimiž jsou jednovláknovost a z ní plynoucí nutnost asynchronicity, poskytuje vybraná platforma mechanismus, díky kterému se ony bloky daří efektivně využívat. Jedná se o *callback* funkce. Ty představují nedílnou součást samotného JavaScriptu a tedy i valné většiny jazyků a knihoven, které jsou na něm postavené nebo jsou do něj kompilovány. Stejně tak se tyto funkce vyskytují v množství typově velice odlišných jazyků, jako je Java, C, C#, Python nebo `PHP`.

Princip callbacku spočívá v možnosti předat funkci jakožto argument do jiné části kódu, která je následně schopna onu funkci zavolat (ať už hned nebo v závislosti na dalších faktorech kdykoliv později, což splňuje asynchronní návrh). `Callback` se tak prakticky chová (v závislosti na použitém jazyce) jako proměnná. Samotný JavaScript je na toto ideálně připraven, neboť v jeho implementaci se funkce považují za objekty (dokonce tzv. *first-class objects*, tedy

[12] `https://apiary.io/`
[13] `https://github.com/apiaryio/mson`
[14] `https://www.promisejs.org/`

objekty první třídy, které mohou obsahovat vlastní proměnné i metody) a lze je posílat i vracet z jiných funkcí. Této vlastnosti je využíváno spolu s *anonymními funkcemi* (jenž nejsou přiřazeny do žádné proměnné, nemají název a jsou vytvářeny až za běhu programu) k řízení posloupnosti a návaznosti vykonávání jednotlivých úloh[7].

Příkladem budiž následující jednoduchý kus skriptu, jenž nejprve vytvoří instanci modulu `express`[15], na níž následně zavolá metodu „listen()“, které předá dva argumenty - prvním je port, na kterém má čerstvě vzniklý server poslouchat, a druhým anonymní `callback` funkce, která je aplikací zavolána až v momentě, kdy je jistota, že server skutečně naslouchá (přičemž tuto informaci pouze vypíše do konzole):

```
const app = require('express')();

app.listen(3000, function() {
    console.log('Listening for incoming requests on port 3000...');
});
```

Tento mechanismus má ale i stinnou stránku, kterou je tzv. *callback hell* („peklo tvořené callback funkcemi“). Tohoto stavu lze dosáhnout v aplikaci, kde je nutné řešit hodně vzájemných závislostí, které by se v případě vícevláknového běhu programu nechaly provádět synchronně (tedy nejprve se provede kus kódu a až poté se vykoná další část, která na něm závisí) tak, že se rozmístí do různých procesorových vláken, aby se vzájemně neblokovaly. `Callback hell` nastává v momentě, kdy je množství asynchronních `callback` volání natolik vysoké a zanořené do tak enormní hloubky, že je výsledný kód nepřehledný, velice špatně udržitelný a ještě hůře rozšiřitelný.[8]

Ukázkou prvopočátku tohoto stavu je dotaz serverové aplikace na vrácení jednoho konkrétního dokumentu, identifikovaného svým ID, k čemuž je potřeba následující kombinace volání (jednotlivé vstupní argumenty jsou pro jednoduchost globální a kód je pseudo nadstavbou nad komunikací přes nativní `MongoDB` knihovnu):

```
connectDB(config, function(err, db) {
    if (err) response(err);
    else db.getCollection(collectionName, function(err, collection) {
        if (err) response(err);
        else collection.findDocument(documentId, function(err, doc) {
            if (err) response(err);
            else {
                doc.parseToJSON(function(err, parsedDocument) {
                    if (err) response(err);
                    else response(null, parsedDocument);
                });
            }
        });
    });
});
```

Posloupnost funkcí vedoucí k žádanému výsledku je takováto:

[15] `https://expressjs.com/`

1. Připojit se k databázi
2. Nalézt v ní potřebnou kolekci dokumentů
3. Z kolekce získat správný dokument
4. Převést dokument do požadovaného formátu
5. Výsledek vrátit v rámci odpovědi

Celou dobu je potřeba kontrolovat výsledky jednotlivých operací, zda při nich nenastala chyba a až v případě jejich úspěchu lze pokračovat s další fází. Je zřejmé, že se stoupajícími úrovněmi zanořování kód připomíná čím dál více rostoucí pyramidu, ve které není okamžitě viditelné, co se kde nachází. Prevence takového stavu zahrnuje především snahu o střídmější strukturování kódu, vynechávání `anonymních funkcí` kdekoliv nejsou potřeba a nahrazovat je funkcemi pojmenovanými, dělení logicky souvisejících celků do modulů a z toho plynoucí izolace, pokusy o psaní obecných a znovupoužitelných funkcí. Existují ovšem i pokročilejší techniky, které tyto postupy v žádném případě nevylučují ani nezbavují programátora zodpovědnosti za kvalitu jím psaného kódu, ale také vedou k výrazně lepší čitelnosti a přehlednosti. Jednou takovou technikou jsou takzvané `Promises`, oficiálně uvedené v roce 2015 v rámci specifikace ECMAScript 6.[16]

`Promise` („příslib“) je proxy objekt, stojící na místě hodnoty, která může být k dispozici buď okamžitě, někdy v budoucnu nebo vůbec. Namísto samotné hodnoty tedy funkce vrací příslib, že hodnotu dodá nebo zamítne (v takovém případě oznámí, co k zamítnutí vedlo). S pomocí tohoto mechanismu lze s asynchronním kódem zacházet takřka jako se synchronním. Jedna funkce něco slíbí, druhá čeká na výsledek jejího slibu zatímco sama něco slíbí třetí funkci atd.[9] Předchozí příklad bude po refaktorizaci s využitím `Promises` vypadat následovně:

```
connectDB(config).then((db) => {
    return db.getCollection(collectionName);
}).then((collection) => {
    return collection.findDocument(documentId);
}).then((document) => {
    return document.parseToJSON();
}).then((parsedDocument) => {
    response(null, parsedDocument);
}).catch((err) => {
    response(err);
});
```

V ukázce jsou v rámci dalších úprav kódu použity JavaScriptové *arrow funkce*, jež jsou ve tvaru `(argumenty) => { tělo funkce }` a zachovávají

[16] `http://es6-features.org/#Constants`

vnější kontext. Je zde také vidět schopnost zachytávat chyby ve všech navazujících `Promises` až na konci řetězu za předpokladu, že s nimi se všemi má být nakládáno stejným způsobem - v tomto případě vrátit chybu prostřednictvím odpovědi. Posloupnost volání tak aktuálně simuluje standardní *try/catch* blok a je díky tomu daleko přehlednější.

Tato technika „příslibů“ je ve světě ECMAScriptu hojně propagována a je jí predikována ještě zářnější budoucnost[10], což je spolu s popisovanými vlastnostmi hlavní důvod, proč bylo rozhodnuto ji extenzivně využívat napříč `backend` aplikací. Již existuje i etablovaná knihovna pro `Node.js`, která je na `Promises` postavená a využívá je navýsost efektivně, s názvem *bluebird*.[17] I přes svou rychlost, zjednodušení mnoha operací a široké nabídky nových funkcí je to ale další závislost a stavební prvek navíc, kvůli čemuž byla vynechána a nahrazena čistou implementací.

Dalším výrazným rozhodovacím prvkem pro volbu `Promises` byl fakt, že je nativní `MongoDB` knihovna podporuje a nabízí ve svých metodách jako alternativu právě ke klasickým `callback` funkcím. Tím pádem je na nich postavená už ta nejnižší potřebná vrstva, což umožňuje na ní stavět bez použití vlastnoručního zaobalování databázových metod do `Promises` (nebo používání `bluebird` modulu k témuž). Tato skutečnost usnadnila i zavržení další techniky, nahrazující `callback` funkce - *Async/Await*. Toto byla dříve externí knihovna, nedávno přidaná přímo do enginu V8, který pohání i `Node.js` a je v něm tedy nyní dostupná přímo, bez vnější závislosti. Tato funkcionalita ale opět staví na `Promises` a ačkoliv je zase o krok dále, vyplatí se spíše pro velké projekty[11].

3.2.1.2 Module a prototype

Při implementaci bylo dbáno na to, aby byl výsledný kód logicky strukturovaný a jednotlivé části fungovaly jako samostatné jednotky se schopností být použité na více místech. Toho bylo dosaženo cíleným využíváním konceptu `Node.js modulů`, do kterých byly uzavírány veškeré kusy aplikace, které takto řešit šly. Každý `modul` je izolovaný od okolního světa s tím, že veškeré externí závislosti jsou mu předány z volajících skriptů - procesem nazvaným *dependency injection*[12]. Každý `modul` se nachází ve vlastním souboru a co z něj vrací je dáno globálním objektem *exports*, který je vždy přítomen prostřednictvím samotného `Node.js`. Jeho inicializaci si lze zjednodušeně představit takto: `var exports = module.exports = {};`, přičemž se do něj dále vkládají další exportované objekty dle předpokládaného záměru. `module.exports` je poté vrácen při zavolání `modulu` pomocí `require("cesta_k_modulu")` - zde je třeba dávat pozor, který ze zmíněných objektů je rozšiřován, jelikož `exports` je pouze reference a pokud bude tato proměnná nastavena na instanci nového objektu, ztratí se vazba na `module.exports`, které tím pádem zůstane ve svém původním stavu.

[17] `http://bluebirdjs.com/docs/getting-started.html`

Každý vytvořený modul využívá prototypovou povahu JavaScriptu. Tato varianta *OOP* (objektového programování) umožňuje řešit dědičnost mezi objekty delegací skrze *prototypy*. Ty si lze představit jako rodičovské třídy, ze kterých podtřídy dědí všechny jejich vlastnosti, což je analogické k třídně orientovanému `OOP`. Každý objekt má svůj `prototyp`, který je z něj přístupný přes vlastní atribut. Podle toho, zda je objekt inicializován pomocí literálu (`var object = {};`) nebo skrze konstruktor (`var object = new Class();`), má svůj `prototype` nastaven buď na `Object.prototype` nebo na `Class.prototype`. Jakékoliv změny se následně provádí na `prototypu` jsou promítané i do všech objektů, které z něj dědí.

Vlastní `moduly` jsou tedy vytvářeny jako konstruktory, které přijímají své závislosti skrze vstupní parametry. Tyto závislosti spolu s dalšími lokálními proměnnými jsou následně dostupné přes referenci `this`, ukazující na konkrétní instanci. `Prototyp` těchto konstruktorů je rozšiřován o další funkce, které jsou poté vně `modulu` přístupné stejně jako metody objektů v třídně orientovanému `OOP`.

3.2.1.3 Struktura

Serverovou část lze rozdělit do tří hlavních celků, z nichž každý obstarává jednu oblast:

1. Import z Open dat - základní prvek, který připraví databázi do výchozího stavu.

2. Generování kvízových otázek - pro fungování serveru ho není třeba, ale kvíz v mobilní aplikaci se bez něj neobejde. Potřebuje mít k dispozici naimportovaný lexikon zvířat.

3. Server - bez naimportovaných dat nemá co nabízet, ale fungovat může.

Každý z prvků bude fungovat zvlášť, izolovaný od ostatních. Vzájemně provázány jsou jen svými výstupy. `Moduly`, které jsou potřebné pro jejich správnou funkčnost, si inicializuje, spravuje a nastavuje každý sám. Externí `moduly` se nachází v adresáři `./modules/`, aplikační závislosti se ukládají do `./node_modules/` a server si své části udržuje v `./routes/`.

Při vývoji je využíván verzovací systém *GIT*[18], díky kterému lze sledovat jednotlivé změny v kódu, zachovávat jejich historii a lépe se v nich orientovat. `GIT` při sledování repozitáře ignoruje adresář s aplikačními závislostmi prostřednictvím souboru `./.gitignore`.

[18] `https://git-scm.com/`

3.2.2 Datová vrstva

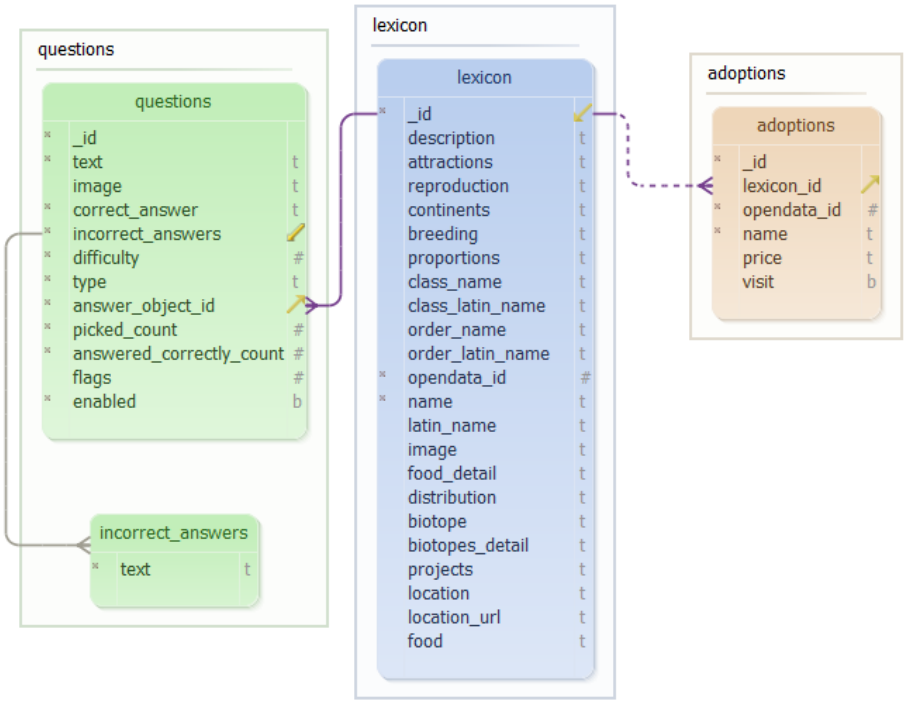
Struktura databáze byla sestavována s cílem převést z Open dat všechny nezbytné informace a neimportovat ty, které by byly zbytečné a nepotřebné. Vytvořené kolekce buď odpovídají originální struktuře zdrojových dat, nebo v případě nezávislosti na Open datech dodržují návrh.

Každému nově vzniklému dokumentu je při vkládání do databáze automaticky vygenerován unikátní řetězec, sloužící jako jeho jednoznačný identifikátor. Tato hodnota odpovídá `MongoDB` datovému typu *ObjectId*, ukládá se do pole s názvem *_id* a tvoří ji dvanáct bajtů s následujícím významem:

- 4 bajty na počet sekund uplynulých od tzv. *Unix epochy*[19], což je pojem reprezentující čas 00:00:00 UTC dne 1.1.1970
- 3 bajty pro identifikaci stroje, kde byl řetězec vygenerován
- 2 bajty pro ID procesu, který řetězec vygeneroval
- 3 bajty na hodnotu interního počítadla, startující z náhodného místa

Konečný návrh kolekcí je předveden na dvou databázových diagramech. Ty dělí kolekce do dvou množin - první tvoří Lexikon zvířat a kolekce na něm závislé (3.1), druhá obsahuje zbytek (3.2). Po každém diagramu následuje tabulka s popisem jednotlivých polí, které se v kolekcích nachází (vynechány jsou identifikátory dokumentů, jež plní všude stejnou úlohu). Pořadí polí je zachováno tak, jak bylo sestaveno při generování dat prostřednictvím `MongoDB`.

[19] `http://www.unixtimestamp.com/`



Obrázek 3.1: Schéma databázových kolekcí závislých na lexikonu zvířat

Název pole	**Popis**
text	Znění otázky
image	URL obrázku, který je součástí otázky
correct_answers	Text správné odpovědi
incorrect_answers	Pole s texty nesprávných odpovědí
difficulty	Obtížnost otázky
type	Typ otázky
answer_object_id	ID zvířete z lexikonu, kterého se otázka týká
picked_count	Kolikrát byla otázka vybrána
answered_correctly_count	Kolikrát byla otázka správně zodpovězena
flags	Kolikrát byla otázka označena za chybnou
enabled	Příznak, zda je otázka aktivní a lze ji vybrat

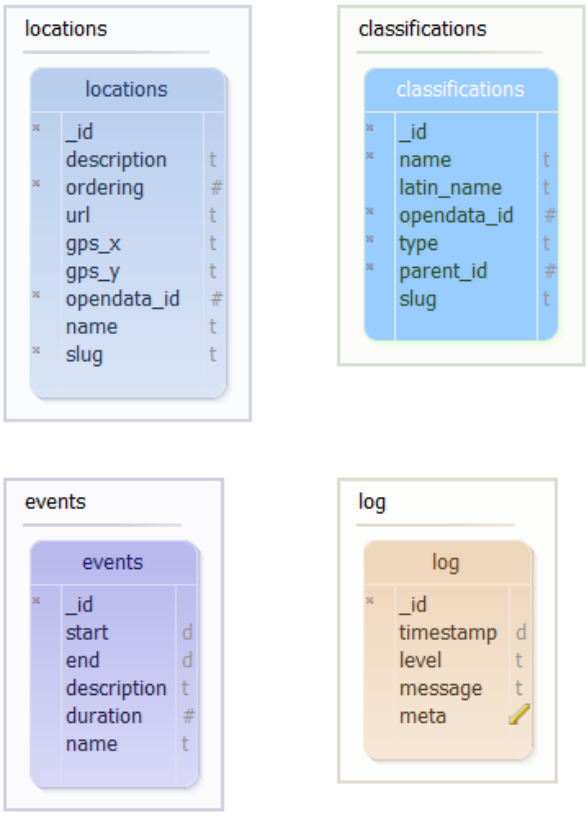
Tabulka 3.1: Popis polí v kolekci *questions* (Kvízové otázky)

Název pole	Popis
lexicon_id	ID zvířete v Lexikonu, pokud takové existuje
opendata_id	ID adopce v Open datech
name	Název zvířete
price	Cena adopce zvířete
visit	Příznak, zda je možné zvíře navštěvovat

Tabulka 3.2: Popis polí v kolekci *adoptions* (Seznam zvířat k adopci)

Název pole	Popis
description	Detailní popis zvířete
attractions	Zajímavosti
reproduction	Reprodukce, délka březosti, počet mláďat apod.
continets	Kontinenty, kde se zvíře vyskytuje
breeding	Chov v ZOO Praha
proportions	Rozměry zvířete
class_name	Název třídy, do které zvíře spadá
class_latin_name	Latinský název třídy, do které zvíře spadá
order_name	Název řádu, do kterého zvíře spadá
order_latin_name	Latinský název řádu, do kterého zvíře spadá
opendata_id	ID zvířete, pod kterým je uloženo v Open datech
name	Název zvířete
latin_name	Latinský název zvířete
image	`URL` obrázku zvířete
food_detail	Detailní popis potravy zvířete
distribution	Detailní popis rozšíření zvířete ve světě
biotope	Biotopy, ve kterých se zvíře vyskytuje
biotopes_detail	Detailní popis biotopů, ve kterých se zvíře vyskytuje
projects	Projekty a programy na ochranu zvířete
location	Umístění zvířete v ZOO Praha
location_url	`URL` umístění zvířete na webu ZOO Praha
food	Základní popis potravy zvířete

Tabulka 3.3: Popis polí v kolekci *lexicon* (Lexikon zvířat)



Obrázek 3.2: Schéma databázových kolekcí nezávislých na lexikonu zvířat

Název pole	**Popis**
description	Detailní popis lokality
ordering	Pořadí v rámci ZOO Praha
url	`URL` lokality na webu ZOO Praha
gps_x	GPS souřadnice reprezentující zeměpisnou šířku lokality
gps_y	GPS souřadnice reprezentující zeměpisnou délku lokality
opendata_id	ID lokality v Open datech
name	Jméno lokality
slug	Jméno lokality transformované pro snadné využití v `URL`

Tabulka 3.4: Popis polí v kolekci *locations* (Lokality v ZOO Praha)

Název pole	Popis
name	Český název klasifikace
latin_name	Latinský název klasifikace
opendata_id	ID klasifikace v Open datech
type	Typ klasifikace (třída / řád / čeleď)
parent_id	Reference na rodičovskou klasifikaci ve formě jejího `opendata_id`
slug	Jméno klasifikace transformované pro snadné využití v `URL`

Tabulka 3.5: Popis polí v kolekci *classifications* (Klasifikace zvířat)

Název pole	Popis
start	Datum začátku události ve formátu ISO 8601[20]
end	Datum konce události ve formátu ISO 8601
description	Detailní popis události
duration	Délka trvání události v minutách
name	Jméno události

Tabulka 3.6: Popis polí v kolekci *events* (Události v ZOO Praha)

Název pole	Popis
timestamp	Zakódovaný čas a datum vzniku dokumentu, využívající `Unix timestamp`
level	Úroveň zprávy (např. „error“)
message	Text zprávy
meta	`Meta` informace o zprávě

Tabulka 3.7: Popis polí v kolekci *log* (Logované zprávy z aplikace)

3.2.3 Popis aplikačního rozhraní

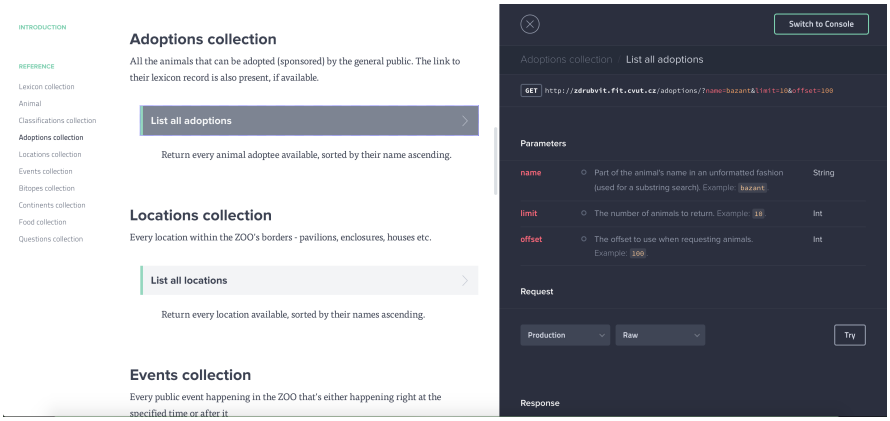
Dostupné `zdroje` přesně reflektují stanovené požadavky. V následující tabulce jsou všechny definovány spolu se seznamem `query` parametrů, které je možné k požadavku přidat. Cesty jsou relativní a předpokládají, že rozšiřují jeden výchozí `endpoint`. Ke každé z nich se přistupuje s využitím `HTTP` metody `GET`.

Název `zdroje`	**Cesta ke `zdroji`**	**`Query` parametry**
Kolekce Lexikon zvířat	/lexicon/	biotope, class_name, continents, description, distribution, food, limit, location, name, order_name, offset
Zvíře z kolekce Lexikon	/lexicon/id/	-
Kolekce Klasifikace zvířat	/classifications/	class, order, family
Kolekce Zvířata k adopci	/adoptions/	name, limit, offset
Kolekce Lokality v ZOO	/locations/	-
Kolekce Události v ZOO	/events/	datetime
Kolekce Biotopy zvířat	/biotopes/	-
Kolekce Kontinenty	/continents/	-
Kolekce Potrava zvířat	/food/	-
Kolekce Kvízové otázky	/questions/	limit

Tabulka 3.8: Návrh zdrojů v `API`

V `Apiary` se nachází všechny kolekce a dotazy, které je možné na příslušný `endpoint` posílat. Výčtem jsou při každé z nich zobrazeny i jejich `query` parametry, u nichž je uveden datový typ (zda se jedná o řetězec, číslo nebo pravdivostní hodnotu), příznak jestli jsou povinné a popis spolu s ukázkovou hodnotou. Dále je u každé poskytnuta vzorová odpověď serveru s `HTTP` kódem `200 - OK` a v rámci kolekce `lexicon` jsou prezentovány i chybové zprávy. Všechny požadavky je možné okamžitě z webového rozhraní testovat proti produkčnímu serveru. Obrázek 3.3 ukazuje editor pro tvorbu dokumentace a snímek 3.4 poté její finální prezentaci, dostupnou z `http://docs.zoologicka.apiary.io/`.

Obrázek 3.3: Editor pro tvorbu dokumentace `API` v rozhraní `Apiary`



Obrázek 3.4: Dokumentace `API` vygenerovaná za pomoci `Apiary`

3.3 Implementace

3.3.1 Komunikace s databází

`Backend` se s databází dorozumívá prostřednictvím klienta v podobě třídy *MongoClient*, nalézající se v modulu `mongodb`. Tato třída obsahuje statickou metodu, díky které se s pomocí `URI` řetězce ve tvaru

`mongodb://[user:pass@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]`

k databázi připojí a vrací `Promise`, který v případě úspěchu poskytne objekt s instancí databáze. Řetězec obsahuje přihlašovací údaje (jméno a heslo), adresu serveru a portu, kde `MongoDB` běží, hosty, název databáze a případné další možnosti.

Nad `MongoDB` byla postavena ještě jedna vrstva, která má za úkol slučovat jednotlivé požadavky do větších celků, které se poté dají volat prostřednictvím

jediné metody. To výrazně snižuje množství opakujícího se kódu a zaobaluje požadavky aplikace do smysluplných funkcí, jež odstiňují zbytečné parametry spodní databázové vrstvy, nabízejí navenek pouze nezbytné argumenty a přidávají další vnitřní funkcionalitu navíc. Většina požadavků například vyžaduje nejprve najít správnou kolekci a nad ní poté provádět další operace. Kolekce nemusí existovat a volající si může přát ji vytvořit nebo při vyhledávání může dojít k chybě. Celý tento proces je schován v jedné metodě, na jejíž výsledek lze libovolně reagovat.

Tato vrstva je implementována ve třídě *CollectionDriver* nalézající se v souboru `./modules/collection-driver.js` a je vyžadována ve většině `backend` aplikace, jelikož je práce s databází všudypřítomná. Využívá `Promises` k řízení správné posloupnosti operací, přehlednosti kódu, odchytávání chyb a kompatibilitě s `MongoDB` vrstvou. Každá z funkcí sama vrací `Promise`, který musí volající strana posléze reflektovat. Přehled metod, které *CollectionDriver* nabízí, je včetně zaobalených metod v tabulce 3.9.

Název metody	**Popis**	**Zaobalované metody**
getCollection	Vrací kolekci a v případě její neexistence ji vytvoří	collection, createCollection
findDocument	Vrací jeden dokument, odpovídající filtru	collection, findOne
findAllDocuments	Vrací všechny dokumenty, odpovídající filtru, přičemz může výsledek omezovat a řadit	collection, find, limit, skip, sort, toArray
findDistinctValues	Vrací seznam originálních hodnot pole v kolekci	collection, distinct
getRandomDocuments	Vrací definovaný počet náhodně vybraných dokumentů	collection, aggregate, toArray
countDocuments	Vrací počet dokumentů odpovídajících filtru	collection, count
insertDocuments	Hromadně vloží dokumenty do kolekce	collection, createCollection, insertMany
truncateCollection	Smaže všechny dokumenty z kolekce	collection, createCollection, remove
renameFields	Hromadnou operací přejmenuje definované pole v kolekci	collection, updateMany
createIndexes	Hromadnou operací vytvoří `indexy` na definovaných polích	collection, createIndexes
closeDB	Zavře spojení s databází	close

Tabulka 3.9: Metody databázové mezivrstvy

3.3.2 Konfigurační proměnné

Veškeré proměnné i se svými hodnotami, jež se využívají napříč všemi přítomnými skripty, jsou dostupné z jednoho výchozího bodu, jímž je soubor `./config.js`. Chová se jako `modul`, který exportuje jediný objekt - *config*. Do tohoto objektu jsou v průběhu skriptu navkládány dílčí podobjekty jakožto jeho vlastnosti, přičemž každý z nich reprezentuje jednu specifickou oblast, ve které se využívá. Ony podřazené úrovně mohou obsahovat další objekty, pole hodnot nebo pole objektů, v závislosti na potřebách aplikace. V první úrovni se nachází následující kořenové objekty:

- *mongodb* - parametry připojení k lokální instanci databázového serveru a názvy kolekcí, jež se v databázi nachází

- *opendata* - definice `endpointu` Open dat a identifikátory datových `zdrojů`
- *zoo* - vlastnosti domény, jako je webová adresa ZOO, využívaná jako základ pro tvorbu `URL`
- *questionTypes* - typy kvízových otázek, jak mají znít a ze kterých polí je třeba sbírat jejich odpovědi
- *filterColumns* - výčet názvů sloupců, které budou vybrány při importu dat
- *transformMethod* - názvy metod využívaných pro transformace příchozích dat
- *fieldMapping* - dvojice hodnot, specifikující přejmenování sloupců z Open dat na pole v lokálních kolekcích
- *fieldIndexes* - jména databázových polí spolu s typy indexů, jež se na nich budou tvořit
- *serialization* - která pole dokumentů budou serializována při komunikaci s `API`
- *allowedApiQuery* - seznamy povolených hodnot v `query řetězci` při dotazování na `API`

Některé z hodnot (např, *filterColumns*) jsou povinné, jelikož se používají na kritických místech, jiné (např. *allowedApiQuery*) jsou volitelné a zadávají se až v případě potřeby. Kořenové objekty jsou do onoho exportovaného vkládány postupně, jeden po druhém. Nejdříve je vždy zadefinován název atributu, který je inicializován jakožto prázdný objekt. Teprve poté jsou nastavovány vlastnosti tohoto atributu. Je to z toho důvodu, aby byl výsledný soubor lépe čitelný a jednotlivé části viditelně oddělené, načež lze navíc ke každé z nich vložit komentář s popisem jejího obsahu a účelu. Modul s konfigurací je následně načítán kdykoliv je potřeba (což je valné většině samostatných skriptů). Jedna hodnota může nabýt takovéhoto tvaru:

```
// Info regarding the ZOO's own domain
config.zoo = {};
config.zoo.host = "https://zoopraha.cz/";
```

3.3.3 Logování

Potřeba zaznamenávat si průběh a chování jednotlivých skriptů je nedílnou součástí korektně fungující aplikace. V `Node.js` platí za jeden z nejspolehlivějších nástrojů *winston*[21]. Tento modul je asynchronní (neblokuje vykonávání kódu), má širokou uživatelskou základnu (`npm` hlásí desítky tisíc stažení

[21] `https://www.npmjs.com/package/winston`

denně), `github` repozitář[22] poskytuje rozsáhlou dokumentaci a příklady využití, jež jsou snadno pochopitelné a implementovatelné. Jedinou nevýhodou může být pomalá reakce vývojářů na existující a vznikající problémy nebo chyby, což ovšem odpadá v případě, že modul splňuje požadavky aplikace. Alternativou může být *Bristol*[23], který se pyšní vysokou měrou nastavení, lepším oddělením zpráv od jejich destinací a tím, že oproti konkurenci není natolik robustní. `winston` ovšem zvítězil díky tomu, že nabízí vše potřebné, což kombinuje s rychlým a přívětivým zavedením do aplikace.

V jeho středu se nachází tzv. *transports*. Jedná se o způsoby logování, reprezentující kam a jak se mají zprávy ukládat. Využívány jsou dva - výpis do konzole a uchování v databázi, kde budou logy persistentní a snadno dohledatelné i agregovatelné. `Transport` pro konzoli je již ve výchozím nastavení obsažen, zprávy pro ní jsou navíc formátovány pomocí modulu *colors*[24], jež umožňuje jejich části obarvovat a tím pádem zvýrazňovat oproti zbytku. Toho je využíváno např. u názvů kolekcí, výsledků operací nebo chybových hlášení. Druhým `transportem` je ten databázový, zprostředkovaný skrze modul *winston-mongodb*[25], jež pochází od těch samých vývojářů, kteří mají na svědomí samotný `winston`. Tomuto modulu stačí pouze předat instanci běžící databáze a nastavit odstraňování již zmiňovaných barev z textu, zbytek zařídí sám.

Typ logované zprávy rozlišuje její závažnost. Pro `winston` je využíván standardní systém, který implementuje i `Node.js` a který vypadá následovně (seřazeno sestupně od úrovně s nejvyšší prioritou včetně svého číselného kódu):

1. error (0) - Chybová hlášení
2. warn (1) - Varování
3. info (2) - Zprávy pouze informačního charakteru
4. verbose (3) - Rozsáhlý výpis
5. debug (4) - Zprávy potřebné při vývoji, jako je aktuální stav proměnné apod.
6. silly (5) - Logování téměř všeho, co lze

Potřebná úroveň se instanci `winston` předá buď při zavádění `transportu` nebo dynamicky za chodu aplikace. Využita byla první možnost s tím, že se výchozí úroveň nikde nemění - konzole má nastaven `debug`, zatímco databáze `info`. Znamená to do jaké maximální úrovně se logování provede. V databázi se tak např. nenachází zprávy s hodnotou `verbose` a výše.

[22] `https://github.com/winstonjs/winston`
[23] `https://github.com/TomFrost/Bristol`
[24] `https://www.npmjs.com/package/colors`
[25] `https://github.com/winstonjs/winston-mongodb`

Samotný `winston` je v aplikační struktuře zaobalen třídou *Logger*, nacházející se ve vlastním modulu v souboru `./modules/logger.js`. Tato třída kromě inicializace poskytuje metodu, která zařídí zobrazení i uložení příchozí zprávy na základě její nadefinované úrovně. Instance `Loggeru` je poté předávána na všechna potřebná místa.

3.3.4 Import dat

Import z Open dat je realizován prostřednictvím skriptu `./import.js`. Ten ke svému běhu potřebuje všechny výše zmiňované moduly - je pro něj nutné komunikovat s databází skrze `CollectionDriver`, extenzivně logovat jednotlivé kroky svého procesu pomocí `Logger` a sbírat proměnné z konfiguračního `config` objektu. K tomu navíc využívá dva další: `Transformer`, nutný pro editace příchozích dat a především `Importer`, jež zařizuje jejich samotné zpracovávání.

Hlavní importní skript se tak pouze připojí k databázi, vytvoří instance jednotlivých modulů, sestaví základní `URL` endpointu v Open datech a následně už jen koordinuje chod importu. K tomu využívá nativní metodu, kterou nabízí implementace `Promise`, nazývající se *Promise.all()*. Tato funkce očekává jako parametr pole `Promise` instancí, načež sama vrací `Promise` navázaný na jejich splnění. Po úspěšném návratu poslední z nich nebo v případě první chyby se výsledek zaloguje na příslušné úrovni a následně se ukončí připojení k databázi. Splnění příslibu v *Promise.all()* znamená získání návratových hodnot jednotlivých dílčích příslibů, včetně zachování pořadí, ve kterém se do původního pole vkládaly.

Třída `Importer` se nachází v souboru `./modules/importer.js` a nabízí veškerou komunikaci s Open daty včetně transformací získavaných dat. Průběh zpracování každého příchozího datového zdroje lze popsat pomocí následující *pipeline* - posloupnosti operací, jež jsou na onen zdroj postupně aplikovány, jedna po druhé. V případě chyby v jedné z nich celý proces skončí, jelikož by data v ten moment byla nekonzistentní.

1. Stažení a zparsování dat
2. Vymazání příslušné kolekce v lokální databázi
3. Transformace dat
4. Vložení hotových dat do databáze
5. Přejmenování polí v kolekci tak, aby odpovídala návrhu
6. Vytvoření databázových `indexů` nad patřičnými poli

Tato `pipeline` je realizována díky metodě *importWrapper(resource)*, která náleží třídě `Importer` a je opakovaně volána hlavním importním skriptem pro

různé typy zdrojů dat. Vstupní argument je zde název konkrétního zdroje, na jehož základě jsou z `config` objektu získány veškeré konfigurační proměnné, jež jsou během procesu potřeba. Metoda onu `pipeline` implementuje za pomoci na sebe sekvenčně navázaných `Promises`.

Prvním krokem je stahování dat, ke kterému se využívá `Node.js` modul *http*[26], sloužící k síťovým operacím na aplikační vrstvě. Funkce, která tuto operaci zaštiťuje, očekává `endpoint`, ke kterému se bude připojovat a seznam polí, které ze vzdáleného zdroje vyselektuje. Seznam k dodané URL připojí ve formě `query řetězce`, k němuž přidá i parametr pro limit počtu vracených dokumentů (na vzdáleném serveru je ve výchozím nastavení tento limit omezen na sto, což je při takovéto jednorázové hromadné operaci, jakou je import, zbytečně málo), aby byla vrácena všechna dostupná data. Oněch dat zdaleka není tolik, aby si s nimi importní skript nedovedl poradit se všemi najednou. Získávaná data jsou rozdělena do jednotlivých kusů pevné velikosti, což je typ přenosu skrze protokol `HTTP` definovaný ve verzi 1.1. Tyto kusy se postupně ukládají do mezipaměti až do uzavření spojení se serverem. Výsledek je následně zparsován z `JSON` do standardního JavaScriptového objektu, který je předán do další části `pipeline`.

Následuje vymazání obsahu kolekce, po kterém je databáze připravena nová data přijmout. Ještě předtím je ovšem nad nimi potřeba provést takové transformace, které je dovedou do korektního stavu. Takové operace nabízí skrze své metody třída `Transformer` ze souboru `./modules/transformer.js`. Pro každý typ dokumentu, který je třeba upravit, je zde samostatná funkce, která si vždy jeden dokument bere jako vstupní parametr a využívá faktu, že JavaScript předává proměnné hodnotou, což se týká i referencí na objekty. Instance dokumentu tedy existuje stále pouze jedna a `Transformer` dělá změny přímo na ní. Jednotlivé dokumenty jsou upravovány asynchronně a je tedy třeba udržovat povědomí o tom, kolik z nich je hotových. To je řešeno počítadlem, jež je inicializováno nulovou hodnotou a s každým zpracovaným dokumentem se inkrementuje. Jakmile hodnota dosáhne počtu dokumentů, `Promise` zastřešující tuto množinovou operaci je úspěšně vyřešen.

Mezi typy transformací patří separování názvů tříd a řádů zvířecí taxonomie do názvu českého a latinského (skrze modul *string*[27], který umožňuje získávat podřetězce obsažené v rodičovském řetězci mezi dvěma specifickými znaky), nebo přidání číselné hodnoty reprezentující délku trvání jedné události v ZOO v minutách (realizované pomocí modulu *moment*[28]). Další úpravy se týkají lexikonu, kde je při absenci `URL` obrázku zvířete učiněn pokus o získání této hodnoty z jeho textového popisu. K tomu slouží modul *cheerio*[29], který text zparsuje a v případě, že se jedná o `HTML` dokument, z něj i vytvoří stromovou strukturu podobnou té, s jakou pracuje internetový prohlížeč - výsledkem

[26] `https://nodejs.org/api/http.html`
[27] `https://www.npmjs.com/package/string`
[28] `https://www.npmjs.com/package/moment`
[29] `https://github.com/cheeriojs/cheerio`

je tedy *DOM*. V tomto stromě lze poté provádět vyhledávání a úpravy stejně jako v případě populární JavaScriptové knihovny *JQuery*[30], jejíž odlehčenou a pro `backend` připravenou verzí `cheerio` ve skutečnosti je. Následuje pokus nalézt element, jež obrázek reprezentuje a jeho zdrojovou `URL` z něj získat. Pokud se toto podaří, ale cesta neobsahuje počáteční doménu, je tato doplněna prostřednictvím hodnoty v `config` objektu. Dále je popis zvířete, který je tvořený `HTML` dokumentem, o tyto speciální značky zkrácen za použití modulu *striptags*[31], čímž zůstane čistý text. Speciálním případem transformace je provázání dokumentů o adopcích zvířat s jejich odpovídajícími záznamy v lexikonu. To probíhá na základě českého názvu, které se musí úplně shodovat. Předpokladem této vazby je existence lexikonu, který proto musí být naimportován ještě před adopcemi, k čemuž v `Importeru` slouží zvláštní metoda, vracející společný `Promise`. Tomuto postupu předcházela snaha využít možnost dotazovat zdroj v Open datech za pomoci `SQL`, jak bylo popsáno v sekci 1.4, a přenechat tak spojení lexikonu s adopcemi na vzdáleném serveru. Bohužel se při tomto postupu vyskytlo množství překážek provázených strohou dokumentací a absencí dostatečného množství příkladů, načež bylo od této varianty upuštěno.

V dalším kroku lze nové dokumenty konečně vložit do kolekce (a v případě že neexistuje ji vytvořit) jednou skupinovou operací namísto vkládání jednoho dokumentu po druhém. Následné přejmenování polí je opět provedeno za pomoci jedné operace, které jsou předány dvojice s mapováním z `config` objektu v předdefinovaném formátu. Posledním bodem je tvorba `indexů` na patřičných místech. Ty jsou potřeba na všech polích, která budou sloužit pro vyhledávání v kolekcích, takže např. jméno zvířete, typ klasifikace nebo začátek konání události v ZOO. `Indexy` jsou po sestavení z `config` objektu zaneseny do databáze opět další jednorázovou operací. Poté je import zdroje úspěšně dokončen.

3.3.5 Generování kvízových otázek

Skript realizující korektní vytvoření otázek pro vědomostní kvíz se nalézá v souboru `./generate-questions.js`. Ke svému chodu potřebuje stejné nástroje, jako importní skript - tedy `Logger`, `MongoClient` kvůli komunikaci s databází, `CollectionDriver` jakožto mezivrstvu a `config` s konfigurací. Kromě nich ale přenechává samotné generování třídě *QuestionGenerator* ze souboru `./modules/question-generator.js`. Hlavní skript si opět pouze vytvoří připojení k databázi a instance zmiňovaných tříd, načež závislosti předá do `QuestionGenerator`. Základ generování tvoří schopnost vybírat dokumenty z databáze náhodně, čehož je dosaženo funkcionalitou obsaženou v `MongoDB`, která přesně toto umožňuje. Vytvoří se v rámci dotazu agregační `pipeline`, která zahrnuje omezení dokumentů jak pomocí `query` tak limitem (ten je

[30] `https://jquery.com/`

[31] `https://www.npmjs.com/package/striptags`

potřeba aplikovat před další fází kvůli odhadu velikosti kolekce), po kterých následuje nastavení počtu náhodně vybraných dokumentů, které je potřeba získat.

Počet generovaných otázek se odvíjí od jejich typů. Jelikož existují tři typy, pro každý z nich se v rámci jednoho běhu skriptu vytvoří stejný počet otázek. Onen počet je udáván prostřednictvím argumentu, předávanému procesu při jeho spouštění a který má při absenci tohoto parametru výchozí hodnotu „5“. V případě aplikace výchozí hodnoty by tedy celkový počet otázek byl „15“, pět pro každý typ. Otázky se vytváří v cyklu, jehož každá iterace znamená vznik tří odlišných dokumentů. Pro každý z nich je ve třídě `QuestionGenerator` dostupná jedna specializovaná metoda.

Kostra metod má vždy podobný průběh, který se ovšem v kritických místech liší. Všechny mají společné to, že vrací `Promise`, který je vždy splněn a vrací pravdivostní hodnotu - buď byla otázka korektně vygenerována a je i originální, nebo v procesu došlo k chybě a otázka uložena nebyla (tento případ zahrnuje i možnost vygenerování duplicity). U prvních dvou typů, tedy čistě textových otázek, kde se hádá buď název zvířete nebo jeho vlastnost, je vždy třeba nejprve vybrat, o jakou variantu se bude v aktuální iterace jednat. Z `config` je tak náhodně vybrán jeden prvek z pole možných variant, který bude následně udávat znění otázky a databázové pole, jež bude stát v jejím centru (např. text „Víš, jaké zvíře se živí tímto druhem potravy: :value?“ a zainteresované pole *food_detail*, přičemž *:value* je tzv. *placeholder* - řetězec, který se bude později nahrazovat reálnou hodnotou). Po tomto kroku se učiní první dotaz do kolekce zvířat, který si klade za cíl získat primární dokument související s otázkou a obsahující i její správnou odpověď. Vytvoří se tím pádem požadavek na jeden náhodný dokument, který ono požadované pole za prvé obsahuje a za druhé ho má neprázdné. Po jeho nabytí se provádí kontrola duplicity - pokud v databázi již existuje otázka s identickým zněním i referencí na zvíře z lexikonu, proces skončí a musí se opakovat. Jestliže je ale nová otázka originální, přistoupí se ke generování jejích chybných odpovědí, které spočívá v získání příslušného počtu dokumentů, které opět obsahují dané pole, jehož hodnota se ale nesmí rovnat té v předchozím dokumentu. Poté se vytvoří nový objekt, reprezentující otázku - její znění vznikne nahrazením `placeholder` řetězce správnou hodnotou, doplní se k ní správná i špatné odpovědi, nastaví se její typ a výchozí hodnoty zbylých polí.

Postup vytváření otázek třetího typu je snazší. Zde je pouze jedna možná varianta - hádá se zvíře na obrázku. Proto stačí náhodně vybrat tolik dokumentů, kolik má otázka odpovědí a z výsledné množiny použít hned první jako zdroj obrázku i správné odpovědi. U zbylých je zajištěno, že se jedná o různá zvířata a mají tím pádem i odlišné obrázky. Kontrola duplicity i následné uložení nově vzniklé otázky už vypadají stejně, jako v případě prvních dvou typů.

Cyklus, který v `./generate-questions.js` koriguje tvorbu otázek, počítá s možností vzniklé duplicity. Řeší jí pomocí speciální `callback` funkce, která

kromě výsledku generování (což je pravdivostní hodnota) obsahuje i druhý argument, kterým je název metody, která se má v případě duplicity zavolat. Tím pádem se jedna metoda bude volat tak dlouho, dokud nevytvoří originální dokument. Duplicity hodně závisí na typu otázky - většinou je počet jejích instancí shora omezen počtem zvířat v lexikonu (za odlišnou otázku se nepovažuje ta, která má pouze jinak znějící špatné odpovědi), číslo je ale například v případě hádání podle obrázku výrazně nižší, jelikož hodně zvířat ho postrádá.

3.3.6 Server

3.3.6.1 Architektura

Aplikační server je lokalizován v `./server.js` a slouží k obsluhování všech příchozích požadavků na `API`. Využívá standardní závislosti v podobě `Logger`, `MongoClient`, `CollectionDriver` a `config`. Dále pracuje s modulem *express*[32], což je *webový framework* (soubor knihoven, pomocných tříd, metod a předdefinovaných postupů, usnadňujících tvorbu webové aplikace) a jež je považován za standard při vývoji takových projektů, jakým je `API` server. Jedná se dlouhodobě o jeden z nejstahovanějších modulů v `npm` a vyznačuje se vysokou rychlostí, snadností implementace, rozsahem funkcí, vysokou výkonností a pokročilým systémem *routování* příchozích požadavků (směřování jejich obsluhy do patřičných míst aplikace na základě dotazované `URL`). Rozsáhlá komunita kolem něj se také neustále podílí na jeho rozvoji, který se díky tomu nezastavuje.

`express` používá během svého fungování tzv. *middleware*, což jsou izolované funkce, kterými každý serverový požadavek prochází během svého životního cyklu. Každá taková funkce ho může modifikovat, validovat nebo provádět jiné činnosti, které jsou na proces zpracování navázané, načež předá požadavek další funkci v řadě. Předání se dá v případě vzniklé chyby vynechat a směřovat obsluhu přímo do speciálního typu `middleware`, který ji zpracuje.

Na principu `middleware` fungují i `routes`, díky čemuž lze aplikaci rozdělit do více logických celků, kde každý spravuje požadavky, které mu přísluší. Jeden `route` modul se nachází ve vlastním souboru, ze kterého exportuje pouze instanci třídy *Router*, do které sám zanáší vlastní `middleware` funkce. Ona instance je poté v kořenovém serverovém skriptu získána a napojena na její vlastní relativní cestu (např. „/adoptions“), díky které lze poté rozpoznat, do jaké části má obsluha požadavku směřovat.

Během vývoje byl také extenzivně používán nástroj *nodemon*[33]. Ten se integruje přímo s `Node.js` tak, že zaobalí soubory v adresáři, kde byl spuštěn, a které následně permanentně monitoruje. Pokud je v některém z nich detekována modifikace, `nodemon` automaticky restartuje běžící aplikaci, v tomto

[32] `https://expressjs.com/`
[33] `https://nodemon.io/`

případě server. Jeho název vznikl spojením slov „node“, jelikož interně využívá (a v tomto směru i nahrazuje) `Node.js` a „daemon“, což je systémový proces běžící na pozadí a není pod přímou kontrolou uživatele. Pomocí tohoto nástroje není třeba neustále restartovat server manuálně, což velice zefektivní a zrychlí vývoj.

Spuštění serveru probíhá až po připojení k databázi. To využívá vlastnosti `mongodb` modulu, který spravuje množinu těchto navázaných spojení a všechny následné požadavky, které budou na `API` přicházet a budou mít co do činění s persistentní vrstvou, se budou paralelně posílat databázovému serveru, jež je bude řešit po svém. Nedochází tak ke zdržování jednoho požadavku jiným a stačí k tomu pouze jedno databázové připojení při startu serveru. Jakmile je spojení navázáno vytvoří se instance potřebných tříd z modulů, přičemž `CollectionDriver` je nastavený přímo do `express` aplikace jako její atribut, přístupný pak ze všech míst, kde je instance frameworku dostupná. Následuje logování příchozího požadavku, které obsahuje `URL` (kterou je potřeba sestavit z protokolu, domény a cesty k datovému zdroji), IP adresu klienta (jeho identifikátor v síti) a aktuální datum. Daná `URL` se udržuje napříč celým životním cyklem požadavku, jelikož je potřeba ke generování posloupnosti odkazů, jak bylo definováno v sekci 3.1.3.1 o nutnosti dodržování `HATEOAS`.

Před zpracováváním požadavku je ještě třeba zkontrolovat jeho validitu, spočívající ve splnění požadavků stanovených specifikací `JSON-API`. K tomu vypomáhá modul *body-parser*[34] - `middleware` parsující tělo požadavku do správného formátu, se kterým je možné dále pracovat. Po parsovacím procesu je k dispozici obsah ve formátu `JSON`, který musí být typu `application/-vnd.api+json`, jak bylo uvedeno v sekci 3.1.3.2. Pokud není, server ho není schopen rozpoznat a zpracovat, což ústí v `HTTP` chybu s kódem `415 - Unsupported Media Type`. Další kontrola se týká typu dat, která klient akceptuje. Pokud není kompatibilní s `JSON-API`, je vygenerována `HTTP` chyba s kódem `406 - Not Acceptable`. Server v každém případě vrací data v příslušném formátu, čili v dalším kroku nastaví odpovídající hlavičku a postoupí k nadefinování jednotlivých `routes` submodulů, které požadavek zpracují.

Posledním místem v hlavním skriptu je zachytávání chyb. První `middleware` funkce zpracovává jak všechny serverové chyby, tak chyby vznikající v `routovacích` submodulech. Pokud obdrží standardní JavaScriptovou chybu, tedy instanci třídy *Error*[35] (která mohla nastat databázovou, syntaktickou nebo logickou chybou), překonvertuje ji na `HTTP` chybu s kódem `500 - Internal server error`. Pokud se o tuto instanci nejedná, předpokládá se, že je objekt již korektně zformátován pro následnou serializaci. Ta probíhá (po zalogování vzniklé chyby) prostřednictvím třídy *Error* z modulu *jsonapi-serializer*[36]. Odpověď je poté se správně nastaveným `HTTP` kódem odeslána

[34] `https://www.npmjs.com/package/body-parser`

[35] `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Error`

[36] `https://github.com/SeyZ/jsonapi-serializer`

zpět klientovi. Poslední `middleware` funkce se volá v momentě, kdy se požadavku neujme žádná předchozí, což je vyhodnoceno jako špatná `URL`. Tehdy se vrátí `HTTP` chyba s kódem `404 - Not found`.

3.3.6.2 Routes

Každá databázová kolekce má svůj vlastní `route` modul, přičemž se všechny nachází v adresáři `./routes/`. Kromě kolekcí existují ještě tři uměle vytvořené zdroje, které se interně všechny dotazují do lexikonu - `biotopes`, `continents` a `food`. Vedle nich je v tom samém adresáři soubor `./routes/middleware.js`, který obsahuje modul exportující třídu *Middleware*. V ní jsou funkce sehrávající podobnou úlohu, jako `middleware` v samotném `express` - nabízejí funkcionalitu společnou pro více `routes`. Funkce jsou obecné a univerzální, pokaždé očekávající jen specifické argumenty, s jejichž využitím poté vrátí potřebný nástroj nebo výsledek. Zahrnují validaci příchozího požadavku, vytvoření `JSON-API` kompatibilního serializátoru, vygenerování databázového query objektu a metodu pro zpracování zmiňovaných uměle vytvořených zdrojů.

Proces zpracování požadavku v některé z `routes` sestává z následujících kroků:

1. Načtení lokálních proměnných z `config` objektu - název kolekce, povolené `query` parametry a pole k serializaci.
2. Inicializace `Middleware` třídy.
3. Validace `query` parametrů.
4. V případě úspěšné validace nastává inicializace zbylých lokálních proměnných (`CollectionDriver` a `Serializer`).
5. Odchycení požadavku správnou `routovací` funkcí.
6. Provedení operací nutných k vyřešení požadavku.
7. Vytvoření, serializace a odeslání odpovědi.

Třetí krok - validace - se odehrává v režii modulu *joi*[37], který umožňuje definovat povolené parametry, nalézající se v `URL` požadavku (jak bylo popsáno v sekci 3.1.3.1). Každému z nich lze dále přiřadit, jaká omezení budou kladena na jejich hodnoty - specifikovat datový typ, délku, rozsah, zda jsou povinné či nikoliv atd. Takto vzniklý objekt, nazývaný „validační schéma“ se předá instanci třídy `Serializer` ve funkci poskytované třídou `Middleware`, která proti němu porovná `query` parametry a vrátí výsledek. Ten může být buď úspěšný nebo obsahovat chyby, na které server reaguje odpovědí klientovi obsahující `HTTP` chybu s kódem `422 - Unprocessable Entity`. Celý tento

[37] `https://github.com/hapijs/joi`

proces je v aplikaci z toho důvodu, aby mohl být klient jasně informován o chybách v návrhu a především aby se předcházelo tzv. *NoSQL injection*[13], což je typ databázového útoku cílící na nedůkladně ošetřené dotazy, prostřednictvím kterých je útočník schopen způsobit rozsáhlé škody, od získání citlivých informací až po smazání všech dat. `MongoDB` tento problém částečně odstiňuje díky způsobu, jakým dotazy provádí - do předpřipravené klauzule doplňuje klientem dodané parametry, čímž zabrání jejich zásahu do samotného dotazu. Vzhledem k možnosti posílat do databáze i požadavky ve formě JavaScriptu, který je databázovým enginem následně prováděn, je třeba znemožnit i injektování objektů a spustitelného kódu. Modul `joi` byl zvolen z důvodu své jednoduchosti - existuje i jeho nadstavba v podobě modulu *celebrate*[38], která se integruje přímo do `express`, ale pro současné požadavky serveru je základní možnost více než dostačující.

Transformace odpovědi je prováděna skrze modul `jsonapi-serializer`, jenž obsahuje třídy pro serializaci i deserializaci dat odpovídajících `JSON-API` specifikaci. Kromě atributů, které dané dokumenty obsahují, třída `Serializer` dále vyžaduje název zdroje (kolekce), do které dokumenty patří, a objekt obsahující nastavení serializace. V něm se nachází možnosti jako název atributu, který se má použít jako identifikátor dokumentu, syntax názvů klíčů (musí být specifikováno, že se jedná o *snake case* a jednotlivá slova jsou tedy oddělena podtržítkem, jelikož tato skutečnost vyplývá z databázového návrhu), odkazy na související zdroje a nastavení vlastností `meta` atributu. Tam se aktuálně vyskytuje pouze jedna - počet vrácených dokumentů. Tato hodnota je počítána pomocí vnořené JavaScriptové funkce, která pracuje nad dokumenty předanými serializátoru. Objekt obsahující odkazy na další související zdroje (jako je navigace v rámci stránkování) má buď jednoduchou podobu a skrývá pouze volanou `URL`, nebo je vytvořen na základě vstupních `query` parametrů. To má opět za úkol jedna z funkcí třídy `Middleware`, která spravuje generování stránkovacích odkazů a jejím výsledkem je čtveřice nových `URL`: první a poslední stránky, předchozí a následující. Po vytvoření instance `Serializer` už jí zbývá jen předat pole dokumentů, které je třeba serializovat.

Další opakující se funkcionalitou mezi `routes` je sestavení databázového dotazu, na základě kterého jsou vráceny dokumenty. To zařizuje další funkce v `Middleware`, která vytváří z příchozích `query` parametrů objekt obsahující kromě samotného dotazu i limit a ofset. Tento objekt očekává jako svůj vstupní argument metoda „findAllDocuments“ třídy `CollectionDriver`, která se na jeho základě dotáže databáze. Při sestavování je kladena zvláštní důraz na `query` parametr „name“, který jako jediný má na sobě ve dvou kolekcích („lexicon“ a „adoptions“) nasazený textový index pro efektivnější fulltextové vyhledávání. Pokud je v příchozím požadavku obsažen, je pro něj vytvořen speciální objekt, kde je definováno, že se s jeho pomocí má vyhledávat spolu s ignorováním diakritiky a rozdílů mezi velkými a malými písmeny. Všechny

[38] `https://github.com/continuationlabs/celebrate`

ostatní `query` parametry jsou také textové, ale používají se už jen jako základ jednoduššího vyhledávaní za pomoci regulárního výrazu. To je sice také schopné ignorovat velikost písmen, ale diakritiku už ne. Navíc je již z principu výrazně pomalejší (porovnávání řetězců proti regulárním výrazům je velice nákladná operace).

Poslední sdílená metoda z `Middleware` je určena ke zpracování uměle vytvořených zdrojů dat, zmiňovaných na začátku sekce 3.3.6.2. Každý z nich je tvořený množinou originálních hodnot z jednoho pole v kolekci „lexicon“. Tyto hodnoty se nejprve získají, poté se z nich musí vytvořit objekty, aby bylo možno je serializovat, načež je pro každý takto vzniklý dokument nalezen počet výskytů v lexikonu s jeho hodnotou. Kdykoliv nastane chybová událost, je propagována výše a je zachycena až v hlavním serverovém skriptu, odkud je klientova vrácena.

Jednotlivé `routes` mají svá specifika. Ne všude je například potřeba validovat `query` parametry, jelikož nejsou žádné vyžadovány. Nejvíce rozdílů je přirozeně při získávání dat pro odpověď. Zde hodně metod aplikuje na vyhledávání vlastní řazení, aby zjednodušila následné zpracování vrácených dat. `Route` pro klasifikace zvířat generuje taxonomii včetně vzájemných závislostí, takže každá třída obsahuje pole svých řádů. `Route` pro události zase vytváří z příchozího požadovaného časového údaje omezení v podobě projekce, která události filtruje na ty, které začnou buď po specifikovaném datumu nebo sice už začaly, ale skončí až po něm. Všechny `routes` ale končí buď odesláním odpovědi klientovi nebo předáním chyby do k tomu určené `middleware` funkce.

3.4 Testování

Kontrola správného fungování importního skriptu a generování kvízových otázek probíhala pouze vizuálně přímo v databázi s využitím grafického rozhraní postaveném pro `MongoDB`. Během vývoje se jednotlivé části ladily pomocí zachytávání chyb a logování debugovacích zpráv.

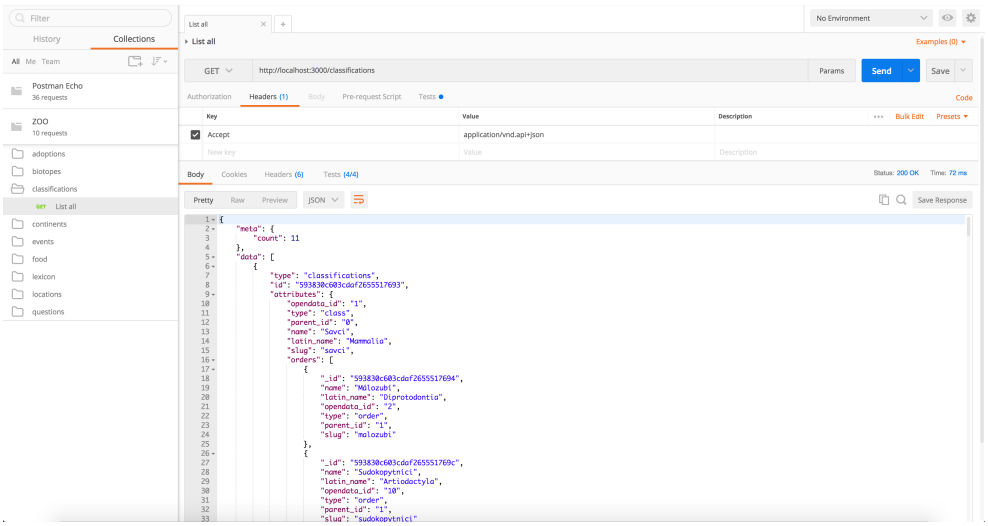
Korektní funkčnost serverového `API` byla testována prostřednictvím nástroje *Postman*[39]. Ten umožňuje vytvářet kolekce dotazů, rozdělovat je do složek a spravovat podle aktuálních požadavků. Podporovány jsou všechny `HTTP` metody, zmiňované v sekci 3.1.3.1 a uživatel má úplnou kontrolu nad jejich využitím, včetně tvorby těla i hlaviček požadavku. Odpovědi, jež se od serveru vrací, je dále možné testovat - zda byly úspěšné nebo zda obsahují tělo a pokud ano, tak jestli je validní a ve správném formátu. Tyto testy jsou psané v JavaScriptu a spustí se automaticky po obdržení odpovědi.

Na každý datový zdroj, definovaný v dokumentaci `API` a popsaný v sekci 3.2.3, vznikl minimálně jeden požadavek, který má za úkol otestovat konkrétní případ volání `URL`. Většinu z nich lze libovolně opakovat se identickými

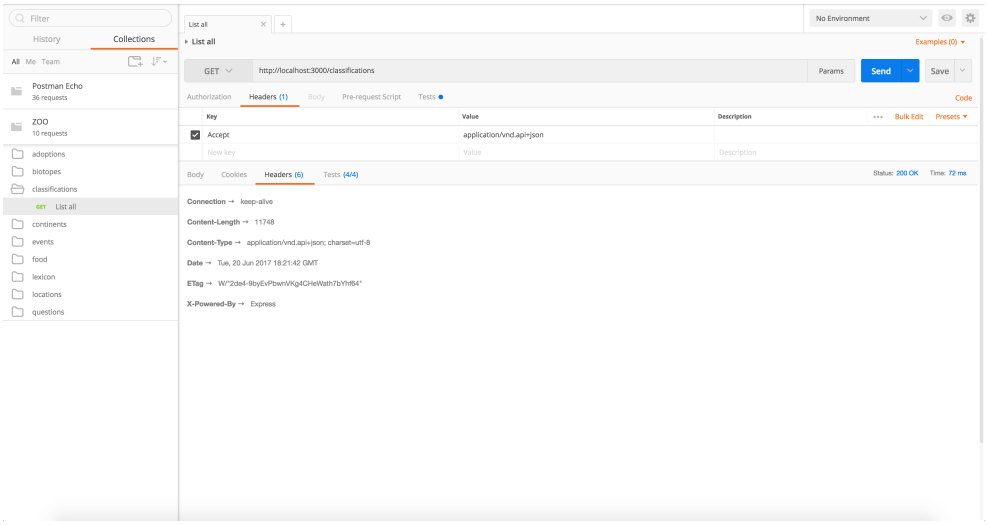
[39] `https://www.getpostman.com/`

výsledky, jiné (např. získání určitého počtu náhodně vybraných kvízových otázek) se budou při opakování lišit.

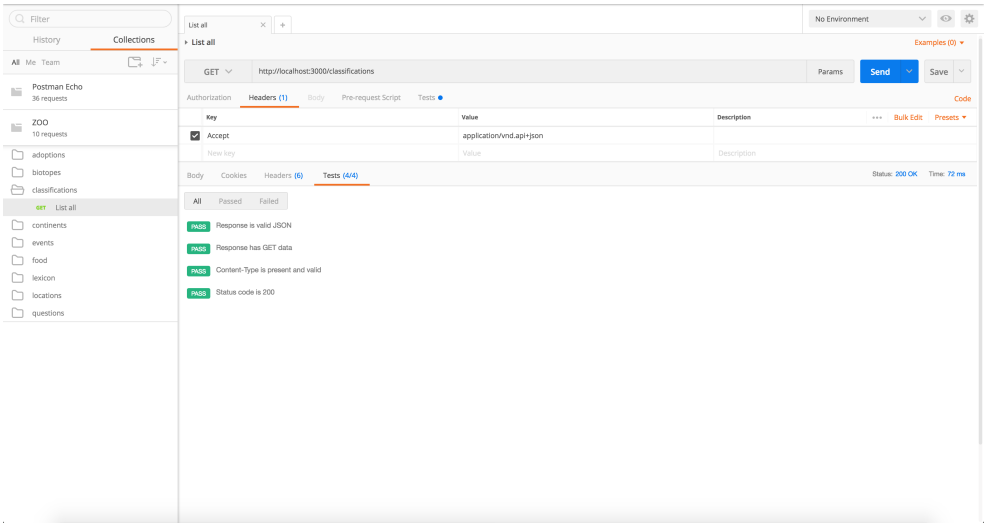
Obrázek 3.5 zobrazuje tělo získané odpovědi, snímek 3.6 její hlavičky a 3.7 výsledky testů.



Obrázek 3.5: Tělo odpovědi z `API` v aplikaci `Postman`



Obrázek 3.6: Hlavičky odpovědi z `API` v aplikaci `Postman`



Obrázek 3.7: Testy odpovědi z `API` v aplikaci `Postman`

3.5 Nasazení

Finálním krokem je nasazení serverové části do produkčního prostředí. `API` bude tím pádem veřejně dostupné, připravené přijímat a odpovídat na požadavky od mobilní aplikace. Pro samotnou platformu `Node.js` existuje množství řešení[14], přičemž vybíráno bylo z těch, které poskytují základní plán zdarma. Z nich bylo na základě dřívějších zkušeností autora práce vybráno *Heroku*[40]. Toto prostředí poskytuje vlastní program, přes který lze se vzdáleným repozitářem komunikovat a je úzce spojený s nástrojem `GIT`, který byl použit i při vývoji `backendu`. Aplikace se do produkce snadno nahrávají právě přes tento verzovací systém, přičemž je dále k dispozici možnost studovat logované záznamy, přistupovat do repozitáře přes konzoli a spouštět tak jednotlivé skripty ručně, případně své aplikace včetně jejich dat spravovat skrze webové rozhraní. Velikost RAM je 512 MB, což je pro aktuální požadavky dostatečné, stejně tak 128 MB místa na disku.

Nevýhodou je nutnost poskytnutí informací o platební kartě při registraci do systému a nemožnost výběru umístění serveru, kde aplikace běží. To vyústilo v její nasazení ve Spojených státech, což spolu s pouze dvěma procesy, které obsluhují požadavky a faktem, že se jediné *dyno* (kontejner, ve kterém se aplikace nachází[15]) při delší neaktivitě ukládá ke spánku, ústí v pomalejší reakční dobu serveru. Naproti tomu ale `Heroku` automaticky poskytuje šifrované spojení přes *HTTPS* a honosí se možností připojit si ke svému účtu díky tzv. *add-on* databázi. Aplikace tak má neustále přístup k vlastní instanci

[40] `https://www.heroku.com`

`MongoDB`, nad kterou pracuje. Její obsah lze stejně jako mnoho jiného kontrolovat a upravovat přes webové rozhraní. Připojení k databázi stejně jako síťový port, na kterém je spuštěný server dostupný, jsou z aplikace dostupné skrze proměnné prostředí *MONGODB_URI* a *PORT*.

Vybrané prostředí současným nárokům vyhovuje a v případě jejich nárůstu lze buď zakoupit jeden z pokročilejších plánů přímo na `Heroku` nebo vybrat jednu z mnoha alternativ. Aktuálně je server funkční a spuštěný na `https://zoo-backend.herokuapp.com/`.

Kapitola 4

Mobilní aplikace

4.1 Vývojové prostředí

Aplikace je cílena na mobilní operační systém Android, vyvíjený společností Google a založený na Linux jádru. Jako taková musí pokrývat požadavky velkého množství koncových zařízení, jelikož se jedná o nejrozšířenější platformu napříč chytrými telefony i tablety, které nad základní verzí systému obvykle staví svůj proprietární software. V kombinaci s množstvím aktivních verzí samotného Android je tak třeba dbát na kvalitní a univerzální návrh takové aplikace, aby bylo dosaženo co nejvyšší kompatibility.

Prostředkem k dosažení stanovených cílů je nástroj Android Studio[41], což je *IDE* (integrované vývojové prostředí) vyvíjené samotnou společností Google speciálně pro vývoj aplikací na platformu Android. Má tedy přímý přístup k `API` tohoto systému a snadno integruje další externí knihovny. Poskytuje rozsáhlé spektrum funkcí, počínaje standardními, jako je kontrola, kompilace, našeptávání a generování kódu. Vedle nich obsahuje emulátor, ve kterém lze vytvářet virtuální mobilní zařízení a na nich poté aplikaci testovat, což výrazně přispívá k variabilitě a rozsahu při optimalizaci pro různé přístroje. Výstup emulátorů je možné ukládat jako obrázky, což je v práci využíváno pro poskytnutí náhledů. Dále je v `IDE` přítomný návrhový editor, díky kterému lze přehledně sestavovat vzhled a rozložení jednotlivých obrazovek aplikace, což usnadňuje tvorbu její vizuální podoby.

4.2 Návrh

4.2.1 Prototyp

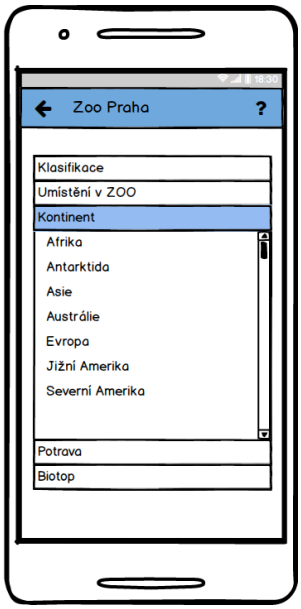
Kvůli testování funkcionality a uživatelského rozhraní aplikace byl vytvořen funkční prototyp, který byl následně poskytnut testovacím subjektům. Jejich

[41] `https://developer.android.com/studio/index.html`

odezva byla zaznamenávána a sloužila jako cenný zdroj informací, využívaných k úpravám konečného návrhu.

K výrobě prototypu byl vybrán nástroj *Balsamiq mockups*[42], který poskytuje intuitivní rozhraní pro tvorbu návrhů mobilních aplikací. Obsahuje všechny požadované části, popsané v sekci 2 včetně jejich funkcí. Vedle nich se v něm objevila i možnost uživatelské registrace a interaktivní mapa areálu ZOO, které byly součástí raného návrhu, ale byly později odloženy. Detailně jsou popsány v sekci 5 jakožto rozšíření do budoucna.

Prototyp byl vygenerován do souboru PDF, ve kterém byl přesně zachován veškerý vzhled a funkce z návrhu. Jednotlivé obrazovky jsou mezi sebou provázány pomocí klikatelných odkazů, které jsou v souboru zvýrazněny. Uživatel se tak může volně přesouvat mezi stavy aplikace a simulovat tak reálný průchod. Snímky několika obrazovek slouží jako ukázka výsledného prototypu: 4.1 zobrazuje menu s filtry lexikonu, 4.2 detail zvířete, 4.3 kvízovou otázku a 4.4 seznam adopcí s aplikovaným vyhledávaným řetězcem. Celý soubor s prototypem se nachází na CD.



Obrázek 4.1: Ukázka obrazovky „menu lexikonu“ v prototypu mobilní aplikace

[42] `https://balsamiq.com/`

Obrázek 4.2: Ukázka obrazovky „detail zvířete“ v prototypu mobilní aplikace

Obrázek 4.3: Ukázka obrazovky „kvízová otázka“ v prototypu mobilní aplikace

Obrázek 4.4: Ukázka obrazovky „vyhledávání v seznamu adopcí“ v prototypu mobilní aplikace

4.2.2 Persistence dat

Během komunikace s `backend API` se budou do mobilní aplikace dostávat aktuální data, která bude třeba uchovávat. Díky tomu bude možné s aplikací manipulovat i v offline režimu, kdy nebude dostupné síťové připojení. Další výhodou je odpadnutí nutnosti data importovat v případě, že se v `backend` databázi od posledního dotazu nijak nezměnila. Kromě toho je nutná persistence informací a struktur, které se používají pouze v rámci mobilní aplikace a s `API` nijak nesouvisí. Tyto důvody přirozeně ústí v potřebu implementovat databázovou vrstvu.

Přirozenou volbou je využití systému *SQLite*[43], který se díky svému rozšíření v internetových prohlížečích, operačních systémech a vestavěných systémech (kterým je i Android) honosí titulem nejčastěji používaného databázového enginu ve světě[16]. Jedná se o relační systém a jako takový implementuje většinu vlastností, známých z jeho masivnějších souputníků, jakými jsou např. MySQL, PostgreSQL (který slouží `SQLite` jakožto vzorový a referenční systém) nebo Oracle Database. Na rozdíl od nich je ovšem vestavěný přímo do koncového programu a nepotřebuje tedy vlastní dedikovaný server, na který by se aplikace připojovala. To je spolu s faktem, že je napsaný v jazyce *C*, faktor přispívající k jeho vysoké rychlosti a efektivitě.

Splňuje transakční podmínky *ACID* - atomicitu, konzistenci, izolovanost a trvanlivost dat ocitajících se v transakcích. Není vhodná pro intenzivní pa-

[43] `https://www.sqlite.org/`

ralelní operace zápisu, jelikož se musí soubor, obsahující data, při každém přístupu uzamknout. Tím pádem je zápis sekvenční, čtení ale může být prováděno simultánně. Výrazným rozdílem oproti konkurenci je dynamické typování sloupců v tabulkách, které tedy nemají pevně stanovené schéma a datové typy jsou navázány na konkrétních hodnotách v rámci záznamů, spíše než globálně. Kvůli tomu není zajištěna doménová integrita.

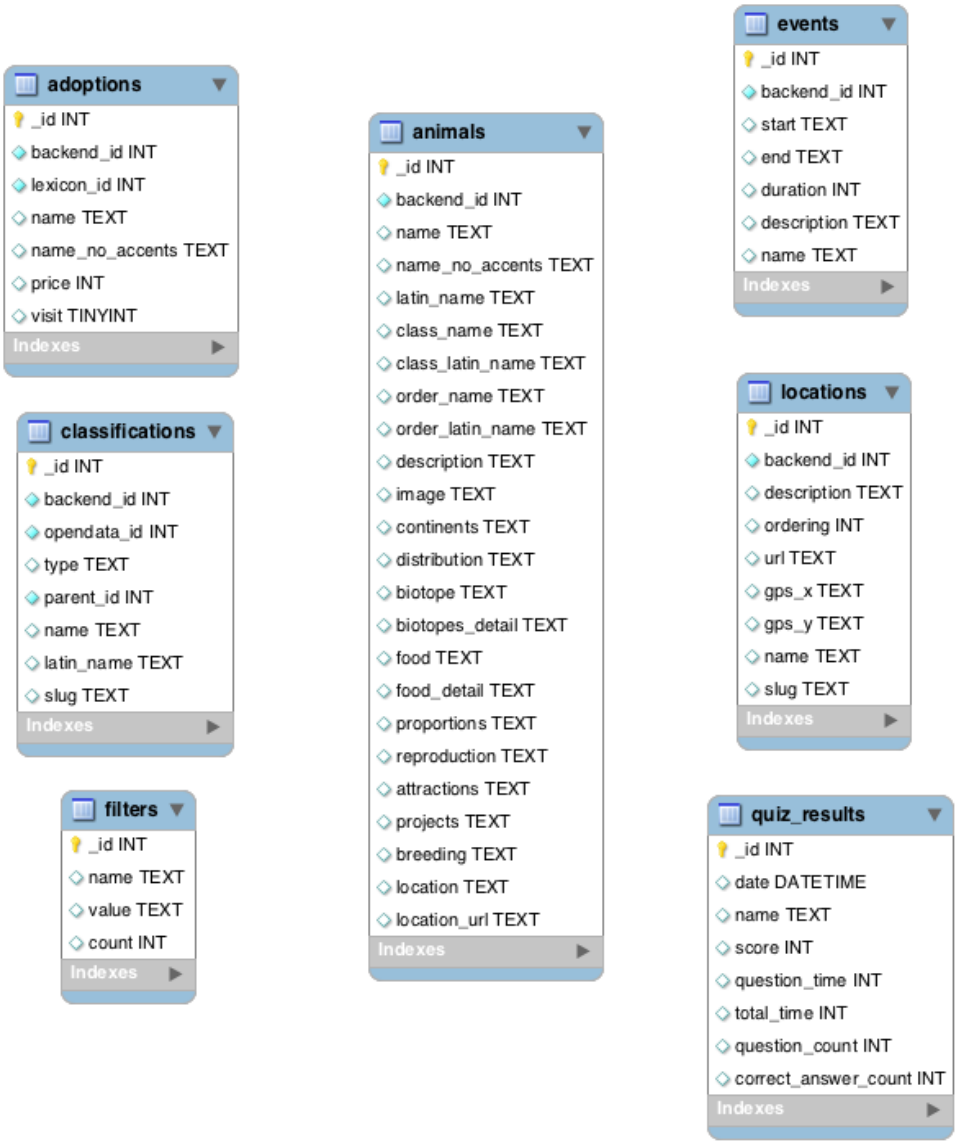
Výhody `SQLite` jednoznačně převyšují nad minusy, jelikož většina operací, prováděných v rámci aplikace, bude data pouze číst. Zápis bude prováděn buď hromadnými jednorázovými požadavky po importu z `API` nebo jednotlivými dotazy na izolovaných místech. Podpora v systému Android je rozsáhlá a lze snadno i rychle implementovat.

4.3 Implementace

4.3.1 Datová vrstva

4.3.1.1 Struktura

Databázové schéma pokrývá veškeré kolekce, které se nachází na `backend` serveru a jsou následně dostupná skrze jeho `API`. Datové typy sloupců jsou stejně tak ekvivalentní dostupným polím kolekcí, které jsou do tabulek promítnuty s odpovídajícími názvy. Jediné přírůstky jsou pole „name_no_accents“ v tabulkách *adoptions* a *animals*, které slouží pro vyhledávání a jsou popsané v sekci 4.3.3.8. Název tabulky `animals` byl změněn z původního názvu kolekce `lexicon`, aby lépe reflektoval třídní návrh mobilní aplikace (pracuje se zde s třídou *Animal*). Popis obsahu tabulek odpovídá tomu v sekci 3.2.2. Tabulky specifické pro mobilní aplikaci jsou na obrázku 4.5.



Obrázek 4.5: Schéma databáze v mobilní aplikaci

Název pole	Popis
date	Datum vzniku výsledku
name	Jméno uživatele
score	Výsledné skóre
question_time	Čas na zodpovězení otázky
total_time	Celkový čas strávený hádáním
question_count	Počet zobrazených otázek
correct_answer_count	Počet správně zodpovězených otázek

Tabulka 4.1: Popis polí v tabulce *quiz_results* (Výsledky kvízu)

Název pole	Popis
name	Název filtru (např. „continents“)
value	Hodnota filtru (např. „Afrika“)
count	Počet zvířat odpovídajících filtru

Tabulka 4.2: Popis polí v tabulce *filters* (Filtry použité v menu lexikonu)

Funkcionalita týkající se přímé komunikace s databází se nachází v `Java package` s názvem *db*. Schéma je definováno ve vlastní třídě *ZooDBSchema*, která pro každou databázovou tabulku obsahuje dedikovanou vnitřní statickou třídu. Ty v sobě stejným způsobem udržují třídu reprezentující názvy sloupců a vedle nich také jméno tabulky. Tento způsob je preferovaný, jelikož tak schéma izoluje od zbytku aplikace a všechny jeho části navenek nabízí skrze statické atributy. Nic dalšího není pro definici schématu potřeba.

Dalším krokem při tvorbě databáze je třída *ZooBaseHelper*, extendující Android třídu *SQLiteOpenHelper*. Ta obsahuje jméno databáze a dvě metody - jedna volaná v případě, že databáze při iniciaci třídy neexistuje (v ní se nachází sestavené dotazy, vytvářející její strukturu včetně `indexů`) a druhá pokud se zvýší verze databáze (tato metoda bude obsahovat update tabulek a struktury). `ZooBaseHelper` používá návrhový vzor *singleton*, což znamená, že za běhu aplikace vždy existuje maximálně jedna její instance. Toho je dosaženo za pomoci jejího navázání na aplikační kontext, díky čemuž jí systém nemůže zničit aniž by se přitom dotkl samotné rodičovské aplikace. Tato volba byla učiněna proto, že je třída dotazována z různých míst a pokaždé za stejným účelem - poskytnutí objektu pro práci s databází. Kvůli tomu je zbytečné pokaždé vytvářet novou instanci.

Poslední třídou v `db Java package` je *ZooCursorWrapper*, dědící z Android třídy *CursorWrapper*. Ta slouží k mapování databázových záznamů na Java třídy - modely. V rámci dotazu vrací databáze kursor, ukazující na první z množiny výsledků a to z toho důvodu, aby se přes ně dalo následně iterovat. Je totiž zbytečné vracet okamžitě veškeré odpovídající výsledky, pokud může

být zpracován pouze jejich zlomek. Tento kursor vždy ukazuje na jeden výsledek, který je reprezentovaný hodnotami svých sloupců. Pomocí jejich názvů ze třídy `ZooDBSchema` jsou získány indexy těchto polí a hodnoty navázány na novou instanci požadované třídy (např. „Animal“). Jedná se tak o základní implementaci `ORM`.

4.3.1.2 Model

Tato vrstva má za úkol reprezentovat obsah databáze v rámci `OOP`, tedy jako entity. Každá tabulka má svou ekvivalentní strukturu, jejíž návrh odpovídá tzv. „POJO“ - jednoduchá třída, nemající ani jednoho rodiče a neimplementující žádný interface. Obsahuje pouze skladbu privátních proměnných, jejichž přístup realizuje prostřednictvím getterů a setterů. Pro každý zdroj, který je dostupný v `backend API`, se v `Java package` s názvem *model* nachází jedna taková třída a stejně tak pro zbylé tabulky, využívané pouze v mobilní aplikaci. Všechny modely používají anotace pro knihovnu *GSON*[44], která je zde využita serializaci a deserializaci dat ve formátu JSON. Umožňuje definovat, jaké atributy v JSON objektu odpovídají kterým proměnným Java třídě a prostřednictvím tohoto namapování provede konverzi. Importovaná data tak prochází dvěma transformacemi - nejdříve z JSON do Java tříd, a z nich poté do databázových entit. Zmiňované anotace taktéž specifikují, které proměnné se konvertovat nemají, jelikož slouží pouze k lokálním účelům.

Speciálním případem modelů jsou třídy *JsonApiObject* a *JsonApiError*, které zaobalují celou odpověď z `backend API`. V případě chyby je transformace jednoznačná, jelikož odpověď serveru obsahuje pouze tři kořenové atributy. Standardní odpověď s `HTTP` kódem typu `2xx` nebo `3xx` má ovšem složitější strukturu, která vyžaduje používání vnitřních tříd a následnou finální fázi, která z nich vypreparuje konkrétní data. Tento postup je popsán v sekcích 4.3.2 a 4.3.3.6.

4.3.2 Komunikace s API

Dotazování na `backend API` je realizováno pomocí knihovny *Retrofit*[45], která zařizuje vše od úvodního navázání spojení, přes obdržení odpovědi až po její zpracování. Její jádro tvoří interface *BackendApi*, definovaný v `Java package` s názvem *api*. V něm jsou implementovány všechny cesty nabízené přes `API` tak, jak byly popsány v sekci 3.2.3. Ke každému odchozímu požadavku je nastavena nová hlavička, značící akceptování dat ve formátu odpovídajícímu `application/vnd.api+json`, jak vyplývá z definice `JSON-API` v sekci 3.1.3.2 a s pomocí anotací je označen typ požadavku, který se bude provádět (aktuálně se vždy jedná o `HTTP` metodu `GET`). Odpověď na požadavek je zaobalena do instance třídy *Call*, která očekává generický typ, do kterého bude

[44] `https://github.com/google/gson`
[45] `http://square.github.io/retrofit/`

data transformovat - v případě úspěšného požadavku se vždy jedná o třídu `JsonApiObject`, která respektuje strukturu `JSON-API` - počítá s kořenovými atributy `links`, `meta` a `data`, jež si na sebe nechá namapovat. `data` dále obsahují vnořený objekt se samotnými získanými dokumenty, který je uchováván v instanci Java třídy *JsonObject* a ze které jsou dokumenty posléze transformované do konkrétních modelů, jak je popsáno v sekci 4.3.3.6.

Instance třídy `Retrofit` je využívána třídou *DataFetcher*. Při jejím vytváření se nejprve vyrobí nový `HTTP` klient ze třídy *OkHttpClient*, jelikož bude potřeba mít kontrolu nad jeho vlastnostmi. Mezi ně patří časové limity, které se na komunikaci s `API` aplikují (týkají se spojení, čtení i zapisování) a mezipaměť, kterou `Retrofit` využívá k ukládání požadavků (ta potřebuje znát svou velikost a umístění). V dalším kroku se tento nově vzniklý klient spolu s `endpointem APi` a instancí `GSON` využije k vytvoření `Retrofit` instance. Té se předá interface, jež bude implementovat (čili `BackendApi`) a výsledná služba již může zprostředkovávat komunikaci.

`DataFetcher` vlastní pro každou z metod z implementovaného interface vlastní metodu, která slouží k jejímu zaobalení. Své vstupní parametry předá dále a ty se poté promění v `query` a `path` parametry, které jsou připojeny do finální `URL`. `Retrofit` umožňuje dvojí volání metod - synchronní a asynchronní. První varianta předpokládá, že se bude volání konat ve vedlejším procesorovém vlákně, poněvadž v hlavním `UI vlákně` se síťové operace provádět nesmí pod hrozbou vyhozené výjimky. Využívá se tak k operacím, které vyžadují více dotazů provedených po sobě a až po jejich provedení pokračovat v práci. Druhá, asynchronní možnost je preferovaná, jelikož sama přesune operaci do jiného vlákna a po jejím zpracování vyvolá příslušnou událost. Na tuto událost reaguje `callback` třída, vnořená do `DataFetcher`, implementující dvě metody - jedna se provede při úspěchu požadavku, druhá při síťové chybě (vypršel časový limit na odpověď, není dostupné síťové připojení, `endpoint` nebyl nalezen apod.). V obou případech je informován vnější naslouchač, který požadavek inicioval. To je objekt, který si vyžádal import dat a nyní čeká na výsledek své žádosti skrze interface třídy `DataFetcher`, který implementoval. Po obdržení tohoto výsledku (kterým je kromě získaného `JsonApiObject` objektu i `HTTP status` a *ETag* hlavička odpovědi, využívané k identifikaci uložené odpovědi) s ním může naslouchač dále manipulovat.

Nevýhodou knihovny `Retrofit` je fakt, že neparsuje odpověď v případě, že je chybová (vrátí se spolu s `HTTP kódem` typu `4xx` nebo `5xx`). V takovém případě je třeba ji na objekt `JsonApiError` namapovat manuálně.

Velkým plusem je ale možnost ukládat si příchozí odpovědi z `API` do mezipaměti a předcházet tak jejich opětovnému načítání ze serveru i v případě, že se zdrojová data nezměnila. Tím pádem využívá jeden ze základních principů `REST` architektury, definovaných v sekci 3.1.3.1. Realizuje to díky mechanismus `HTTP` s názvem *ETag*[17], což je řetězec znaků, vytvářený nejčastěji hash funkcí, která se pustí na zdrojový obsah. Výsledný řetězec data jasně reprezentuje, jelikož každá jejich změna vede k odlišnému výsledku hash funkce.

Generování HTTP hlavičky s ETagem zařizuje na backend serveru framework express, přičemž používá tzv. „weak“, tedy „slabý“ ETag, který indikuje, že jsou zdroje sémanticky ekvivalentní (a tedy ne ekvivalentní bajt po bajtu), což pro potřeby aplikace postačuje. Retrofit spolu s požadavkem posílá HTTP hlavičku If-None-Match, jejíž hodnotou je ETag posledního poptávaného zdroje. Pokud se data na serveru nezměnila, vrací se aplikaci pouze informace o této skutečnosti prostřednictvím HTTP kódu 304 - Not Modified bez těla odpovědi. Tělo pak Retrofit získá z mezipaměti.

Na tuto variantu musí být aplikace ale připravena reagovat po svém. Pokud se v minulosti odpověď od serveru vrátila a byla v pořádku uschována v cache, předpokládá se, že byla následně naslouchačem korektně zpracována a uložena. Při každém dalším požadavku, vráceném z cache, by se tak data zpracovávala znovu a zbytečně. Tomu je třeba v rámci zrychlení aplikace předcházet, kvůli čemuž byl implementován mechanismus využívající *internal storage* - úložný prostor, který má ve své soukromé správě samotná aplikace a který může využívat k hospodaření se soubory. Tuto možnost zastřešuje třída *InternalStorageDriver*, která má dvě hlavní úlohy, reprezentované odpovídajícími metodami - uložit do souboru ETag odpovědi, která byla v pořádku zpracována a případně i uložena v databázi, a prohledání toho samého souboru kvůli zjištění, zda v něm daný ETag je už uložen. Vzhledem k unikátnosti řetězce, který je vygenerován hash funkcí, není třeba ETag mazat, jelikož pravděpodobnost jeho duplicity je mizivá. Očekávané chování objektu, obsluhujícího odpověď serveru, je tedy následující:

1. Získání originálního HTTP kódu a ETag hodnoty přímo z odpovědi serveru
2. Kontrola, ze se jedná o odpověď typu 304 - Not Modified
3. Pokud ne, zpracovat ji a uložit její ETag do internal storage
4. Pokud ano, nahlédnout do internal storage, zda byla odpověď již zpracována
5. Jestliže nebyla, vykonat třetí krok a skončit
6. Jestliže ale zpracována byla, obsluha okamžitě končí

4.3.3 Programová část

4.3.3.1 Konfigurace projektu

Aplikace musí obsahovat soubor AndroidManifest.xml, který slouží jako informační základ pro Android. Definuje název Java package, ve které se aplikace nachází (cz.zdrubecky.zoopraha), specifikuje povolení, které bude aplikace vyžadovat pro přístup do chráněných oblastní systému (přístup k Internetu), popisuje jednotlivé komponenty, jejich chování a provázanost (názvy

a umístění aktivit, případně jejich rodičovské třídy ve smyslu závislosti - ne dědičnosti), ikonu aplikace, grafické téma atd.

Dalším nutným souborem je `build.gradle`, lokalizovaný v kořenovém adresáři. V něm jsou informace pro kompilátor, který bude aplikaci sestavovat - především její závislosti (seznam externích knihoven) a verze *Android API*, které aplikace využívá a mající následující role:

- *compileSdkVersion* - verze, proti které je aplikace kompilována a může tedy využívat její funkce (i z verzí starších - ale ne novějších), nastavena na 25 (Nougat 7.1.2)
- *minSdkVersion* - minimální podporovaná verze, na které bude aplikace fungovat, nastavena na 16 (Jelly Bean 4.1)
- *targetSdkVersion* - verze, na které byla aplikace testována (ideálně i na všech nižších mezi `minSdkVersion` a touto), nastavena na 25 (Nougat 7.1.2)

Volba těchto hodnot (především `minSdkVersion`) úzce souvisí s procentem zařízení, pro které bude aplikace dostupná, což je v roce 2017 pro API verze 16 kolem 95 %. Dále je na nich závislý rozsah funkcí, které lze při vývoji používat, kvůli čemuž byla pro `compileSdkVersion` hodnota nastavena na aktuálně nejvyšší možnou. Napříč aplikací byly často aplikovány možnosti Android *support* knihoven, které dovolují novou funkcionalitu zpětně portovat do aplikací cílených na starší varianty `Android API`. Verze těchto knihoven musí souhlasit s `compileSdkVersion`, aby nedošlo ke konfliktům.

4.3.3.2 Struktura

Při navrhování struktury bylo dbáno na to, aby byly jednotlivé související části drženy co nejvíce pohromadě. Třídy jsou tak rozmístěny do několika celků, které jsou svou funkcionalitou od ostatních oddělené s důrazem na jejich izolovanost. Každý takový celek se nachází v `Java package` a třídy v něm obsažené jsou vně této skupiny dostupné prostřednictvím importu. Snaha o tuto separaci měla za důsledek i rozdělení jednotlivých částí aplikace do vlastních *sekcí*, čímž se vždy v rámci jedné *Java package* sdružuje množství tříd spadajících pod jeden datový zdroj. Výsledná struktura spolu s popisem jednotlivých částí je zobrazena ve stromovém schématu, které následuje.

- `api` – Třídy potřebné pro komunikaci s `backend API` a získávání externích obrázků
- `database` – Přímá komunikace s databází i její struktura
- `manager` – Správa jednotlivých modelů, jejich mapování mezi databází a ostatními vrstvami
- `model` – Modelové třídy
- `section` – Sekce aplikace
  - `adoption` – Sekce související s adopcemi zvířat
  - `event` – Sekce související s událostmi v ZOO
  - `lexicon` – Sekce související s lexikonem zvířat
    - `menu` – Menu lexikonu - rozbalovací seznamy a data v nich zobrazovaná
  - `quiz` – Sekce související s vědomostním kvízem
- `LoadingScreenFragment.java` – Univerzální fragment značící právě probíhající operaci na pozadí
- `MainActivity.java` – Hlavní vstupní bod aplikace
- `SearchActivity.java` Abstraktní třída poskytující možnost vyhledávání, používaná v sekcích aplikace
- `SingleFragmentActivity.java`. Abstraktní třída, reprezentující základní funkcionalitu aktivity v aplikaci

Třída *MainActivity* je první aktivitou, která je uživateli po spuštění aplikace nabídnuta. V `manifestu` obsahuje element `<intent-filter>`, který stanovuje, jak má aktivita reagovat na příchozí `intents`. Hodnoty v něm naznačují, jak k ní má Android přistupovat - v tomto případě je aktivita vstupním bodem a neočekává tedy žádný `intent`, k čemuž přidává indikaci, že její ikona se má zobrazit v hlavním menu operačního systému. Dále funguje v módu `singleTop`, díky kterému se udržuje pouze jedna instance této třídy (pokud existuje) a nemusí se tedy opakovaně vytvářet.

Třída *LoadingScreenFragment* je jednoduchým fragmentem, který pouze zobrazuje nekonečně rotující `View ProgressBar`, informující uživatele o aktivitě na pozadí. Tento fragment je využíván většinou aktivit, které zastřešují síťové operace a je po jejich dokončení nahrazen jiným fragmentem, který obsahuje výsledek dané operace. Je nastaven jako výchozí fragment ve třídách extendujících `SingleFragmentActivity`.

Třída *SingleFragmentActivity* je rodičovskou abstraktní třídu pro mnohé aktivity ze `sekcí`. Sama extenduje *AppCompatActivity* a zprostředkovává metody vhodné pro iniciaci základní struktury, provázející vznik aktivity vlastnící jeden fragment. Její `layout` obsahuje pouze kontejner, do kterého se fragment vkládá (hned po kontrole, zda ještě neexistuje - v takovém případě se použije stávající). Výchozím fragmentem je vždy `LoadingScreenFragment` a jeho následné nahrazení si každá dědící třída implementuje sama prostřednictvím abstraktní metody `createReplacementFragment`. Další funkcí je přetěžování

metody `onBackPressed`, čímž se zamezí nežádoucí navigaci při stisknutí tlačítka „zpět“ na mobilním zařízení. Díky tomuto přetížení se pouze viditelná aktivita odstraní ze `stacku` a kontrola se předá jejímu rodiči. Poslední důležitou rolí třídy `SingleFragmentActivity` je aktualizace příznaku, zda se současná aktivita zobrazuje uživateli a je tedy ve stavu `resumed`. Znalost tohoto faktu je podstatná při nahrazování fragmentu, jelikož se tato operace provádí po doběhnutí vedlejšího vlákna, které poté předá kontrolu své rodičovské aktivitě, která již nemusí být viditelná a provádět v ní změny by vedlo k chybě.

4.3.3.3 Sdílené preference

Shared preferences je jedna z variant uchovávání dat v systému Android. Skrze `API` nabízí objekt, který pracuje nad souborem, v němž udržuje dvojice `key-value`, tedy hodnoty vázané na své klíče. Tento soubor je součástí aplikační struktury a jako takový je smazán až s jejím odinstalováním. Do té doby jsou data v něm persistovaná, přičemž se jich nedotkne ani update aplikace. Jeho služeb je využíváno v několika `sekcích`, kde je potřeba mezi jednotlivými třídami přenášet informace navázané na vyhledávání, filtrování, stránkování nebo nastavení.

Každá `sekce` má pro přístup ke svým `shared preferences` vlastní třídu a v ní množství statických metod - pro každý `key-value` pár se zde nachází po jednom getteru a setteru. Získávání dat ze souboru má vždy definované výchozí hodnoty, které se mají vrátit v případě neexistence požadovaného páru.

4.3.3.4 Zobrazení seznamu položek

Velice častým úkolem v mobilních aplikacích je vykreslování a správa seznamu homogenních položek. S ním je uživatel zvyklý manipulovat pomocí scrollování a výběru prvků, často vedoucí na jejich detail s podrobnými informacemi. Tohoto mechanismu využívají všechny `sekce` této aplikace, přičemž jim k tomu slouží Android třída *RecyclerView*. Ta vyžaduje pro svůj chod tzv. „adaptér“, který spravuje jednotlivé položky seznamu, a implementaci třídy *ViewHolder*, jež každou položku reprezentuje (udržuje v sobě její obsah a obstarává interakce s ní). Propojením těchto tří částí do jedné lze dosáhnout konzistentního a efektního kýženého výsledku.

4.3.3.5 Správa modelů

Jelikož je nutné s modely provádět některé časté operace, byla pro každý z nich vytvořena tzv. *manager* třída, nalézající se v `Java package` s názvem *manager* a zastřešující metody pro práci se svými modely. Konstruktor vypadá vždy podobně - dotáže se `ZooBaseHelper` o objekt *SQLiteDatabase*, který umožňuje čtení i zápis do databáze a vedle toho inicializuje svůj privátní list, obsahující

instance modelu. Primárním cílem je poskytnout způsob, jakým uložit data získaná z `backend API` do lokální databáze. To je prováděno hromadnou operací, která vezme onen list objektů (které do `manageru` byly již dříve vloženy zvenku) a do připraveného databázového dotazu (reprezentovaného řetězcem) jeden po druhém doplní pomocí procesu nazvaném „binding“, který je SQL ekvivalentem `NoSQL injection` popisované v sekci 3.3.6.2. Všechny takto vytvořené dotazy jsou pak najednou puštěny do databáze v rámci transakce, což dramaticky urychluje jejich ukládání.

Dalšími zásadními metodami `manageru` jsou získávání objektů z databáze - buď jednotlivě nebo po skupinách pomocí filtrů. Ty zahrnují tradiční projekci, selekci, limitování nebo řazení výsledků. Dotaz vždy vrátí kursor ukazující na první z vrácených záznamů. Tento kursor je následně zabalen do instance třídy `ZooCursorWrapper` a může se k němu přistupovat, jak je definováno v sekci 4.3.1.1.

Jednotlivé `sekce` zde implementují své specifické funkce, jako je sestavování hierarchické taxonomie v klasifikacích, výchozí filtr podle aktuálního datumu v událostech nebo sestavování dotazu pro vyhledávání v lexikonu a seznamu adopcí. Výjimečný je také fakt, že `manager` pro otázky je `singleton` a to z toho důvodu, že je seznam otázek pro každou započatou kvízovou hru udržován v paměti aplikace až do momentu startu hry nové. Otázky včetně jejich aktuálního stavu jsou v něm zakonzervované a dostupné odkudkoliv v rámci struktury kvízové `sekce`. Kromě tohoto faktu je také aplikace připravena na možnost přerušení a opětovného pokračování ve hře, jež se může v budoucnu stát její součástí. `Manager` otázek dále poskytuje metody potřebné k získání informací o jejich stavu - celkový čas strávený hádáním nebo počet správně a špatně zodpovězených otázek.

4.3.3.6 Import a zpracování dat

Získávání dat je společné pro všechny `sekce` a liší se pouze v tom, s jakými modely pracují skrze jejich `manager` a jakým způsobem tvoří požadavek na `backend API`. Tuto činnost zařizuje aktivita, která je v hierarchii každé `sekce` nejvýše postavená a aktivity i fragmenty, které za ní následují, jsou na importovaných datech závislé.

Prvním krokem je vždy vytvoření instance `DataFetcher`, která stahování zastřešuje. Následuje zaregistrování naslouchače včetně implementace metody, která bude po dokončení stahování o této skutečnosti informována. Naslouchající objekt poté kontroluje, zda se v odpovědi vrátil platný `JsonApiObject` a pokud ne, vynutí zobrazení dat z lokální databáze. Pokud ale ano, předá se její zpracování procesorovému vláknu na pozadí. To je realizováno vnitřní třídou extendující Android třídu *AsyncTask*. Přetíženy jsou v ní dvě metody - první řeší update dat a druhá, volaná po ní, spravuje nahrazení `LoadingScreenFragment` novým fragmentem, obsahujícím reálná data. V rámci první z nich se nejprve provede kontrola duplicity, popsaná v sekci 4.3.2. Pokud se jedná

o nová data, dostává se ke slovu parsování odpovědi za pomoci `GSON`. Postupně jsou vytvářeny nové instance příslušného modelu, které jsou poté vkládány do `manageru` a na závěr hromadně uloženy do databáze procesem definovaným v sekci 4.3.3.5. Po skončení této metody se předá řízení do správy té druhé, která už běží v hlavním `UI vlákně`. Ta se podívá, zda je její aktivita stále viditelná a může tedy bezpečně provést update její obrazovky.

Import a aktualizace dat v hlavním menu `sekce` Lexikon se od ostatních liší tím, že stahuje více zdrojů dat naráz. Potřebuje mít k dispozici co nejnovější data ke všem svým filtrům, podle kterých lze zvířata vybírat. Řeší to sekvenčním dotazováním, čili se nejdříve vyřeší jeden datový zdroj a až po něm se přejde na další v řadě. Pokud doběhne poslední z nich, pak teprve se může přistoupit k aktualizaci obrazovky. Tato varianta byla zvolena vzhledem ke složité správě několika asynchronně spuštěných volání prostřednictvím `Retrofit`.

4.3.3.7 Načítání obrázků

Ke stahování a zpracování obrázků zvířat slouží externí knihovna *Picasso*[46]. Ta je využívána třídou *ImageLoader*, lokalizovanou v `Java package` s názvem *api*, jelikož také vyžaduje síťovou komunikaci. `ImageLoader` je `singleton`, poněvadž je při stažení a zobrazení fotografie nutná co nejnižší latence. Proto se instance `Picasso` vytvoří jen jedna a na ní se poté už jen podávají žádosti o obrázky.

Knihovna vyžaduje pouze `URL` obrázku a `View`, do kterého se po stažení vloží - další parametry jsou volitelné. `Picasso` bohužel nenásleduje přesměrování `URL`, tedy `HTTP` návratový kód `301 - Moved Permanently` spolu s `302 - Found`. To je dáno jejich vlastním `downloader` objektem, který stahování zařizuje a musí se nahradit jiným - instancí třídy *OkHttpDownloader*. Rozměry `View` musí být při volání již známy a komponenta vykreslena na obrazovce, protože pak má `Picasso` možnost stažený obrázek škálovat a zmenšit tak jeho rozměry, jelikož originální zdrojové fotografie jsou pro potřeby aplikace ve zbytečně vysokém rozlišení. Díky přeškálování je tak rychlejší jak jeho vykreslení, tak efektivnější využití cache, kterou si knihovna sama udržuje. To je zásadní tehdy, kdy se v aplikaci zobrazuje mnoho zmenšenin obrázků najednou, přičemž se uživatel mezi nimi scrollováním rychle přesouvá.

K dosažení toho, aby si `View` o fotografii zažádalo až v momentě, kdy jsou spočítány její rozměry, se na něj navěsí metody, čekající na uskutečnění dané události. V závislosti na místě volání se používá jedna z možností `addOnGlobalLayoutListener()` a `addOnLayoutChangeListener()`, jež se volá ve chvíli, kdy je dokončeno vykreslení obrazovky. Až poté je možné obrázek vložit, čemuž ještě předchází odstranění aktivního naslouchající metody, poněvadž už není potřeba.

[46] `http://square.github.io/picasso/`

Uživatelskou přívětivost výrazně zvýší možnost zaplnit obsah `View placeholderem`, který se poté nahradí skutečnou fotografií. Také chyba při stahování je indikována nahrazením výchozího `placeholderu` symbolem rozbitého obrázku. Stejně tak je důležitá možnost informovat na problém při stahování, což se děje pomocí chybového logování.

4.3.3.8 Vyhledávání

Vyhledávání je realizováno abstraktní třídou *SearchActivity*, která dědí ze `SingleFragmentActivity` a funguje tak jako jako mezivrstva, která k ní přidává další možnosti. Jejím cílem je zobrazit uživateli menu s možností začít vyhledávat v příslušném seznamu. Na typu ani obsahu onoho seznamu nezáleží, samotnou funkcionalitu si každá extendující třída implementuje sama. Úlohou `SearchActivity` je pouze čekat na události a reagovat na ně, načež jejich obsluhu abstrahuje. V rámci `layout` využívá elementu `<menu>`, které reprezentuje třídu, integrující svůj obsah do `toolbaru` aplikace.

Po kliknutí na položku vyhledávání v menu se na obrazovce rozbalí klávesnice a uživatel může začít psát požadovaný řetězec (toto zastřešuje použitá Android třída `SearchView`). Po jeho odeslání se klávesnice skryje, proběhne update seznamu na základě nově získaných dat a řetězec se uloží do `shared preferences` příslušné `sekce`, aby mohl být brán v úvahu při návratu na seznam z detailu jeho položky. Vyhledávaný řetězec má uživatel možnost skrze menu sám smazat, nebo pokud se přesune o úroveň výše ve `stacku`, je mu smazána automaticky.

Vyhledávání je použito v `sekcích` „Adopce zvířat“ (4.3.4.4) a „Lexikon“ (4.3.4.1), kde se díky němu dají filtrovat zvířata podle jejich názvu. `Backend API` vrací výsledky vyhledávání korektně, ale `SQLite` nepodporuje vyhledávání s ignorováním diakritiky. To je ovšem důležité podporovat stejně jako to s diakritikou, jelikož uživatelé jsou zvyklí psát požadovaný řetězec bez ní. Z toho důvodu byla vytvořena dvě nová pole s názvem „name_no_accents“ v tabulkách `adoptions` a `animals`, která obsahují název zvířete bez diakritiky a spolu s původními názvy jsou používány při porovnávání s hledaným řetězcem. Pro uživatele se tak výsledek vyhledávání tváří stejně ať už použije diakritiku nebo ne. Realizace těchto nových sloupců proběhla za pomoci Java třídy *Normalizer*, jež řetězec rozloží na jednotlivé znaky, po čemž se následně smažou veškeré výskyty těch, co nejsou písmeny (tedy v případě českého jazyka háčky a čárky).

4.3.3.9 Stránkování

`Sekce` „Adopce zvířat“ (4.3.4.4) používá při zobrazování seznamu adopcí stránkování. Projevuje se tak, že se při scrollování seznamem postupně donačítají další záznamy v závislosti na tom, jak daleko v seznamu se uživatel nachází. Velikost jedné stránky (což je počet záznamů získaných v rámci jed-

noho dotazu na databázi) byla postupných testováním stanovena na dvaceti a je reprezentována konstantou. Aktuální stránka je ukládána v rámci `shared preferences` a resetována na jedničku vždy po navigaci uživatele zpět ze seznamu do hlavního menu aplikace.

Zjištění momentu, kdy je třeba přejít na další stránku, probíhá neustálou kontrolou aktuálního viditelného prvku seznamu na obrazovce. Každé uživatelské scrollování obrazovky znamená vyvolání události, na kterou reaguje metoda porovnávající index onoho viditelného prvku se současným maximem - jakmile se tyto hodnoty rovnají, znamená to, že je uživatel na konci seznamu a je třeba zvýšit aktuální stránku. To je okamžitě provedeno, načež se získají a zobrazí čerstvá data. Poté je třeba index v seznamu ještě posunout o jeden prvek níž, aby stránkování působilo hladce.

4.3.4 Sekce aplikace

4.3.4.1 Sekce „Lexikon“

Nejrozsáhlejší strukturu má lexikon zvířat. Mezi úvodní obrazovkou aplikace a detailem zvířete je třeba projít dalšími dvěma aktivitami, které uživateli při výběru vypomáhají. Nejdříve je třeba vybrat jeden z nabízených filtrů, pomocí kterého se nabízený seznam zvířat zúží. Tyto filtry jsou rozděleny do dvou skupin: první obsahuje kompletní taxonomii (hierarchii všech tříd a řádů), druhá se soustředí na atributy zvířat (druhy potravy a místa výskytu - kontinenty, biotopy nebo umístění v ZOO Praha, spolu s informačním počtem zvířat, které tuto hodnotu splňují). Po zvolení jednoho z nich je uživatel přenesen na seznam vybraných zvířat, ve kterém se může libovolně pohybovat. Z něj už přechází na detail zvířete se všemi jeho vlastnostmi. Hierarchie tříd, ve které jsou vidět závislosti mezi nimi, je v následujícím schématu.

- `LexiconMenuActivity` ..... Zastřešuje import dat pro filtry.
  - `LexiconMenuFragment` ..... Uchovává fragmenty s filtry.
    - `LexiconMenuTaxonomyFragment` ..... Zobrazuje filtr pro výběr z klasifikací.
    - `LexiconMenuFiltersFragment` ..... Zobrazuje ostatní filtry.
- `LexiconListActivity` ..... Importuje zvířata na základě aktivních filtrů.
  - `LexiconListFragment` ..... Zobrazuje seznam vyfiltrovaných zvířat.
- `AnimalDetailActivity` ... Uchovává fragment s detailem jednoho zvířete.
  - `AnimalDetailFragment` ..... Reprezentuje jedno zvíře.
    - `ImageDialogFragment` ..... Zobrazuje zvětšeninu obrázku zvířete.
- `AnimalPagerActivity` ..... Spravuje více fragmentů se zvířaty.
  - `AnimalDetailFragment` .....
- `LexiconPreferences` ... Správa `shared preferences`, platných pro tuto `sekci`.
- `menu` ..... Obsahuje třídy pro zobrazení filtrů v menu.
  - `LexiconExpandableListDataMapper` .... Sestavuje struktury naplněné daty pro filtry.
  - `LexiconExpandableListAdapter` ..... Vytváří z dodaných dat seznam filtrů.

Prvním krokem je zobrazení menu lexikonu, vyžadující dvě nosné třídy - *ViewPager* a *TabLayout*. První z nich má možnost udržovat skupinu fragmentů, mezi kterými se lze tzv. *swipe* dotykem přepínat (simuluje se tak otáčení stránek knihy). Druhá její možnosti ještě rozšiřuje o schopnost ke každému ze skupiny fragmentů zobrazovat záložku, indikující místo, na jakém z nich se uživatel nachází. Díky spojení těchto tříd je dosaženo požadovaného efektu, kdy jsou na obrazovce nabídnuty různé skupiny filtrů lexikonu, mezi nimiž lze snadno přepínat. Každá skupina musí být co možná nejaktuálnější, proto zobrazení menu předchází import dat z `backend` serveru, prováděný synchronně (zdroje se tedy aktualizují jeden po druhém) v `LexiconMenuActivity`.

Obě skupiny filtrů je poté třeba naplnit daty. Ta jsou zobrazována díky třídě *ExpandableListView*, rozdělující filtry do podskupin, kde každá z nich obsahuje seznam svých hodnot. Podskupinu je možné dotykem rozbalit a poté vybrat konkrétní filtr. Každá záložka vlastní fragment, který toto realizuje s následující posloupností kroků:

- Získání správně namapovaných dat metodou třídy *LexiconExpandableListDataMapper*
- Vytvoření rozbalovatelného seznamu pomocí třídy *LexiconExpandableListAdapter*, kde musí být definováno chování při interakcích s ním
- Napojení vzniklého adaptéru na instanci `ExpandableListView`, která je schopna data v něm zobrazit

- Tvorba naslouchajících metod, které čekají na události vzniklé interakcemi s tímto `View`

Při výběru konkrétního filtru je nutné jeho název i hodnotu sestavit, jelikož je není možné ze seznamu získat přímo. V naslouchající funkci jsou k dispozici pouze pozice skupiny a jejího potomka, který byl vybrán. S využitím těchto indexů lze přistoupit k původnímu zdroji dat, použitém adaptérem, a v něm hodnoty nalézt. Vzhledem k tomu, že je uživateli také poskytnuta možnost zobrazit zvířata z konkrétní taxonomické třídy, je během sestavování brán ohled na to, zda byla zvolena první položka v seznamu - ta totiž obsahuje dynamicky vloženou hodnotu, která právě toto umožňuje. Předání sestaveného filtru aktivitě ze třídy *LexiconListActivity* je posledním úkolem úvodního menu.

Následuje zobrazení části lexikonu podle dodaného filtru. Ten se vždy po výběru ukládá do *LexiconPreferences*, aby byl aktivní i po návratu z detailu zvířete. Jeho využití při dotazu na `backend API` je usnadněno díky třídě *LexiconQueryBuilder*, která se nachází mezi ostatními modely v `Java package` s názvem `model`. Její role spočívá v zaobalení všech `query` parametrů, které `API` umožňuje pro lexikon použít, do jednoho objektu, jež se následně předá jako argument do metody v `DataFetcher` a ta ho rozloží na jednotlivé parametry. Každý `query` parametr je samostatný atribut této třídy a jeho nastavení se tedy rovná zavolání příslušného setteru. Seznam zvířat, který je výsledkem aplikace vybraných filtrů, je zobrazen pomocí `RecyclerView` a seřazen vzestupně podle abecedy. V něm je třeba dbát zvýšené péče ohledně rychlosti načítání jednotlivých položek, jelikož každá z nich obsahuje zmenšeninu obrázku zvířete, jež je náročná na získání i vykreslení. Z toho důvodu se kromě přednastavené cache ve třídě `ImageLoader` nastavuje ještě jedna přímo na seznamu, která je spojena s přednačítáním položek seznamu - při pohybu v něm se tedy na pozadí stahují a připravují následující záznamy. Po klepnutí na jakýkoliv z nich se uživatel přesune na detail zvířete.

Ten o něm obsahuje všechny dostupné informace v přehledném, scrollovacím fragmentu. Jeho rodičovská aktivita opět implementuje variantu využití třídy `ViewPager`, jejímž prostřednictvím umožňuje správu všech zvířat ve vyfiltrovaném seznamu, mezi kterými lze dotykem přepínat. Detail poskytuje i obrázek, jež je schopen po klepnutí na něj vyvolat instanci třídy *ImageDialogFragment*, což je dialogové okno s jeho zvětšeninou. Sám fragment s detailem se také může chovat jako dialog, jelikož extenduje třídu *DialogFragment* - takto může být využit v kvízové `sekci`, což je podrobně popsáno v 4.3.4.2. Detail disponuje i možností zavolat `backend API` s žádostí o dodání zvířete, pokud lokálně neexistuje (a nebylo tedy získáno hromadným dotazem na lexikon z `LexiconListActivity`) - tato skutečnost může nastat v momentě, kdy je fragment volán ze `sekce` s adopcemi, což lze nalézt v 4.3.4.4.

4.3.4.2 Sekce „Kvíz“

Výchozím bodem vědomostního kvízu je jeho úvodní menu, sloužící jako rozcestník. Hierarchie této `sekce`, která je skrze menu dostupná, je zobrazena zde:

- `QuizMenuActivity` ..... Rozcestník.
  - `QuizActivity` ..... Iniciace kvízu.
    - `QuestionFragment` ..... Reprezentuje jednu otázku.
  - `QuizResultActivity` ..... Výsledek kvízu.
  - `QuizResultListActivity` ..... Žebříček výsledků kvízu.
  - `QuizSettingsActivity` ..... Formulář s nastavením parametrů kvízu.
  - `QuizHelpActivity` ..... Nápověda.
  - `QuizPreferences` Správa `shared preferences`, platných pro tuto `sekci`.

Průběh kvízu, jeho účel a popis parametrů jsou popsány ve třídě *QuizHelpActivity*, obsahující nápovědu. Parametry lze definovat na základě vlastních preferencí, k čemuž slouží aktivita v *QuizSettingsActivity*, obsahující formulář se standardními editovatelnými prvky. Ty jsou ukládány do *QuizPreferences*, aby byly dostupné po celou existenci aplikace. Možnosti jsou takovéto:

- Časový limit pro zodpovězení jedné otázky (možná je libovolná kladná hodnota).
- Počet otázek v jedné hře (na výběr je několik předdefinovaných možností, dostupných v roletovém menu).
- Uživatelské jméno, pod kterým budou výsledky zaznamenávány (jedná se o textové pole).

Samotný kvíz je zastřešován třídou *QuizActivity*. Ta nejprve stáhne definovaný počet otázek z `backend API`, které poté uloží do `singleton manageru`, aby byly dostupné i při konfiguračních změnách. Pokud se už tento krok nepovede úspěšně dokončit (např. proto, že je síťové připojení nedostupné), uživatel je o tom spraven prostřednictvím krátké informační bubliny se zprávou, kterou poskytne Android třída *Toast*. Bez otázek nemá kvíz smysl a persistence otázek neexistuje, jelikož je pokaždé vyžadována jejich co nejvíce originální náhodná skladba, kterou zajišťuje `backend` server.

V případě úspěšného získání otázek se vytvoří první instance třídy *QuestionFragment*, která obsahuje výchozí otázku. Číslo otázky a spolu s ním i schopnost v kvízu iterovat má na starosti rodičovská aktivita. Fragment aktuální index také obsahuje a k němu přidává dosavadní skóre, průběh odpočítávání času, text otázky (po němž může následovat doprovodný obrázek) a seznam odpovědí. Odpočet je realizován extendováním Android třídy *CountDownTimer*, která obdrží délku intervalu pro odpověď v milisekundách a je napojena

na *ProgressBar*, který poskytuje pro odpočet vizualizaci. Každou půlsekundu se interně volá metoda, která časovou hodnotu (a spolu s ní i vizuální informaci) aktualizuje. Tato metoda běží na pozadí a v případě ukončení jejího rodičovského fragmentu je třeba jí ukončit také. Doběhnutím intervalu bez reakce uživatele skrze volbu odpovědi se otázka považuje za špatně zodpovězenou.

Pokud se mu ale v daném intervalu povede odpovědět, je provedena kontrola její správnosti. V každém případě se seznam odpovědí nahradí dalšími možnostmi, které výsledek kontroly zobrazí. Validitu a korektnost otázky má uživatel možnost zpochybnit prostřednictvím tlačítka, které zobrazí dialogové okno, vyzývající k potvrzení dané akce. Pokud je operace potvrzena, otázka je označena a tento příznak bude sloužit jako indikátor její pochybného stavu, což by mělo být v budoucnu bráno v úvahu, jak popisuje sekce 5.5. Další možností je zobrazení detailu zvířete, kterého se otázka týká. K tomu je využita třída `AnimalDetailFragment` a její schopnost chovat se v případě potřeby jako dialogové okno. Toto okno se efektně rozbalí a zachovává všechny vlastnosti detailu, včetně možnosti v něm scrollovat. Jedním dotykem kamkoliv v okně ho lze opět zavřít, vrátit se tak ke kvízu a pokračovat posledním tlačítkem na další otázku nebo na výsledek kvízu.

Ten obsahuje statistiky právě dokončené hry, jako je počet špatných odpovědí nebo skóre, což je hodnota vycházející z celkového počtu otázek, jejich spočítaných vah a správných odpovědí. Tyto hodnoty jsou spolu s dalšími uloženy do databáze skrze novou instanci modelu *QuizResult*. Díky tomu je historické výsledky možné zobrazovat ve třídě *QuizResultListActivity*, která využívá listovací schopnosti `RecyclerView` k tomu, aby vykreslila pořadí nejlepších dosavadních skóre. Tento žebříček je dostupný z menu, které třída *QuizResultActivity* nabízí, nebo z hlavního rozcestníku kvízu.

4.3.4.3 Sekce „Události“

Tato část je oproti předchozím značně jednodušší, hierarchie tříd následuje:

- `EventListActivity` ..... Importuje události na základě aktivních filtrů.
  - `EventListFragment` ..... Zobrazuje seznam vyfiltrovaných událostí.
- `EventDetailActivity` ..... Uchovává fragment s detailem jedné události.
  - `EventDetailFragment` ..... Reprezentuje jednu událost.

Prvním krokem je opět snaha o získání nejnovějších dat z `backend API`, provázená automatickým filtrem událostí podle aktuálního datumu ve formátu *ISO 8601*, tedy např. „2017-06-06’T’12:20:25“. Po jejich získání je vykreslen seznam položek, splňujících požadavek a přes který je možné zobrazit detail některé z nich. Ten je obsažený ve třídě *EventDetailFragment*, která se vyznačuje zpracováním obou dostupných datumů události (její začátek a konec) do srozumitelné podoby. To zahrnuje parsování řetězce, převedení původního

formátu do standardního českého ve tvaru „6. 6. 2017, 12:20“ a odstranění časové informace v případě, že je datum zadáno na jednodenní událost od půlnoci do půlnoci. Takový případ by neodpovídal otevírací době ZOO Praha a zobrazený čas by byl nesmyslný.

4.3.4.4 Sekce „Adopce zvířat“

Ani poslední `sekce` s adopcemi neobsahuje mnoho nového, skladba tříd v ní je takováto:

- `AdoptionListActivity` .... Importuje adopce na základě aktivních filtrů.
  - `AdoptionListFragment` ..... Zobrazuje seznam vyfiltrovaných zvířat k adopci.
- `AdoptionPreferences` .. Správa `shared preferences`, platných pro tuto `sekci`.

Po aktualizaci dat (s případným vyhledávacím řetězcem, který je popsán v sekci 4.3.3.8 a jež tato `sekce` implementuje) je za pomoci `RecyclerView` zobrazen seznam zvířat určených k adopci. Tento list používá možnost stránkování, definované v sekci 4.3.3.9, jelikož obsahuje velké množství prvků, jejichž okamžité načítání a zobrazování by bylo neefektivní. Předpokládá se zde totiž využívání vyhledání konkrétního zvířete podle jeho názvu namísto náhodného prohledávání seznamu.

Detail adopce je zde realizovaný skrze třídu `AnimalDetailFragment`, se kterou je adopce propojená pomocí jejího identifikátoru v lexikonu, jak bylo zmíněno v sekci 3.3.4. Tato vazba tedy nemusí existovat, na což je uživatel upozorněn pomocí `Toast` bubliny. V případě fungující vazby je cílový fragment zobrazen s pomocí třídy *AnimalDetailActivity*, která nepodporuje žádný pohyb v seznamu zvířat, jak tomu bylo při vstupu na detail v `sekci` lexikonu, ale spravuje pouze jednu instanci.

4.3.5 Zobrazení

4.3.5.1 Struktura

Zobrazování dat uživateli je v systému Android řešeno pomocí tzv. *layouts*[18], které definují rozvržení prvků na každé obrazovce, jež bude uživateli nabídnuta, prostřednictvím `XML`. Každá ze zobrazovaných komponent je reprezentována Android třídou, v rámci které lze volit její atributy. Tlačítko tak můžou být definovány rozměry, text na něm (včetně velikosti fontu) nebo umístění na obrazovce. Komponenty se následně skládají do skupin, které je možné hromadně organizovat, navzájem jim určovat pozice relativně k sobě a specifikovat váhy, jež budou prvky v nich obsažené mít v rámci jejich vykreslení.

Každá aktivita nebo fragment, který má za úkol nějaký obsah zobrazovat, má svůj vlastní soubor, ve kterém se patřičné části nachází. Soubory

jsou izolované, ale mohou využívat možnost zakomponovat do sebe cizí strukturu. Toho je využito při konstruování obrazovky pro kvízovou otázku - do její spodní části je nejprve nahrán seznam odpovědí, jež se po vypršení časového limitu nebo zodpovězení otázky nahradí novou strukturou, obsahující výsledek a možnosti, kterak pokračovat. Vlastnosti začleňovaných souborů lze dynamicky měnit a přizpůsobovat je tak současným požadavkům. Tohoto systému taktéž hojně využívá `RecyclerView`, kterému stačí pro každý ze svých prvků jeden soubor, který poté opakovaně aplikuje na všechny zobrazované záznamy.

`Layouts` se nachází v jednom společném adresáři, s výjimkou souboru určenému pro `toolbar` menu aplikace, v němž jsou prvky pro vyhledávání a který má vlastní, oddělený adresář.

Napříč aplikací jsou využívány i další zdroje vyjma těchto souborů. Veškeré řetězce jsou například umístěné zvlášť a nejsou tak přímo součástí `layouts`, jelikož by tak jejich vyhledávání kvůli případným změnám bylo složité. Kromě toho je ideální je mít na jednom místě kvůli dotazování na ně z různých částí aplikace nebo i pro jejich případnou budoucí lokalizaci do cizích jazyků. Důležitým zdrojem jsou také obrázky, vytvořené vždy v několika velikostech pro odlišné typy zobrazovacích zařízení, jež mají různé rozlišení a potřebují tak mít obrázky s odpovídající kvalitou. Pro jejich tvorbu byl využit nástroj Android Studia *Asset Studio*, který na základě předpřipravených clipart souborů (tedy jednoduchých obrázků) nebo vlastních fotografií tyto výsledné varianty sám vygeneruje.

4.3.5.2 Material design

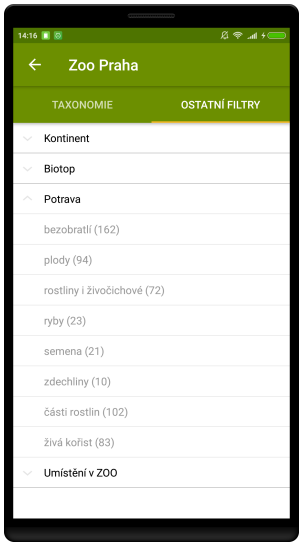
Vzhled aplikace, barevné sladění, rozložení jednotlivých komponent a jejich vlastnosti, to vše probíhalo na základě návrhů definovaných v *Material design*[47]. Jedná se o vizuální jazyk, definovaný společností Google v roce 2014 a oficiálně uvedený do mobilních zařízení používajících systém Android v rámci jeho verze 5.0. Jazyk je doprovázený detailní, přehlednou a precizně navrženou dokumentací, obsahující mnohé příklady popisovaných mechanismů. Sám o sobě není vázaný pouze na mobilní zařízení, ale zakládá si na univerzálnosti. Popisuje i pohyb prvků, logické řazení v seznamech, chování světla při kontaktu s materiálem nebo role barev, obrázků a typografie. Vzhledem k podpoře starších mobilních zařízení byl v aplikaci `Material design` použit skrze `support knihovnu`.

Realizace konceptů tohoto jazyka se vztahuje na vzhled všech obrazovek v aplikaci. U seznamů jsou definovány velikosti položek, jejich písmo a rozložení, vzdálenosti od okrajů i od sebe navzájem. U tlačítek je dbáno na to, aby byly dostatečně velké a text na nich čitelný. Nesmí také uživateli ztěžovat manipulaci s nimi a naopak musí být snadno viditelné, neztrácet se mezi okolními prvky. Podobnými doporučeními se řídí všechny komponenty.

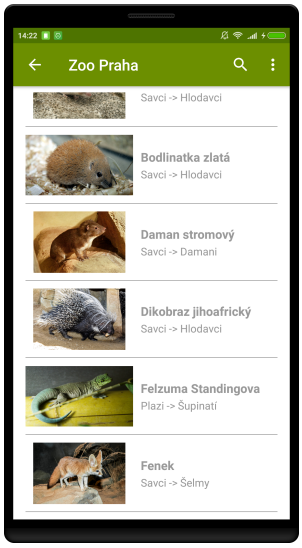
[47] `https://material.io/`

Implementaci značně usnadňuje využití stylů, které v sobě sdružují více vizuálních vlastností a jež se poté dají hromadně aplikovat na jednotlivé prvky `layoutu`. Díky tomu není potřeba stejné styly definovat opakovaně na různých místech a naopak je lokalizovat do jednoho výchozího bodu. Styly podporují dědičnost a lze tak mezi nimi vytvářet hierarchický systém.

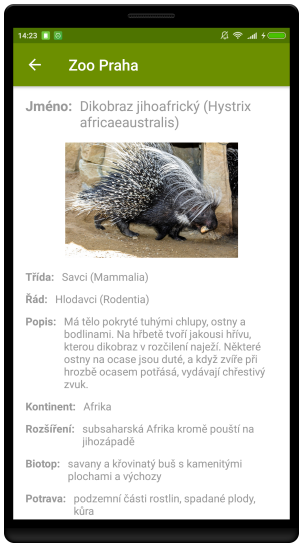
Snímky obrazovek zobrazují menu lexikonu s výběrem filtru v 4.6, seznam vybraných zvířat v 4.7, detail zvířete v 4.8 a kvízovou otázku v 4.9.



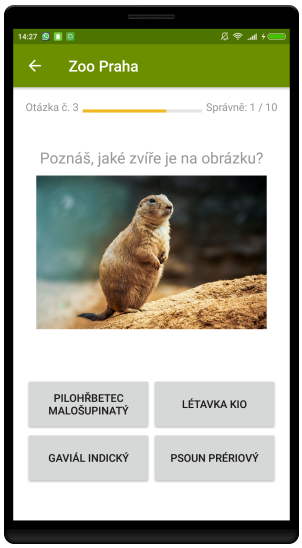
Obrázek 4.6: Ukázka výsledné obrazovky „menu lexikonu“ v aplikaci



Obrázek 4.7: Ukázka výsledné obrazovky „seznam zvířat“ v aplikaci



Obrázek 4.8: Ukázka výsledné obrazovky „detail zvířete“ v aplikaci



Obrázek 4.9: Ukázka výsledné obrazovky „kvízová otázka“ v aplikaci

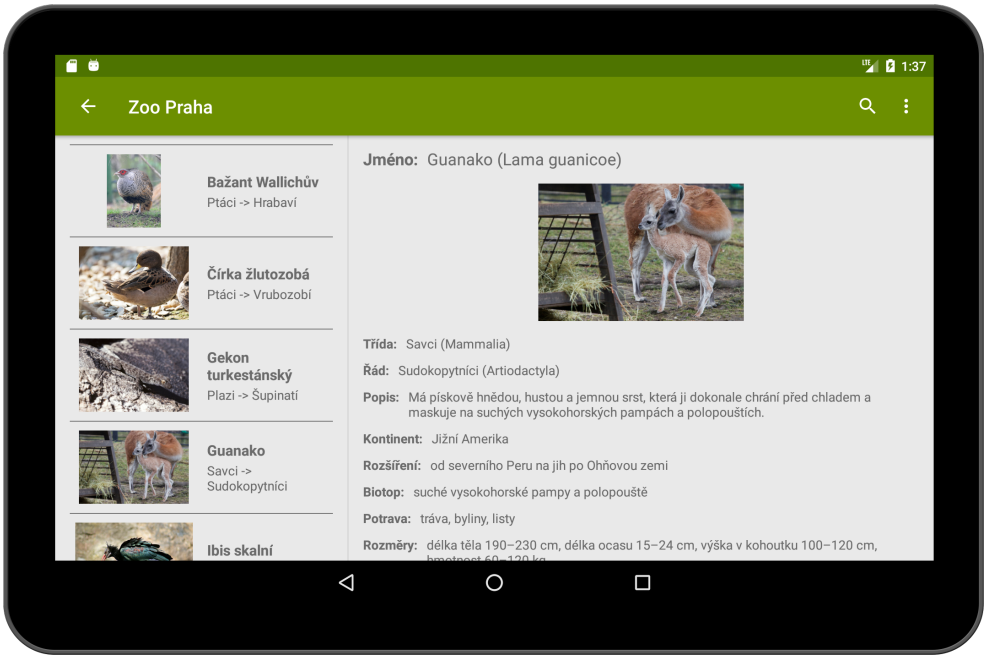
4.3.5.3 Podpora různých zařízení

Všechny pevně stanové rozměry zobrazovaných prvků jsou definováno pomocí jednotek *dp*, jež jsou nezávislé na hustotě pixelů cílového zařízení[19]. Ne jejím základě se poté dynamicky škálují tak, aby vždy dodržovaly stejné fyzické rozměry. Samotný `dp` je stanoven jako velikost jednoho pixelu na obrazovce s hustotou 160 *dpi* (bodů na čtvereční palec). Navíc jakýkoliv text má svou

velikost definovánu jednotkami *sp*, které kromě vlastností `dp` bere v potaz i uživatelské preference ohledně velikosti písma.

Tablety se vyznačují většími rozměry a tedy často i vyššími rozlišeními, než chytré telefony. Tuto skutečnost aplikace zužitkovává a nabízí tak uživatelům její pohodlnější využívání. Kromě rozložení prvků na obrazovce, které je strukturována právě s ohledem na tablety, se toto pohodlí projevuje ve všech seznamech, které vedou na detail některé ze svých položek. Na menších zařízeních je tento přechod standardně řešen zavoláním nové aktivity, která tu stávající nahradí na vrcholu `task` zásobníku. Rozměry tabletu ale umožňují větší flexibilitu a s ní možnost ukázat v rámci jedné obrazovky jak seznam na levé straně, tak vybraný detail položky na straně pravé. To je možné díky tzv. *referenci*, což je typ Android zdroje nacházejícím se ve vlastním souboru a poskytujícím odkazy na jiné zdroje. Volba `reference` je podmíněna minimální šířkou obrazovky v pixelech, kterou musí zařízení splňovat. Pokud je dostatečně vysoká, je zvolena `reference` na `layout` o dvou částech, připravený poskytnout místo dvěma různým obsahům.

Následnou realizaci obstarává každá rodičovská aktivita, schopná posílat `intents` dalším třídám. Implementuje Java interface, definovaný ve fragmentu, jež aktivita zastřešuje a který danou implementaci od své rodičovské aktivity vyžaduje. Fragment má ve správě samotné `RecyclerView` a pozná tím pádem, jaká z jeho položek byla uživatelovým dotykem zvolena. Skrze metodu interface tuto informaci o výběru konkrétní položky předá aktivitě a ta učiní finální rozhodnutí na základě typu zvoleného `layout` - pokud má dvě části, sama vytvoří nový fragment s detailem položky a vloží ho na pravou stranu obrazovky. Pokud je k dispozici pouze jedna část, vytvoří se `intent` mířený na další aktivitu, která detail vykreslí. Výsledek je zřejmý z obrázku 4.10.



Obrázek 4.10: Využití velikosti tabletu pro zobrazení dvou fragmentů naráz

Vzhledem k širokému rozsahu zařízení, která mají Android jako svůj základní operační systém, není snadné pokrýt všechna. Různé verze, velikosti, rozlišení, rozšíření a uživatelské preference jakýkoliv návrh značně znesnadňují. Podpora co nejširšího pole různých typů je tak založena na neustálém procesu optimalizace.

4.4 Testování

4.4.1 Testovací zařízení

Během testování bylo často využíváno emulovaných zařízení, které lze vytvářet prostřednictvím Android Studio. Možnosti při tvorbě jsou rozsáhlé a zahrnují kromě typu zařízení i operační systém, hardwarové parametry nebo simulují speciální případy, jako je např. stav sítě. K takto vytvořenému zařízení lze následně přistupovat jako k fyzickému a testovat na něm aplikaci bez výraznějších omezení.

Emulátoru bylo často využíváno kvůli jeho variabilitě a možnosti vytvořit si specifické testovací prostředí, které by se jinak ve fyzické podobě složitě shánělo. Lze tak pokrýt množství různých verzí operačního systému Android a zkoušet na nich chování programové části aplikace, nebo sledovat chování grafiky a rozvržení jednotlivých obrazovek v závislosti na rozlišení a velikosti zařízení. Vedle virtuálních byly použity i přístroje fyzické, které jsou buď ve

vlastnictví autora práce nebo byly testovacím účelům se svolením propůjčeny. Jejich přehled je v tabulce 4.3.

Název	Typ	OS	Rozměr	Rozlišení	CPU	RAM
Xiaomi Mi 4	fyzické	6.0.1	5.0"	1080x1920 px	Quad core, 2.5 GHz	2 GB
Asus Memo Pad 7	fyzické	4.4.2	7.0"	800x1280 px	Quad core, 1.86 GHz	1 GB
Samsung Galaxy Alpha	fyzické	5.0.2	4.7"	720x1280 px	Octa core, 1.8 GHz	2 GB
Nexus 5	virtuální	5.1	4.95"	1080x1920 px	Dual core	1 GB
Nexus 7	virtuální	6.0	7.02"	1200x1920 px	Dual core	1 GB
Nexus S	virtuální	4.1	4.0"	480x800 px	Dual core	500 MB
Pixel XL	virtuální	7.1.1	5.5"	1440x2560 px	Dual core	2 GB

Tabulka 4.3: Použitá testovací zařízení

4.4.2 Průběh

V kódu aplikace se na styčných místech nacházejí logované zprávy, informující o důležitých skutečnostech a místech i čase, kde nastaly. Toho je dosaženo skrze Android třídu *Log*, která má pro každou úroveň zprávy jednu metodu - ať už se jedná o chybové hlášení nebo debuggovací informaci[20]. Vedle sledování těchto událostí bylo hojně využíváno debuggovacího nástroje, který je zabudován v Android Studio a který po napojení na testovací zařízení umožňuje krokovat jednotlivé příkazy v kódu a tím pádem provádět instrukce postupně, spolu se sledováním aktuálního stavu proměnných a struktur v každém kroku. Vedle toho je v Android Studio přítomný i *linter*, což je nástroj na kontrolu kódu, který po důkladné inspekci projektu navrhuje možné změny a vylepšení, jakými ho lze optimalizovat.

Všechny tyto mechanismy jsou zásadní pro samotný vývoj, ale testování se kromě autora práce zúčastnili i testeři, vybraní z jeho blízkého okolí. Ti buď obdrželi testovací zařízení nebo byla aplikace nainstalována přímo na to

jejich, načež procházeli všechny popsané `sekce` a zkoušeli funkcionalitu i mezní případy použití. Jejich připomínky a zkušenosti byly do vývoje dynamicky zanášeny.

Pro testování byl zásadní funkční server, na který se aplikace mohla připojovat - toho bylo nejčastěji docíleno jeho spuštěním v lokální síti na soukromém počítači, na který se aplikace dotazovala pomocí pevně nastavené IP adresy v jejím kódu, která se liší i v závislosti na tom, zda se jedná o virtuální nebo fyzické testovací zařízení.

4.4.3 Automatické testy uživatelského rozhraní

K tomuto typu testů slouží knihovna *Espresso*[48], která umožňuje simulovat průchod uživatele aplikací a prostřednictvím toho kontrolovat správnou funkci rozhraní.[21] Realizovány jsou všechny typy interakcí založených na dotyku obrazovky, jako je scrollování, swipe nebo samostatný stisk. Tyto testy lze buď spouštět každý zvlášť nebo hromadně při nějaké události, jakou je zkompilování kódu. Pokryty byly všechny `sekce` aplikace, přičemž v rámci každého testu byla snaha o zahrnutí co největšího počtu operací.

Jednotlivé testy se nachází ve vlastních třídách, které jsou navíc ve zvláštním adresáři, oddělené od zbytku kódu. Ke svému chodu vyžadují množství závislostí, které je třeba zanést do `build.gradle`. Každý úkon v testu sestává ze tří komponent:

1. ViewMatcher - vyhledá `View` v aktuálně zobrazené hierarchii
2. ViewAction - na nalezeném `View` provede nějakou akci
3. ViewAssertion - výsledek akce je porovnán s očekáváním a zvalidován

Posloupnost úkonů, sestavených z těchto částí, tvoří jeden test. S jejich rostoucím počtem stoupá i jeho složitost. Kontrolovat a validovat lze existenci určitých prvků, definovaných identifikátorem, obsahem nebo umístěním. Nevýhodou `Espresso` je obtížné nastavení akcí na některých prvcích, jakými jsou např. `Spinner` nebo `ExpandableListView`.

4.5 Nasazení

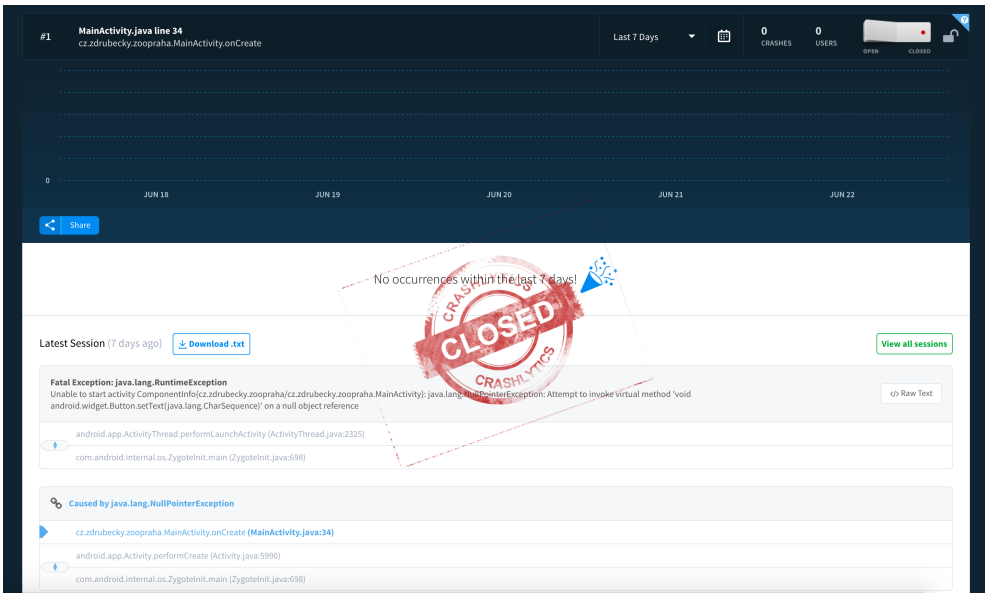
4.5.1 Sběr chybových hlášení

Finálnímu nasazení aplikace do produkčního prostředí předchází její provázání s analytickým nástrojem, který se bude starat o sběr chyb a výjimek ústící v její pády. Díky takovému mechanismu bude snadné rychle reagovat na vzniklé potíže a dynamicky je řešit i opravovat. Ideálním prostředkem k takovýmto úkolům je *Crashlytics*[49] od společnosti *Fabric*. Ten se s aplikací sváže

[48] `https://google.github.io/android-testing-support-library/docs/espresso/`
[49] `https://docs.fabric.io/android/crashlytics/overview.html`

pomocí vlastní knihovny a `pluginu` v Android Studio, který vlastníka kódu provede procesem propojení s jeho uživatelským účtem, který si ve webovém rozhraní `Fabric` vytvořil. Jakákoliv chyba se tím pádem v aplikaci vyskytne bude díky `Crashlytics` zachycena a odeslána do onoho účtu k prozkoumání, včetně upozornění jejího majitele emailem.

Účet obsahuje rozsáhlé možnosti správy všech aktuálních i vyřešených hlášení. Zprávy tu lze třídit, označovat jejich závažnost, zkoumat závislost na operačním systému a sledovat historické statistiky všech výskytů. Díky tomu je aplikace připravena podstoupit uvedení do produkčního prostředí, poněvadž je sledování jejího chování okamžité a flexibilní[22]. Webové rozhraní `Crashlytics` je na obrázku 4.11 .



Obrázek 4.11: Ukázka práce s webovým rozhraním Crashlytics

4.5.2 Google Play

Přirozenou volbou místa pro umístění výsledné aplikace je Google Play[50], distribuční služba spravovaná samotnou společností Google a organizovaná centrála pro miliony aplikací. Zde je možné si za menší poplatek zřídit vývojářský účet, který slouží jako knihovna vlastních verzí aplikací a analytický nástroj v jednom.

Než se aplikace dostane mezi koncové uživatele, musí splňovat několik požadavků. Primárně je třeba ji podepsat vlastním certifikátem, který s ní autora jednoznačně spojuje. Díky tomu je možné nabízet uživatelům k instalaci jaké-

[50] `https://play.google.com/store?hl=en`

koliv nové verze s jistotou, že nejsou podvržené. Dále je nutné aplikaci popsat, aby byla snadno vyhledatelná a nepůsobila odbytým dojmem. K tomu slouží snímky jejích obrazovek, specifikace typu a kategorie, kam spadá, hodnocení nezávadnosti obsahu, kontaktní email nebo země, na jejichž trhy cílí. Množství technických detailů (jako jsou verze podporovaných zařízení) lze získat přímo z aplikace a není třeba je vyplňovat zvlášť.

Aplikace byla nasazena jakožto alfa verze a není tedy finální. Může být ještě nestabilní a projde si dalším vývojem, než dosáhne oficiálního uvedení. Dostupná je na adrese `https://play.google.com/store/apps/details?id=cz.zdrubecky.zoopraha`.

Kapitola 5

Budoucnost vývoje

Jak `backend` část tak mobilní aplikace mají široký prostor k růstu. Veškeré možnosti nebyly v rámci této práce zdaleka využity a potenciál koncové aplikace bude i nadále využíván. Kromě technických vylepšení by prvotní kroky vedly k následujícím funkcím:

5.1 Aktualizace Open dat

Primárním cílem je dosáhnout pokroku ohledně aktualizace informací v Open datech ze strany ZOO Praha a na ní navázané synchronizaci s `backend` databází. Do doposud kvůli pracovní vytíženosti a jinak zaměřeným úkolům nebylo možné, čili se většinu času pracovalo se statickým obsahem. Při pravidelných obnovách by měli uživatelé neustále k dispozici nejnovější možné informace, jako jsou nové zvířecí přírůstky a naopak jedinci, kteří se už v ZOO nenachází, změny v lokalitách a rozmístění zvířat, jejich detailní popis a zajímavosti, ceny a možnosti adopcí, nebo nově vznikající události. Stejně tak by se zlepšila a rozšířila kvalita kvízových otázek, jelikož by bylo doplňováno množství dosud chybějících dat. Synchronizace s Open daty by poté probíhala pomocí pravidelných spouštění importního skriptu, který by databázi mohl namísto kompletního přemazání pouze updatovat.

5.2 Lepší synchronizace dat mezi mobilní aplikací a `backend` serverem

Na pravidelné aktualizaci Open dat by byla navázána lepší funkce přenosu těchto informací do mobilní aplikace, kam by se dostávala zpráva o smazání dokumentu. V současné situaci by byla lokální data v aplikaci neaktuální a obsahovala by neexistující události nebo zvířata, která v ZOO již nejsou. Vyjma toho by si záznam v `backend` databázi musel udržovat svůj identifikátor, jelikož se jeho existence v aplikaci kontroluje právě pomocí něj - v případě

jeho změny (při novém importu z Open dat) by se záznam pro aplikaci tvářil jako úplně nový.

5.3 Optimalizace filtrů lexikonu zvířat

Současná podoba filtrů umožňuje vybrat pouze jeden výchozí a ten případně zkombinovat s vyhledávaným řetězcem v názvech zvířat. Optimalizací tohoto procesu by se dosáhlo stavu, kdy má uživatel možnost vybrat filtrů libovolné množství a to si poté na obrazovce se seznamem zvířat dodatečně spravovat. Přidával by například taxonomické třídy nebo naopak ubíral druhy biotopů. Tím by se dosáhlo výrazně specifičtějšího výběru z lexikonu.

Kromě toho by bylo vhodné do menu zanést možnost nechat si zobrazit všechna zvířata, bez jakýchkoliv výchozích filtrů. Seznam by musel implementovat stránkování a pokročilou cache, aby jeho procházení nepůsobilo trhaně a zdlouhavě.

5.4 Zobrazování novinek ze ZOO

ZOO na svém webu nabízí *RSS feed* aktuálních zpráv a novinek, které pravidelně aktualizuje. Tento kanál je postaven na bázi *XML* a je veřejně přístupný, čili by šel snadno integrovat jak do `backend` části tak do mobilní aplikace, kde by měl buď vlastní sekci nebo se zobrazoval na úvodní stránce.

5.5 Průběžné přizpůsobování kvízových otázek

Kvíz by bral v úvahu otázky označené uživateli jako chybné a podle toho by je depublikoval a vynechal tak z dalších dotazů na `API`. Z počátku by kontrola správnosti mohla probíhat manuálně, s rostoucím počtem takových případů by se automatizovala. Depublikované otázky by se postupně mazaly a nahrazovaly novými stejného typu, aby byla zachována diverzita.

Obtížnost se aktuálně silně odvíjí od náhodného systému, jakým se otázky generují. Dochází tak k případům, kdy jsou odpovědi zbytečně dlouhé (obsahují například výčet všech druhů potravy, jimiž se zvíře živí a kterých může být mnoho). Měla by se také brát v úvahu diverzita odpovědí, zakládaná třeba na podobnosti biotopů nebo společná třída taxonomických řádů.

Dalším rozšířením je možnost stanovovat obtížnost otázek na základě poměru dvou hodnot: kolikrát byly vybrány a zobrazeny uživatelům proti počtu správných odpovědí. Kvíz by se tak dal rozdělit do úrovní s postupně rostoucí obtížností, případně by si uživatel sám volil, jak složité otázky mu budou v rámci kvízu nabízeny.

5.6 Jazyková lokalizace

Aktuálně je celé uživatelské rozhraní mobilní aplikace v českém jazyce a je tak určena pouze pro česky mluvící publikum. Lokalizace tohoto rozhraní je v Android aplikaci snadným úkolem, který vyžaduje pouze překlad všech řetězců do cílového jazyka a udržování nově vzniklých souborů v jiném místě aplikační struktury. Větší výzvou je překlad všech informací v Open datech, bez čehož by předchozí rozšíření nemělo význam.

5.7 Registrace uživatelů

Registrace v mobilní aplikaci by umožnila přesnou identifikaci uživatele, který by se mohl prezentovat vlastním uživatelským jménem a avatarem (personalizovaným obrázkem), které by byly následně využívány ke globálním statistikám kvízu, které by se udržovaly a počítaly na `backend` serveru. Díky unikátnosti jména by se zavedl i žebříček nejlepších skóre, který by nemusel být lokální pouze pro jedno koncové zařízení. Přihlašování by se řešilo prostřednictvím *JWT*[51], pro které je jak v Android systému[23] tak v `Node.js`[24] podpora. Registrace by byla nepovinná a poskytovala by uživateli další výhody v podobě interaktivní mapy a kvízových medailí.

5.8 Interaktivní mapa

S touto možností se před návrhem mobilní aplikace pracovalo, ale byla již v rané fázi kvůli složitosti implementace vypuštěna. Jednalo by se o mapu areálu ZOO Praha, ve kterém by byly vyznačeny jednotlivé lokality (pavilony, terária, kiosky, výběhy apod.) a sloužily tak uživatelům ke snadnější orientaci. Interaktivita by spočívala v možnosti zadávat do mapy vlastní styčné body včetně jejich popisu, které by definovaly zajímavá místa v rámci areálu, kam se uživatel plánuje dostat nebo která již navštívil. Místa by měla různou roli (např. právě příznak o tom, zda mají být součástí plánované trasy), pomocí které by se skrz ně vytvořil okruh, buď definovaný samotným uživatelem nebo generovaný automaticky. Kvůli uchovávání těchto informací na `backend` serveru by byla vyžadována registrace.

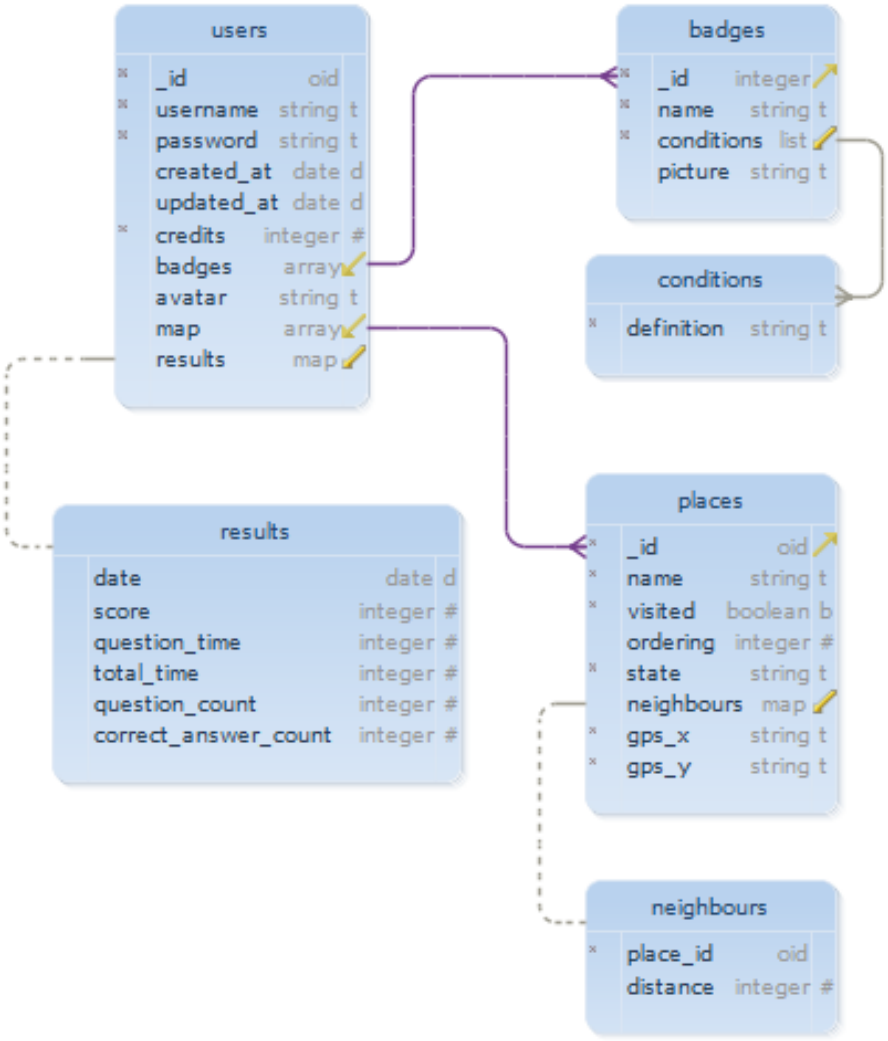
5.9 Kvízové medaile

Další funkce, jež byla z výsledné aplikace vyřazena. Jednalo by se o částečnou gamifikaci kvízu, tedy jeho rozšíření o systém odměn. Každá medaile by byla podmíněna množinou cílů, kterých by bylo třeba dosáhnout pro její získání (např. počet odehraných kvízů, počet správně zodpovězených otázek v rámci

[51] `https://jwt.io/`

jedné hry, minimální výsledné skóre apod.). Spravovány by byly v databázi na serveru a udělovaly by se na základě příchozích požadavků z mobilní aplikace. Přístupné by v ní byly skrze kvízové menu. Zvýšil by se tak zájem o hru a jejím prostřednictvím i znalosti o zvířatech.

Databázové kolekce pro uživatele, mapy i medaile spolu s jejich vzájemným propojením byly již navrženy a měly by mít podobu znázorněnou na obrázku 5.1.



Obrázek 5.1: Schéma plánovaných databázových kolekcí pro uživatele, mapy a medaile

Závěr

Vzniklá mobilní Android aplikace obsahuje rozsáhlý lexikon zvířat, který poskytuje množství zajímavých informací, zprostředkovaných ZOO Praha a lze ho prohlížet i filtrovat. Uživatelé se dále mohou dobře orientovat v nastávajících událostech nebo možnostech adopce některých zvířat. Vedle toho se zde nachází možnost si zahrát vědomostní kvíz, který čerpá z rozsáhlé databáze náhodně generovaných otázek. To vše je podporováno neustále běžícím `backend` serverem, který importuje informace z Open dat a ty následně transformuje, navzájem propojuje a nabízí k odebírání prostřednictvím `API`. Aplikace splňuje požadavky na moderní a přehledný vizuální návrh, přičemž podporuje co možná nejširší spektrum zařízení, které ji mohou instalovat a provozovat. Její funkcionalita počítá i s absencí síťového připojení a je tak schopna po úvodním získání dat pracovat i v offline režimu. Obě části, jak serverová tak mobilní, byly řádně testovány i laděny a fungují v produkčním prostředí, kde jsou uživatelům volně k dispozici.

Jak bylo popsáno v kapitole 5, možností rozšíření stávající funkcionality je skutečně mnoho, čehož si je autor vědom a plánuje jednotlivé prvky postupně implementovat a poskytovat tak čím dál rozsáhlejší množinu funkcí i větší pohodlí při používání aplikace. S tím je ovšem úzce spjata konzistence, integrita a pravidelná aktualizace zdrojových dat, což je úkol také pro ZOO Praha, se kterou je třeba budoucí vývoj konzultovat a synchronizovat. Nyní stále množství informací o zvířatech chybí nebo je neaktuální, což vede k častým výjimkám v kódu a prázdným místům v aplikaci. Kvůli tomu je aplikace zatím dostupná ke stažení pouze jako vývojová verze.

Literatura

[1] Kitchin, R.: *The data revolution: Big data, open data, data infrastructures and their consequences*. Sage, 2014.

[2] Buckler, C.: SitePoint Smackdown: PHP vs Node.js. [online], 2015. Dostupné z: https://www.sitepoint.com/sitepoint-smackdown-php-vs-node-js/

[3] Florisson, M.: How to solve dependency hell. [online], 2015. Dostupné z: https://www.cl.cam.ac.uk/~mbf24/2015/06/22/how-to-solve-dependency-hell/index.html

[4] Buna, S.: Understanding Node.js Event-Driven Architecture. [online], 2017. Dostupné z: https://medium.freecodecamp.org/understanding-node-js-event-driven-architecture-223292fcbc2d

[5] Masse, M.: *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.", 2011.

[6] Klabnik, S.: Latest Specification (v1.0). [online]. Dostupné z: http://jsonapi.org/format/

[7] Bowell, R.: Understand JavaScript Callback Functions and Use Them. [online], 2016. Dostupné z: http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/

[8] Hodi, T.: Node.js Async Best Practices and Avoiding the Callback Hell. [online], 2017. Dostupné z: https://blog.risingstack.com/node-js-async-best-practices-avoiding-callback-hell-node-js-at-scale/

[9] Archibald, J.: JavaScript Promises: an Introduction. [online], 2017. Dostupné z: https://developers.google.com/web/fundamentals/getting-started/primers/promises

[10] Behrens, M.: JavaScript Promises – How They'll Work Someday. [online], 2016. Dostupné z: https://spin.atomicobject.com/2016/02/18/future-javascript-promises/

[11] Gaafar, M.: 6 Reasons Why JavaScript's Async/Await Blows Promises Away. [online], 2017. Dostupné z: https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9

[12] Fowler, M.: Inversion of control containers and the dependency injection pattern. 2004. Dostupné z: https://martinfowler.com/articles/injection.html

[13] Vladimir: MongoDB will not prevent NoSQL injections in your Node.js app. [online], 2016. Dostupné z: https://blog.sqreen.io/mongodb-will-not-prevent-nosql-injections-in-your-node-js-app/

[14] Leslie, A.: Where to Find Free Node.js Hosting — Top 8 Hosts. [online], 2017. Dostupné z: http://www.hostingadvice.com/blog/where-to-find-free-node-js-hosting/

[15] Heroku: Dynos and the Dyno Manager. [online], 2017. Dostupné z: https://devcenter.heroku.com/articles/dynos

[16] Leslie, A.: Most Widely Deployed and Used Database Engine. [online]. Dostupné z: https://www.sqlite.org/mostdeployed.html

[17] Fielding, R.; Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. 2014.

[18] Google: Layouts. [online]. Dostupné z: https://developer.android.com/guide/topics/ui/declaring-layout.html

[19] Google: Supporting Multiple Screens. [online]. Dostupné z: https://developer.android.com/guide/practices/screens_support.html

[20] Google: Write and View Logs with Logcat. [online]. Dostupné z: https://developer.android.com/studio/debug/am-logcat.html

[21] Google: Testing UI for a Single App. [online]. Dostupné z: https://developer.android.com/training/testing/ui-testing/espresso-testing.html

[22] Fabric: The most powerful, yet lightest weight crash reporting solution. [online]. Dostupné z: https://docs.fabric.io/android/crashlytics/overview.html

[23] Muzaffar, A.: Use Android Studio to stub web API; simple JWT login as an example. [online], 2016. Dostupné z: https://medium.com/@ali.muzaffar/use-android-studio-to-stub-web-api-simple-jwt-login-as-an-example-615ee8efe1b0

[24] Sevilleja, C.: Authenticate a Node.js API with JSON Web Tokens. [online], 2015. Dostupné z: https://scotch.io/tutorials/authenticate-a-node-js-api-with-json-web-tokens

Příloha **A**

Seznam použitých zkratek

ACID Atomicity, consistency, isolation, durability

API Application interface

CC0 Creative commons zero

CSV Comma-separated values

DP Density independent pixel

DPI Dots per inch

DOM Document object model

GPS Global positioning system

GSON Google JSON

HATEOAS Hypermedia as the engine of application state

HTML Hypertext markup language

HTTP Hypertext transfer protocol

HTTPS Hypertext transfer protocol secure

IP Internet protocol

IDE Integrated development environment

ISO International organization for standardization

JIT Just in time

JSON JavaScript object notation

JWT JSON web token

MIME Multipurpose Internet mail extensions

MSON Markdown syntax for object notation

NoSQL Not only SQL

OOP Object oriented programming

ORM Object relational mapping

OS Operating system

PHP PHP: hypertext preprocessor

RAM Random-access memory

REST Representational state transfer

RSS Rich site summary

SQL Structured query language

SP Scale independent pixel

UI User interface

URL Uniform resource locator

XLSX Excel Microsoft Office open XML format spreadsheet file

XML Extensible markup language

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace mobilní aplikace
- `design` ..... adresář s mockup návrhem mobilní aplikace
- `test` ..... adresář s testy backend API
- `src`
  - `backend` ..... zdrojové kódy implementace backend serveru
  - `app` ..... zdrojové kódy implementace mobilní aplikace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF