VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SERVEROVÁ APLIKACE A MOBILNÍ APLIKACE PRO ZAŘÍZENÍ IOS KE SNÍMANÍ A HODNOCENÍ KVALITY SNÍMKU ZA ÚČELEM IDENTIFIKACI ZVĚŘE

SERVER APPLICATION AND MOBILE APPLICATION FOR IOS DEVICES TO CAPTURE AND EVALUATE IMAGE QUALITY IN ORDER TO IDENTIFY ANIMALS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

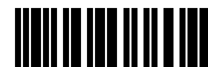
AUTOR PRÁCE — **Bc. ŠIMON ŠESTÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ŠTĚPÁN RYDLO**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce



146446

Ústav: Ústav inteligentních systémů (UITS)
Student: **Šesták Šimon, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Vývoj aplikací
Název: **Serverová aplikace a mobilní aplikace pro zařízení iOS ke snímaní a hodnocení kvality snímku za účelem identifikaci zvěře**
Kategorie: Mobilní aplikace
Akademický rok: 2022/23

Zadání:

1. Seznamte se možnostmi vývoje aplikací pro zařízení s operačním systémem iOS s následným zpracováním a vyhodnocením kvality obrazu. Dále se seznamte s tvorbou serverových aplikací ke sběru dat a informací z aplikací na mobilních zařízení.
2. Prostudujte literaturu ohledně způsobů identifikace osob pro určení možností použití k identifikaci zvěře na základě snímku nosu zvěře. Zaměřte se na algoritmy pro kontrolu kvality snímků ke stanovení možností následného získání identifikačních markantů.
3. Navrhněte serverovou aplikaci s možností identifikace uživatele pro vytvoření databáze ke sběru a ukládání záznamů vytvořených pomoci mobilní aplikace.
4. Navrhněte mobilní aplikaci pro snímání nosu zvířat s kontrolou kvality snímku a následným nahráváním dat na vzdálený server. Detekujte pomoci zpracování obrazu nebo neuronových sítí snímanou oblast nosu zvířat v reálném čase za účelem zaostření obrazu.
5. Implementujte navržené řešení serverové aplikace a klientské mobilní aplikace z předchozích bodů zadání.
6. Otestujte serverovou aplikaci a zhodnoťte dosažené výsledky mobilní aplikace ohledně detekce kvality snímku. Navrhněte vylepšení aplikace pro případné zvýšení kvality snímků.

Literatura:

- J. R. PARKER. *Algorithms for Image Processing and Computer Vision*. John Wiley, 2010. ISBN 0470643854.
- NEUBURG, Matt. *IOS 15 programming fundamentals with Swift: Swift, Xcode, and Cocoa basics*. Eighth edition. Sebastopol, CA: O'Reilly, 2021. ISBN 978-109-8118-501.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rydlo Štěpán, Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 3.11.2022

Abstrakt

Diplomová práca sa zaoberá návrhom aplikácie a REST API servera a ich implementáciami. Aplikácia je vyvíjaná pre operačný systém iOS, jej funkcionalita by mala zabezpečiť jednoduchú komunikáciu so serverom, ktorý bude ukladať a spravovať dáta. Nakoľko server ukladá aj fotografie zvierat určené pre identifikáciu, aplikácia pred nahratím obrázka na server obrázok analyzuje a odošle ho iba v prípade, že obsahuje potrebné informácie. Na analýzu obrázka sa používa natrénovaný model neurónovej siete.

Abstract

The diploma thesis deals with the design of the application and REST API server and their implementations. The application is developed for the iOS operating system. Functionality of application should ensure simple communication with the server that will store and manage data. Since the server also stores photos of animals intended for identification, the application analyzes the image before uploading it to the server and sends it only if it contains the necessary information. A trained neural network model is used to analyze the image.

Kľúčové slová

REST API, ukladanie na vzdialený server, mobilná aplikácia, Swift, SwiftUI, architektúra MVVM-C, iOS, rozpoznanie vzoru v obraze.

Keywords

REST API, saving to remote server, mobile application, Swift, SwiftUI, MVVM-C architecture, iOS, pattern recognition in an image.

Citácia

ŠESTÁK, Šimon. *Serverová aplikace a mobilní aplikace pro zařízení iOS ke snímaní a hodnocení kvality snímku za účelem identifikaci zvěře.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Štěpán Rydlo

Serverová aplikace a mobilní aplikace pro zařízení iOS ke snímaní a hodnocení kvality snímku za účelem identifikaci zvěře

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne a pod vedením Ing. Štěpána Rydla. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

....
Šimon Šesták
14. mája 2023

Poďakovanie

Chcel by som poďakovať vedúcemu práce pánovi Ing. Štěpánovi Rydlovi, ktorý mi umožnil realizovať prácu pod jeho vedením. Ďakujem za poskytnuté rady, udeleniu prístupu k serveru na účel trénovania modelov neurónovej siete a za konštruktívnu kritiku počas návrhu a pomoc pri úprave odborného textu.

Obsah

1 Úvod **3**

2 Zmysel vývoja **4**
- 2.1 Problém identifikácie zabitej zveri . . . 4
- 2.2 Riešenie problému . . . 4
- 2.3 Poskytnuté dáta . . . 5

3 Vývoj aplikácií pre operačný systém iOS **7**
- 3.1 XCode . . . 7
- 3.2 Swift . . . 7
- 3.3 Perzistentné úložiská . . . 9
- 3.4 Kódovanie a dekódovanie . . . 9
- 3.5 Architektonické vzory . . . 10
  - 3.5.1 Model-View-Controller . . . 10
  - 3.5.2 Model-View-ViewModel . . . 11
  - 3.5.3 Model-View-ViewModel-Coordinator . . . 12
- 3.6 Swift package manager . . . 13
- 3.7 Spracovanie obrazu . . . 13

4 Vývoj servera **15**
- 4.1 Bezstavový protokol . . . 15
- 4.2 Metódy HTTP . . . 15
- 4.3 Webové rozhranie - API . . . 16
- 4.4 REST . . . 16
- 4.5 Flask-RESTX . . . 16
- 4.6 JSON Web Token . . . 17

5 Neurónové siete **18**
- 5.1 Detekcia objektov v obraze . . . 18
  - 5.1.1 Histogram orientovaných gradientov . . . 20
- 5.2 Keras . . . 21
- 5.3 Core ML . . . 22
- 5.4 CreateML . . . 22

6 Návrh **23**
- 6.1 Server . . . 23
  - 6.1.1 Špecifikácia požiadavkou . . . 23
  - 6.1.2 Databáza . . . 24

6.1.3 Koncové body . . . 26
6.2 Aplikácia . . . 27
6.2.1 Špecifikácia požiadavkou . . . 27
6.2.2 Prípady použitia . . . 28
6.2.3 Užívateľské rozhranie . . . 28
6.2.4 Databáza . . . 28
6.2.5 Vytvorenie fotografie . . . 31
6.3 Iterácie vývoja . . . 33

7 Implementácia **35**
7.1 Server . . . 35
7.1.1 Architektúra . . . 35
7.1.2 Autentifikácia . . . 36
7.2 Aplikácia . . . 36
7.2.1 Užívateľské rozhranie . . . 37
7.2.2 Ukladanie lokálnych dát . . . 38
7.2.3 Práca so vzdialeným serverom . . . 39
7.2.4 Vytváranie fotografie . . . 41
7.3 Model neurónovej siete . . . 42
7.3.1 Trénovanie modelov . . . 43

8 Testovanie **46**
8.1 Server . . . 46
8.2 Aplikácia . . . 46
8.2.1 Prepojenie aplikácie a servera . . . 47
8.3 Modely neurónovej siete . . . 47

9 Záver **49**

Literatúra **51**

A Diagramy návrhov iterácií **54**

B Snímky obrazovky **56**

Kapitola 1

Úvod

Táto diplomová práca sa zameriava na vývoj aplikácie pre poľovníkov, ktorá dokáže zabezpečiť evidovanie a identifikáciu zabitej zveri. Aplikácia kladie dôraz na intuitívnosť a pomoc poľovníkom pri postupe evidovania zabitej zveri a na kvalitu vyhotovených fotografií, ktoré sú zaslané na server na vyhodnotenie identifikácie zabitého zvieraťa. Súčasťou práce je návrh servera, ktorý musí archivovať poskytnuté fotografie od jednotlivých poľovníkov a zároveň musí zabezpečiť autentifikáciu poľovníka.

V tejto diplomovej práci sa nachádza rozpracovanie návrhu servera, kde je možné sa dočítať ako sa postupovalo a čo boli hlavné požiadavky, ktoré formovali výsledný návrh. Okrem návrhu servera práca obsahuje aj návrh aplikácie, kde počas návrhu bol zvolený iteratívny prístup. V práci sú rozpísané jednotlivé iterácie a zároveň aj zamýšľaný finálny stav aplikácie, do ktorého vývoj smeruje. Súčasťou návrhu servera aj aplikácie sú databázové schémy, ktoré by mali čo najpresnejšie odrážať schémy databáz použité vo finálnych produktoch. Zároveň sa v návrhu aplikácie nachádzajú aj obrázky z prototypu užívateľského rozhrania.

Okrem návrhu spomínaných častí sa v práci pojednáva o možnostiach zautomatizovania vyhodnotenia kvality fotografie, ktorá zachytáva oblasť slúžiacu na identifikovanie divokej zveri. Touto oblasťou je hlava zvieraťa, špecifickejšie okolie nosa. Na automatizáciu tohto vyhodnotenia sa trénovali modely neurónových sietí, z ktorých bol následne vybraný ten s najlepšími výsledkami a bol integrovaný do aplikácie. Aplikácia tento model využíva pri vytváraní fotografie, tak že každý snímok zachytený počas využívania fotoaparátu je zaslaný do modelu a ten daný snímok vyhodnocuje. Jeho výsledky aplikácia propaguje užívateľovi v réalnom čase na displej zariadenia.

Kapitola 2

Zmysel vývoja

Mobilné zariadenie je bežnou súčasťou skoro každého človeka. Väčšina z vlastníkov mobilných zariadení má k dispozícii inteligentné mobilné zariadenie, ktoré dokáže využívať rôzne aplikácie na rôzne účely. Vychádzajúc z tohto faktu, aplikáciu na identifikáciu zabitej zveri môže využívať majorita ľudí, ktorí musia problematiku identifikácie zabitej zveri riešiť. Dôvody potreby riešenia tejto problematiky sú vysvetlené v tejto kapitole spolu s možným riešením tohto problému.

2.1 Problém identifikácie zabitej zveri

Povinnosťou každého poľovníka, je zaevidovať každé ulovené zviera. Evidencia zvieraťa doposiaľ prebiehala označením nohy štítkom a následným vyhotovením fotografie, tak aby bol štítok viditeľný a možná jasná identifikácia. Tento spôsob evidencie môže byť zneužitý a to tak, že poľovníci po vyhotovení fotografie štítok strhli a nahradili ho novým. Takže už zaevidované zviera po určitom čase znova označili novým štítkom a vyfotografovali, následne ho zaevidovali ako nové zabité zviera. V prípade, že sa jednalo o zver, za ktorú bola vyplácaná odmena, mohli poľovníci byť vyplatení niekoľkokrát za to isté zviera.

V praxi nastávajú situácie kedy je potrebné preriediť určité druhy zvierat, nakoľko môžu byť premnožené, prípadne sa vyskytla medzi nimi choroba. No takéto falšovanie zabitých kusov zvierat by malo za následok, že by sa v štatistikách javilo, že zvieratá začínajú dosahovať hranicu kedy ich počet dosahuje požadovaných hodnôt, ale skutočnosť bola iná a tak mohli tieto zvieratá napríklad roznášať choroby aj na domestikované zvieratá.

2.2 Riešenie problému

Tak ako ľudia majú jedinečný vzor otlačku prstov, bolo vedecky dokázané, že otlačok nosa zvieraťa je jedinečný vzor, ktorý môže slúžiť k jeho identifikácii. Na základe toho by malo byť možné vytvoriť aplikáciu, ktorá bude dostupná na mobilných zariadeniach a po zabití zvieraťa bude možné vyfotografovať jeho časť určenú k identifikácii. Týmto sa zamedzí falšovaniu zabitej zvere a to môže mať pozitívny vplyv na ekonomiku poľovníckej komory.

Vedecké výskumy

Na dokázanie teórie o jedinečnosti vzoru, nachádzajúcom sa na nose zvierat, boli vypracované viaceré štúdie zaoberajúce sa zväčša konkrétnym druhom zvierat.

Pravdepodobne prvou takouto štúdiou sa zaoberal *W.E. Petersen* už v roku *1921*, kedy na vzorke 350 kusov dobytka sa snažil dokázať, že ani jeden otlačok dvoch rôznych kusov dobytka nie je rovnaký. Na vytvorenie otlačku nosa bolo potrebné najskôr nos zvieraťa očistiť a vysušiť. Následne bol na nos nanesený atrament a potom priložený papier tak, že spodná časť papiera sa dotýkala hornej pery. Na papier sa rolovaním postupne vytvoril otlačok celého nosa. Počas štúdie boli otlačky rozdelené do šiestich typov nosov, na základe toho pod akým uhlom vychádzajú línie zo stredu otlačku. A tak prvým krokom overenia dvoch otlačkov nosa bolo zistiť, či sú otlačky rovnakého typu. Následne sa postupovalo ako pri identifikácii otlačkov prstov. Na jednom otlačku boli nájdené určité detaily takzvané markanty, ktoré sa následne hľadali a porovnávali aj na druhom otlačku [36].

Ďalšou takouto štúdiou sa zaoberali *Takahiro Murakami*, *Kohji Uraguchi* a *Go Abe* ktorí dokázali, že je možné z fotografií otlačku nosa zvierat druhu mezokarnivorov[1] jedinečne identifikovať konkrétneho jedinca [33].

Za posledný zverejnený výskum sa môže považovať výskum viacerých autorov, ktorí dokázali jedinečnosť vzoru na nose psov. V tomto výskume bolo analýze podrobených 70 psov rôznych plemien a veku. Okrem vizuálneho dokazovania jedinečnosti bola použitá aj algoritmizácia za pomoci Gáborovovej transformácie a následného porovnávania pomocou Hammingovej vzdialenosti. Postupovali tak, že najskôr vytvorili pre každého psa z jednej fotografie šablónu. Na jej vytvorenie orezali pôvodnú fotku tak, aby výrez reprezentoval iba časť nosa, na ktorom je najlepšie vidieť jedinečný vzor identifikujúci konkrétneho psa. Následne na takto upravenú fotografiu použili Gáborovú transformáciu, ktorá im vyprodukovala pole jedničiek a núl, reprezentujúce šablónu daného psa. Takto vytvorenú šablónu uložili do databázy a tým umožnili nasledujúce porovnávania. Porovnávanie pozostávalo z úpravy vstupnej fotografie na vytvorenie nového poľa reprezentované jedničkami a nulami. Toto pole porovnali so šablónami uloženými v databáze a to pomocou Hammingovej vzdialenosti. Na základe šablóny z databázy, ktorá dávala hodnotu Hammingovej vzdialenosti najmenšiu, mohli určiť o ktorého jedinca sa na vstupnej fotografii jednalo[19].

2.3 Poskytnuté dáta

Na účel tejto diplomovej práce, boli poskytnuté fotografie zabitej zveri od poľovníkov. Tieto fotografie bolo potrebné pretriediť, nakoľko sa v nich nachádzali aj fotky, ktoré neobsahovali žiadne potrebné informácie na riešenie tohto problému. Aplikácia by nemala zbytočne prenášať veľké počty dát, ktoré nebudú použiteľné k identifikácii zvieraťa na serveri. Preto je potrebné, aby pred odoslaním na server fotografia prešla základnou analýzou. O túto analýzu sa bude starať natrénovaný model neurónovej siete, ktorý overí kvalitu fotografie, a to tak, že zistí či fotografia obsahuje nos zvieraťa, a či daná oblasť na ktorej sa nos nachádza je v dostatočnej kvalite.

Fotografie boli rozdelené do troch kategórií, kde každá kategória predstavuje ostrosť a viditeľnosť nosa zvieraťa na fotografii. Tieto kategórie môžme nazvať *ostré*, *viditeľné* a *rozmazané*. Príklady fotografií z jednotlivých kategórií sa nachádzajú na obrázkoch 2.1, 2.2 a 2.3.

[1] Do čeľade tohto druhu patria: líška, kojot, medvedík čistotný, ...

Obr. 2.1: Ostré

Obr. 2.2: Viditeľné

Obr. 2.3: Rozmazané

Kapitola 3

Vývoj aplikácií pre operačný systém iOS

Vývoj mobilných aplikácií stále napreduje a vznikajú rôzne prístupy a názory k tomu ako správne navrhnúť a následne vytvoriť mobilnú aplikáciu.

Mobilné aplikácie bežiace na zariadeniach s platformou *iOS* od spoločnosti *Apple*, musia spĺňať určité pravidlá tejto spoločnosti a zároveň ich tvorba musí prebiehať na zariadeniach od tejto spoločnosti.

Apple ponúka veľkú škálu rôznych rámcov a knižníc, ktoré zapúzdrujú často používané funkcie mobilných aplikácií.

V tejto kapitole sú opísané knižnice, rámce a vývojové prostredie zapúzdrujúce podporu funkcií, ktoré bude obsahovať výsledná mobilná aplikácia na podporu identifikácie zabitej zveri.

3.1 XCode

Pre vyvíjanie aplikácií bežiacich na platforme iOS sa používa integrované vývojové prostredie *Xcode*. Xcode taktiež podporuje vyvíjanie aplikácií bežiacich na macOS, watchOS, tvOS, iPadOS. Bol vyvinutý firmou *Apple* v roku 2003. Jeho posledná verzia *14.0.1* vyšla 26.9.2022. A beta verzia *14.1* je dostupná od 18.10.2022. Xcode je možné využívať len na počítačoch s operačným systémom macOS.

Súčasťou *Xcode* je textový editor, debugger program pre ladenie vyvíjanej aplikácie, interface builder program pre vytvorenie grafického užívateľského rozhrania aplikácie, podpora správy zdrojového kódu pomocou riadenia verzií GIT, ktorý užívateľovi umožnuje verzovanie projektu, pomocou repozitárovej štruktúry. V neposlednom rade *Xcode* obsahuje väčšinu vývojovej dokumentácie od *Applu* a simulátor zariadení od spoločnosti *Apple* pre testovanie vytváraných aplikácií. *Xcode* implicitne podporuje tieto programovacie jazyky: C, C++, Objective-C, Java, AppleScript, Swift, SwiftUI. [16]

3.2 Swift

Swift je multi-paradigmatický, kompilovaný programovací jazyk vytvorený spoločnosťou Apple a je určený na vytváranie aplikácií v systémoch Apple, Mac a Linux. Hoci je založený na Objective-C a iných jazykoch, bol navrhnutý tak, aby bol ľahší, intuitívnejší, bezpečnejší a ľahšie použiteľný v porovnaní s mnohými alternatívami.

Chris Lattner, bývalý hlavný riaditeľ vývojárskych nástrojov spoločnosti Apple, začal vyvíjať Swift v júli 2010. Lattner a tím vývojárov ho založili po vzoroch jazykov Objective-C, Ruby, Pythone a mnohých ďalších.

V júni 2014 spoločnosť Apple vydala beta verziu pre registrovaných vývojárov spoločnosti Apple a v októbri 2014 bola vydaná verzia Swift 1.1 spolu so spustením vývojového prostredia Xcode 6.1.

Najpozoruhodnejšou vlastnosťou Swift je jeho podobnosť s Objective-C. Používa runtime knižnicu Objective-C, ktorá umožní kódu C, Objective-C, C++ a Swift bežať v rámci jedného programu. Podporuje koncept rozširovania tried, štruktúr a typov pomocou protokolov. Tento koncept prezentuje spoločnosť *Apple* ako naozajstnú zmenu a takýto jazyk nazývajú „protokolovo orientované programovanie". Jazyk je schopný písať programy pre macOS, iOS, iPadOS, watchOS, tvOS, ako aj Linux. Keďže Swift je open-source, je pravdepodobné, že by sa dal začleniť aj do iných platforiem, vrátane webového dizajnu. [18]

UIKit

UIKit je rámec, vyvíjaný spoločnosťou *Apple*, ktorý poskytuje množstvo funkcií na vytváranie aplikácií vrátane komponentov, ktoré sa môžu použiť na zostavenie základnej infraštruktúry aplikácií pre iOS, iPadOS alebo tvOS. Poskytuje viaceré architektúry na implementáciu používateľského rozhrania, infraštruktúru spracovania udalostí na poskytovanie multidotykov a iných typov vstupu do aplikácie a hlavnú slučku spustenia na správu interakcií medzi používateľom, systémom a aplikáciou.

Taktiež zahŕňa podporu pre animácie, dokumenty, kreslenie a tlač, správu a zobrazenie textu, vyhľadávanie, správu zdrojov a získavanie informácií o aktuálnom zariadení. Zabezpečuje možnosť prispôsobiť podporu dostupnosti a lokalizovať rozhranie aplikácie pre rôzne jazyky, krajiny alebo kultúrne oblasti.[29]

SwitUI

SwiftUI podobne ako *UIKit* 3.2 je vyvíjaný spoločnosťou *Apple* a poskytuje zobrazenia, ovládacie prvky a štruktúry rozloženia na deklarovanie používateľského rozhrania aplikácie. Ďalej poskytuje obslužné nástroje udalostí pre klepnutia, gestá a iné typy vstupov aplikácie. Obsahuje nástroje na riadenie toku údajov z modelov aplikácie až po zobrazenia a ovládacie prvky, ktoré používatelia vidia a s ktorými interagujú.[12][13]

- Hlavnými nevýhodami využívania tohto rámca oproti *UIKit*-u:
  - Užívatelia so starším operačným systémom nebudú schopný aplikáciu využívať.
  - Nakoľko je stále tento rámec celkom nový, môžu sa v ňom vyskytovať chyby s ktorými sa ešte nikto nestretol.
  - Ešte nepodporuje úplne všetky vlastnosti čo *UIKit*.
- Jeho hlavnými výhodami oproti *UIKit*-u:
  - Podporuje využívanie živých náhľadov užívateľského rozhrania a tak uľahčuje a urýchluje jeho vývoj.
  - Umožňuje vytvárať užívateľské rozhranie, použitím zdrojových kódov vyššej abstrakcie.
  - Má potenciál byť v budúcnosti viac využívaný.

3.3 Perzistentné úložiská

Väčšina aplikácií potrebuje svoje dáta ukladať lokálne a nestratiť ich pri ukončení behu aplikácie. Pre takéto ukladanie dát sa využívajú perzistenté úložiská, do ktorých sa dáta zapíšu a sú v nich uložené, pokiaľ ich odtiaľ užívateľ nevymaže. V rámci programovania pre operačné systémy iOS sa k tomuto účelu môže využiť trieda *UserDefaults* 3.3 pre uloženie malých objemov dát alebo využiť rámec *Core Data* 3.3, ktorý spravuje objekty modelovej vrstvy v aplikácii.

UserDefaults

Je to interface umožňujúci prístup k užívateľovej štandartnej databáze. Funguje ako slovník. To znamená, že sa dáta ukladajú pre určitý kľúč, vďaka ktorému môžeme potom k nim pristupovať. Využíva sa primárne pre ukladanie preferencií užívateľa, pre prispôsobenie behu aplikácie, ako je napríklad určiť rýchlosť prehrávania médií, prípadne aké merné jednotky chce užívateľ používať.

UserDefaults si svoje dáta ukladá do cache pamäte, aby sa pri každom vyžiadaní uloženej hodnoty nemusela otvárať užívateľova štandartná databáza. Pri uložení novej hodnoty sa daná hodnota zapíše synchrónne s procesom, ktorý ju zmenil ale zápis do perzistentného úložiska prebieha asynchrónne.

Takto uložené dáta sa vyskytujú lokálne len na konkrétnom zariadení. Pre možnosť zdieľať tieto dáta so všetkými užívateľovými zariadeniami je potrebné využiť triedu *NSUbiquitousKeyValueStore* [30].

Core Data

Poskytuje riešenia na bežné úlohy vykonávané nad životným cyklom objektov a ich perzistentným ukladaním. Za pomoci tohto rámca sa kód zväčša zmenšuje o 50% - 70% vďaka už naprogramovaným vlastnostiam [31].

Core Data vytvorí súbor s príponou .xcdatamodel, v ktorom je možné vytvoriť databázový model, s ktorým bude aplikácia pracovať. Takýto model obsahuje entity a každá jeho entita obsahuje určité atribúty. Entita musí odpovedať určitej triede a tá musí dediť vlastnosti z triedy *NSManagedObject*, vďaka tejto triede nadobúda vlastnosti potrebné pre perzistentné ukladanie dát [31]. Inštanciu takto vytvorenej triedy je možné si predstaviť ako riadok tabuľky, zatiaľ čo atribúty triedy ako jej stĺpce. Ďalej je možné modelovať väzby v rámci modelu medzi jednotlivými entitami. Medzi dôležité vlastnosti vzťahov patrí ich názov, kardinalita a cieľová entita vzťahu.

Kľúčenka

Kľúčenka je najlepším miestom na ukladanie malých tajomstiev, ako sú heslá a kryptografické kľúče. Je možné využívať funkcie rozhrania služieb kľúčenky na pridávanie, získavanie, odstraňovanie alebo upravovanie položiek uložených v kľúčenke. [25]

3.4 Kódovanie a dekódovanie

Pre posielanie dát po sieti je nutné dáta zakódovať. Takto zakódované dáta môžu byť odoslané a prijaté. Prijaté dáta je pred ich využitím nutné dekódovať. V rámci jazyka *Swift* podporuje túto činnosť alias **Codable**, ktorý sa skladá z dvoch protokolov: *Encodable* a

Decodable. Ako je možné rozpoznať z názvov deklarujú, že dátový typ, na ktorý sú použité môže byť zakódovateľný a rozkódovateľný.

V prípade, že štruktúra obsahuje prvky, ktoré vyhovujú aliasu *Codable*, je štruktúru možné zakódovať a rozkódovať. V prípade, že štruktúra obsahuje prvky, ktoré nevyhovujú aliasu *Codable*, štruktúru nie je možné zakódovať alebo rozkódovať. V prípade, že potrebujeme, aby táto štruktúra vyhovovala aliasu *Codable* je možné určiť pre nevyhovujúce prvky, akým spôsobom sa majú zakódovať a rozkódovať.

JSON

JSON (JavaScript Object Notation) je jednoduchý formát na výmenu údajov. Pre ľudí je táto forma kódovania ľahko čitateľná a zapisovateľná, zatiaľ čo pre stroje je ľahko generovateľná a analyzovateľná. Je založený na podmnožine štandardu JavaScript Programming Language Standard ECMA-262 3. vydanie – december 1999. JSON je textový formát, ktorý je úplne nezávislý na jazyku. Vďaka týmto vlastnostiam je JSON ideálnym jazykom na výmenu údajov.

JSON je postavený na dvoch štruktúrach:

- Kolekcia párov názov a hodnota. V rôznych jazykoch sa to realizuje ako objekt, záznam, štruktúra, slovník, hašovacia tabuľka alebo asociatívne pole.
- Usporiadaný zoznam hodnôt. Vo väčšine jazykov sa to realizuje ako pole, vektor, zoznam alebo sekvencia.

Ide o univerzálne dátové štruktúry. Prakticky všetky moderné programovacie jazyky ich podporujú v tej či onej forme. Výnimkou nie je ani swift, ktorý pre tieto akcie obsahuje triedy **JSONEncoder** a **JSONDecoder** [7].

3.5 Architektonické vzory

V informatike sa termín softvérová architektúra vzťahuje na návrh systému na vysokej úrovni, metódy, ktorými sa vytvárajú komponenty, a dokumentácia štruktúr systému. Architektúra definuje, ako jej rôzne časti „zapadajú do seba“.[23]

3.5.1 Model-View-Controller

Architektonický vzor *MVC* priraďuje objektom v aplikácii jednu z troch rolí: model, view alebo controller. Vzor definuje nielen úlohy, pre dané objekty v aplikácii, ale definuje aj spôsob, akým objekty medzi sebou komunikujú. Každý z troch typov objektov je oddelený od ostatných abstraktnými hranicami a komunikuje s objektami iných typov cez tieto hranice. Kolekcia objektov rovnakého typu v aplikácii sa niekedy označuje ako vrstva – napríklad vrstva modelu.

Existuje mnoho výhod využívania tohto architektonického vzoru. Veľa objektov v týchto aplikáciách má tendenciu byť znova použitých a teda ich rozhrania majú tendenciu byť lepšie definované. Aplikácie, ktoré využívajú tento architektonický vzor sú tiež ľahšie rozšíriteľné ako iné aplikácie.

Model

Objekty modelu zapuzdrujú údaje špecifické pre aplikáciu a definujú logiku a výpočty, ktoré manipulujú a spracúvajú tieto údaje. Napríklad objekt modelu môže predstavovať

postavu v hre alebo kontakt v adresári. Objekt modelu môže mať vzťahy typu jeden k jednej alebo jeden k mnoho, s inými objektami modelu a tak niekedy modelová vrstva aplikácie efektívne pozostáva z jedného alebo viacerých objektových grafov. Veľká časť údajov, ktoré sú súčasťou trvalého stavu aplikácie (či už je tento trvalý stav uložený v súboroch alebo databázach), by sa po načítaní údajov do aplikácie mala nachádzať v objektoch modelu. Pretože modelové objekty predstavujú znalosti a odborné znalosti súvisiace so špecifickou problémovou doménou, môžu byť opätovne použité v podobných problémových doménach. V ideálnom prípade by objekt modelu nemal mať žiadne explicitné spojenie s objektami typu view, ktoré prezentujú jeho dáta a umožniť tak používateľom tieto údaje upravovať.

Akcie používateľa vo vrstve zobrazenia (vrstva obsahujúca modely typu view), ktoré vytvárajú alebo upravujú údaje, sú prenášané cez objekt typu controller a vedú k vytvoreniu alebo aktualizácii objektu modelu. Keď sa objekt modelu zmení (napríklad sa prijmú nové údaje cez sieťové pripojenie), upozorní na to objekt typu controller, ktorý aktualizuje príslušné objekty typu view.

View

Objekt typu view je objekt v aplikácii, ktorý môžu používatelia vidieť. Tento objekt vie, ako má správne vykresliť jemu poskytnuté dáta a môže reagovať na akcie používateľa. Hlavným účelom objektov typu view je zobraziť údaje z objektov modelu aplikácie a umožniť úpravu týchto údajov. Napriek tomu sú tieto objekty zvyčajne oddelené od objektov modelu v aplikácii, ktorá využíva architektonický vzor MVC.

Pretože sa zvyčajne znova používajú a prekonfigúrovajú, objekty typu view poskytujú konzistenciu medzi aplikáciami. Rámce UIKit aj AppKit poskytujú kolekcie tried zobrazenia a Interface Builder ponúka vo svojej knižnici desiatky objektov zobrazenia.

Objekt typu view sa dozvie o zmenách v dátach modelu prostredníctvom objektov typu controller a komunikuje zmeny iniciované používateľom – napríklad text zadaný do textového poľa – prostredníctvom objektov typu controller s objektmi modelu aplikácie.

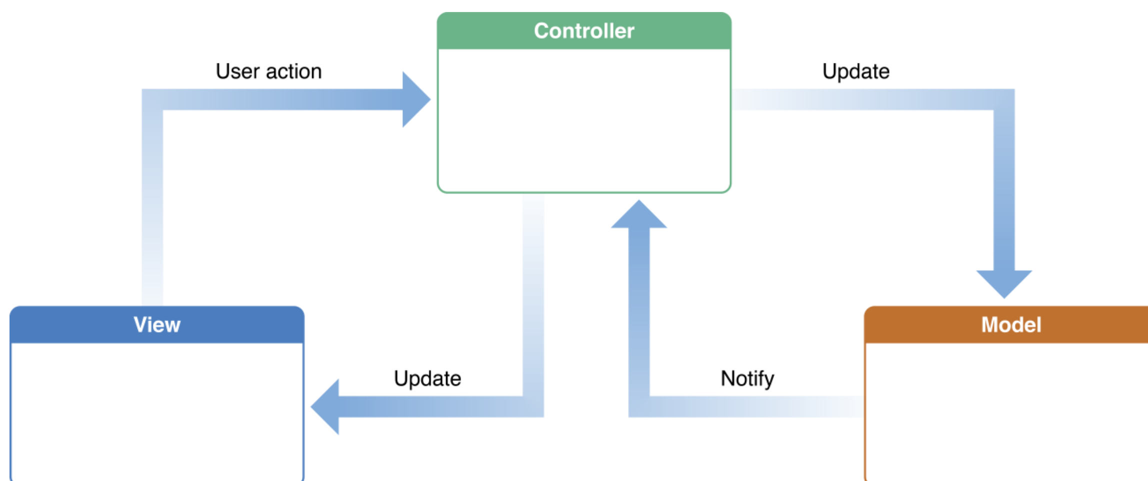
Controller

Objekt typu controller funguje ako sprostredkovateľ medzi jedným alebo viacerými objektami typu view a jedným alebo viacerými objektami modelu. Objekty typu controller sú teda kanálom, cez ktorý sa objekty typu view dozvedia o zmenách v objektoch modelu a naopak. Objekty typu controller môžu tiež vykonávať nastavovacie a koordinačné úlohy pre aplikáciu a spravovať životné cykly iných objektov.

Objekt typu controller interpretuje akcie používateľa vykonané v objektoch typu view a komunikuje nové alebo zmenené údaje do modelovej vrstvy. Keď sa objekty modelu zmenia, objekt typu controller oznámi tieto nové údaje modelu, objektom typu view, aby ich mohli zobraziť. [27]

3.5.2 Model-View-ViewModel

Architektonický vzor *MVVM* uľahčuje oddelenie vývoja grafického používateľského rozhrania (view) – či už prostredníctvom značkovacieho jazyka alebo kódu GUI – od vývoja obchodnej logiky alebo back-endovej logiky (modelu) tak, aby pohľad nebol závislý od žiadnej konkrétnej modelovej platformy. Viewmodel je prevodník hodnôt, čo znamená, že viewmodel je zodpovedný za konvertovanie dátových objektov z modelu takým spôsobom, že objekty sa dajú ľahko spravovať a prezentovať. V tomto ohľade je viewmodel viac modelom



Obr. 3.1: Ukážka komunikácie v rámci MVC [27]

ako view a zvláda väčšinu, ak nie celú logiku zobrazenia. Viewmodel môže implementovať prístup k back-endovej logike okolo skupiny prípadov použitia podporovaných zobrazením.

Tento architektonický vzor je variáciou vzoru prezentačného modelu *Martina Fowlera*. Vymysleli ho architekti Microsoftu *Ken Cooper* a *Ted Peters* špeciálne na zjednodušenie programovania používateľských rozhraní riadeného udalosťami.

Model

Model sa vzťahuje buď na doménový model, ktorý predstavuje obsah v reálnom stave (objektovo orientovaný prístup), alebo na vrstvu prístupu k dátam, ktorá predstavuje obsah (prístup zameraný na dáta).

View

Rovnako ako v architektonickom vzore *MVC* 3.5.1, view predstavuje rozloženie a vzhľad toho, čo používateľ vidí na obrazovke. Zobrazuje reprezentáciu modelu a prijíma interakciu používateľa so zobrazením (kliknutia myšou, vstup z klávesnice, gestá ťuknutia na obrazovku,...) a postupuje ich spracovanie do viewmodel-u prostredníctvom dátovej väzby (premenné, spätné volania udalostí,...), ktorá je definovaná na prepojenie view a viewmodel-u.

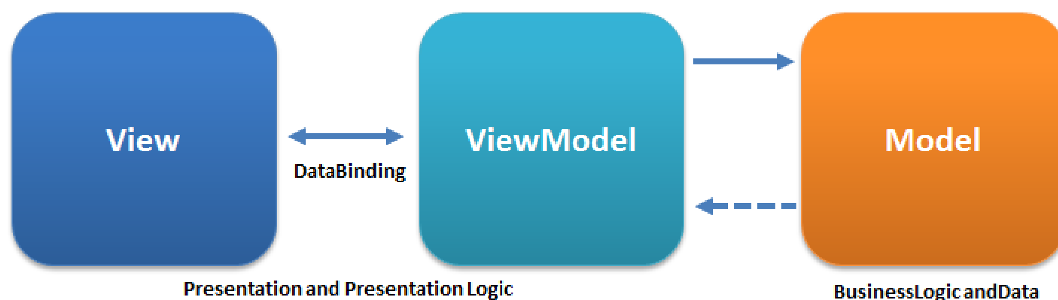
ViewModel

Je abstrakcia view, ktorá odhaľuje verejné premenné a príkazy.Viewmodel bol opísaný ako stav dát v modeli.

Viewmodel nedrží odkaz na view, ktorému poskytuje svoje dáta. Namiesto toho sa view priamo viaže na premenné viewmodel-u a na odosielanie/prijímanie aktualizácií. Aby to fungovalo efektívne, vyžaduje to technológiu viazania. [8]

3.5.3 Model-View-ViewModel-Coordinator

Architektonický vzor *MVVM* 3.5.2 odvádza skvelú prácu pri rozdeľovaní obchodnej a používateľskej logiky. Rieši problém obrovských ovládačov zobrazenia, ktoré sa vyskytujú pri používaní *MVC* 3.5.1.



Obr. 3.2: Ukážka komunikácie v rámci MVVM [8]

Architektonický vzor *MVVM-C* umožňuje ešte viac zmenšiť objekt ovládača zobrazenia, oddelením navigačnej logiky do samostatného objektu nazývaného koordinátor. Všetky ostatné objekty spĺňajú rovnaké úlohy ako v architektonickom vzore *MVVM*. [35]

Coordinator

Koordinátor je objekt, ktorý má, ako už názov napovedá, výhradnú zodpovednosť za koordináciu navigácie aplikácie. V podstate určuje, ktorá obrazovka by sa mala zobraziť, aká obrazovka by sa mala zobraziť ako ďalšia a tak ďalej. [39]

3.6 Swift package manager

Swift Package Manager je nástroj pre jednoduchú distribúciu a znovu použiteľnosť zdrojových kódov. Je integrovaný so systémom vytvárajúcim výstupné súbory, tak aby zautomatizoval proces sťahovania, kompilovania a vytvárania závislostí.

Module, Swift organizuje svoj kód do modulov. Zdrojový kód môže obsahovať všetko v jednom module, alebo obsahovať viacero modulov. Väčšina závislostí vyžaduje stiahnutie a vytvorenie daného kódu na jeho využívanie. Pri využívaní modulov pre určité oblasti problematiky vzniká možnosť takto vytvorený kód znova použiť.

Package, obsahuje súbory so zdrojovým kódom a manifest súbor obsahujúci metadáta pre daný balíček. Manifestový súbor pomenovaný „Package.swift“ definuje meno balíčka a jeho obsah pomocou modulu „PackageDescription“. Balíček obsahuje jeden, alebo viacero cieľov. Každý cieľ špecifikuje produkt, ktorý môže deklarovať jednu, alebo viacero závislostí.

Target, môže ako svoj produkt vytvoriť knižnicu, alebo spustiteľný súbor. Knižnica obsahuje modul, ktorý môže byť importovaný do zdrojového kódu.

Dependecies, závislosti cieľa sú moduly, ktoré vyžadujú kód v balíčku. Závislosť pozostáva z relatívnej alebo absolútnej cesty ku balíčku a požadovanej verzie daného balíčka.

Hlavná úloha *Swift Package Manager*-a pozostáva zo zníženia náročnosti koordinácie procesu sťahovania a vytvárania všetkých závislostí, nakoľko je tento proces rekurzívny (závislosť môže obsahovať vlastné závislosti) vytvára sa graf závislostí [28].

3.7 Spracovanie obrazu

Spracovanie obrazu je súčasťou počítačového videnia, ktoré je v súčasnosti na vzostupe, najmä kvôli možnosti využívať strojové učenie. Priebeh spracovania obrazu sa môže jedno-

ducho vysvetliť tak, že na vstupe je obrázok na ktorý použijeme filter. Filter je program, ktorý algoritmicky skúma pixely a aplikuje na nich určitý efekt, vďaka ktorému je možné vygenerovať nový výstupný obrázok. Skratka obrazového prvku, pixel je hodnota, ktorá predstavuje intenzitu farieb, ktoré tvoria obraz.

Počítačové videnie je oblasť, kde programujeme tak, aby počítač videl prostredie okolo nás, vďaka čomu dokáže rozpoznávať a extrahovať informácie z objektov v reálnom čase. Počítačové videnie je priamo súvisiace so strojovým učením [21].

VisionKit

Je rámec vytvorený spoločnosťou *Apple*. Vykonáva vysoko výkonnú analýzu obrazu pre identifikovanie čiarových kódov, detekovanie textu, tvárí, ale za pomoci integrácie *CoreML* 5.3 môže detekovať aj gestá človeka, mimiku tváre,. . . Analýza môže prebiehať v rámci práve natáčaného videa, uloženého videa, alebo fotografie.

Kapitola 4

Vývoj servera

Server je počítačový program alebo zariadenie, ktoré poskytuje služby alebo zdroje požadované inými programami alebo zariadeniami, známymi ako klienti. Tento vzor pripojenia servera ku klientovi je známy ako model klient-server.

Server môže mať veľa klientov naraz alebo jeden klient môže používať viacero serverov. Servery tiež ponúkajú množstvo funkcií, od hostingu webových stránok a webových aplikácií až po poskytovanie zdieľaného prístupu k disku, pripojenia tlačiarní a databázových služieb. [22]

Aplikácie vytvorené pre mobilné zariadenia často používajú vzdialené servery, napríklad na uchovanie dát a informácií vytvorených v aplikáciách.

4.1 Bezstavový protokol

Je protokol, ktorý si nedokáže zapamätať výsledky a údaje súvisiace s interakciami, ktoré riadi. Pravdepodobne najznámejší bezstavový protokol je Hypertext Transfer Protocol (HTTP) používaný na prenos webových stránok z webového servera do prehliadača. Tento protokol si nepamätá výsledok predchádzajúcej inštrukcie na prenos stránky. Existuje množstvo zariadení, ktoré sa používajú na prekonanie tohto problému: napríklad v prípade HTTP je jedným z riešení vložiť súbor cookie do klientskeho počítača, ktorý uchováva údaje na sledovanie interakcií. [9]

4.2 Metódy HTTP

HTTP definuje množinu metód, požiadaviek na označenie požadovanej akcie, ktorá sa má vykonať pre daný zdroj. Hoci to môžu byť aj podstatné mená, tieto metódy žiadosti sa niekedy označujú ako slovesá HTTP. Každý z nich implementuje inú sémantiku, ale niektoré spoločné črty zdieľa skupina z nich: napr. metóda žiadosti môže byť **bezpečná**, **idempotentná** alebo **cacheovateľná**. [4]

Metóda HTTP je *bezpečná*, ak nemení stav servera. Inými slovami, metóda je bezpečná, ak vedie k operácii iba na čítanie. Všetky bezpečné metódy sú tiež idempotentné, ale nie všetky idempotentné metódy sú bezpečné. [11]

Metóda HTTP je *idempotentná*, ak zamýšľaný účinok na server pri vykonaní jednej požiadavky je rovnaký ako pri vykonaní niekoľkých rovnakých požiadaviek. [5]

Metóda HTTP ktorá je *cacheovateľná*. Je odpoveď s možnosťou uloženia do vyrovnávacej pamäte, ktorá sa uloží na neskoršie získanie a použitie, čím sa uloží nová požiadavka na server. Nie všetky odpovede HTTP je možné uložiť do vyrovnávacej pamäte. [2]

4.3 Webové rozhranie - API

Webové API je aplikačné programovacie rozhranie pre webový server alebo webový prehliadač. Ide o koncept vývoja webu, ktorý sa zvyčajne obmedzuje na stranu klienta webovej aplikácie a preto zvyčajne nezahŕňa podrobnosti o implementácii webového servera. Serverové API pozostáva z jedného alebo viacerých verejne prístupných koncových bodov pre definovaný systém správ: žiadosť – odpoveď.

Koncové body

Koncové body sú dôležitými aspektami interakcie s webovými rozhraniami API na strane servera, pretože určujú, kde sa nachádzajú zdroje, ku ktorým môže pristupovať softvér tretích strán. Zvyčajne je prístup cez URI, na ktoré sa odosielajú požiadavky HTTP a od ktorého sa teda očakáva odpoveď. Webové rozhrania API môžu byť verejné alebo súkromné, pričom druhé z nich vyžaduje prístupový token.

Koncové body musia byť statické, inak nemožno zaručiť správne fungovanie softvéru, ktorý s nimi spolupracuje. Ak sa zmení umiestnenie zdroja (a s ním aj koncové body), softvér ktorý ich využíva sa pokazí, pretože požadovaný zdroj už nemožno nájsť na rovnakom mieste. Aby bolo možné API aktualizovať, zavádza sa verzovací systém v URI, ktorý ukazuje na koncový bod. [14]

4.4 REST

REST (*Representational state transfer*) je softvérový architektonický štýl na poskytovanie štandardov medzi počítačovými systémami na webe, čím sa systémom uľahčuje vzájomná komunikácia. [20] REST je často využívaný v architektúre klient-server. Používa sa v celom softvérovom priemysle na vytváranie spoľahlivých webových rozhraní API 4.3, ktoré spĺňajú bezstavový protokol 4.1.

Webové rozhranie API, ktoré dodržiava obmedzenia REST, sa neformálne označuje ako RESTful. Vo všeobecnosti sú webové rozhrania API RESTful založené na metódach protokolu HTTP, ako sú GET a POST. Požiadavky HTTP sa používajú na prístup k údajom alebo zdrojom vo webovej aplikácii prostredníctvom parametrov zakódovaných v URL.

4.5 Flask-RESTX

Po dlhej diskusii pod rámcom Flask-RESTPlus, ktorá bola zameraná na to, že tento rámec umiera, nakoľko jeho autori už dané zdrojové kódy neudržujú a nereagujú na vzniknuté chyby, sa komunita programátorov rozhodla, že si vytvoria vlastnú vetvu, ktorú budú môcť sami upravovať. To malo za príčinu vznik rámca *Flask-RESTX*.

Flask-RESTX je rozšírenie pre Flask, ktoré pridáva podporu pre rýchle vytváranie REST API. Flask-RESTX podporuje osvedčené postupy s minimálnym nastavením. Poskytuje

ucelenú zbierku dekoratérov a nástrojov na popis API a správne vystavenie jeho 'interaktívnej' dokumentácie (pomocou Swagger). [15]

Swagger

Swagger je sada nástrojov na písanie rozhraní API založených na REST. Zjednodušuje proces písania API, špecifikuje štandardy a poskytuje nástroje potrebné na písanie bezpečných, výkonných a škálovateľných API. Swagger štandardizuje celý proces písania rozhraní API, vďaka tomu šetrí programátorovi čas.

Swagger Editor

Swagger Editor je nástroj, ktorý pomáha overiť správnosť vytvorených API už počas vývoja v reálnom čase. Je možné ho spúšťať lokálne a poskytuje okamžitú spätnú väzbu a tak pomáha pri detekcii chýb prípadne neočakávaného chovania vytváraných API. [37]

4.6 JSON Web Token

JSON Web Token známy aj ako JWT je otvorený štandard (RFC 7519), ktorý definuje kompaktný a samostatný spôsob bezpečného prenosu informácií medzi stranami vo formáte JSON objektu. Tieto informácie môžu byť overené a dôveryhodné, pretože sú digitálne podpísané. JWT môže byť podpísaný pomocou tajomstva (s algoritmom *HMAC*) alebo párom verejných a súkromných kľúčov pomocou algoritmov *RSA* alebo *ECDSA*.

JWT môžu byť šifrované, aby sa zabezpečilo aj utajenie medzi stranami, zameriame sa na digitálne podpísané tokeny. Podpísané tokeny môžu overiť integritu prenášaných dát, ktoré sú v nich obsiahnuté, zatiaľ čo zašifrované tokeny skryjú tieto dáta pred ostatnými stranami. Keď sú tokeny podpísané pomocou páru verejný a súkromný kľúč, podpis tiež potvrdzuje to, že iba strana, ktorá má súkromný kľúč, je tá, ktorá ho podpísala. [6]

Kapitola 5

Neurónové siete

Neurónové siete, známe tiež ako umelé neurónové siete alebo simulované neurónové siete, sú podskupinou strojového učenia a sú jadrom algoritmov hlbokého učenia. Ich názov a štruktúra sú inšpirované ľudským mozgom a napodobňujú spôsob, akým si biologické neuróny navzájom presúvajú informácie.

Umelé neurónové siete sa skladajú z vrstiev uzlov, ktoré obsahujú vstupnú vrstvu, jednu alebo viac skrytých vrstiev a výstupnú vrstvu. Každý uzol alebo aj nazývaný umelý neurón sa spája s iným a má priradenú hmotnosť a prah. Ak je výstup ktoréhokoľvek jednotlivého uzla nad špecifikovanou prahovou hodnotou, tento uzol sa aktivuje a odosiela údaje do ďalšej vrstvy siete. V opačnom prípade sa do ďalšej vrstvy siete neprenesú žiadne údaje.

Neurónové siete sa spoliehajú na trénovacie údaje, aby sa naučili a časom zlepšili svoju presnosť. Keď sú však tieto algoritmy učenia doladené na presnosť, sú to výkonné nástroje v informatike a umelej inteligencii, ktoré nám umožňujú klasifikovať a zoskupovať údaje vysokou rýchlosťou. Úlohy v oblasti rozpoznávania reči alebo obrazu môžu trvať minúty oproti hodinám v porovnaní s manuálnou identifikáciou ľudskými odborníkmi. Jednou z najznámejších neurónových sietí je vyhľadávací algoritmus Google a nový online nástroj ChatGPT.[24]

Neurónová sieť sa skladá z troch typov vrstiev:

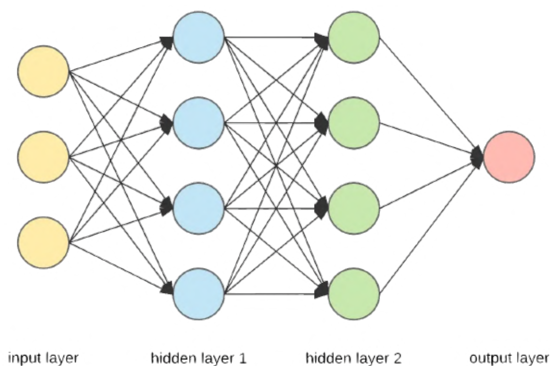
- *Vstupná vrstva* — počiatočné údaje pre neurónovú sieť.
- *Skryté vrstvy* — medzivrstva medzi vstupnou a výstupnou vrstvou a miesto, kde sa vykonávajú všetky výpočty.
- *Výstupná vrstva* — vytvorí výsledok pre dané vstupy.

Na obrázku 5.1 žlté kruhy predstavujú *vstupnú vrstvu* a zvyčajne sa označujú ako vektor **X**. Modré a zelené kruhy predstavujú *skryté vrstvy*. Tieto uzly sú nazývané aj ako „aktivačné" uzly a zvyčajne sú označené ako **W** alebo θ. Červený kruh je *výstupná vrstva* alebo ináč povedané predpokladaná hodnota (prípadne hodnoty viacerých tried/typov výstupov).

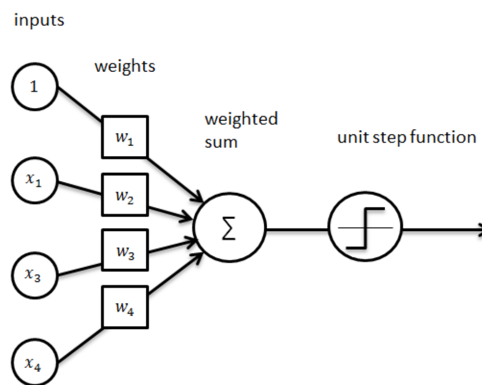
Každý uzol je spojený s každým uzlom z nasledujúcej vrstvy a každé spojenie má určitú váhu. Hmotnosť môže byť vnímaná ako vplyv, ktorý má tento uzol na uzol z ďalšej vrstvy. Vyobrazenie jedného konkrétneho uzla sa nachádza na obrázku 5.2.[34]

5.1 Detekcia objektov v obraze

Rozpoznávanie objektov je jednou z techník počítačového videnia, ktorá je kombinovanou úlohou detekcie objektov a klasifikácie obrazu. Ľudia dokážu na fotografii alebo videu iden-



Obr. 5.1: Architektúra neurónovej siete



Obr. 5.2: Detail uzla neurónovej siete

tifikovať čokoľvek, napríklad jednotlivcov alebo predmety, pretože majú na to prispôsobený zrakový orgán. Ale pre počítače identifikácia a pochopenie obsahu obrázkov je náročná. Cieľom je naučiť počítač robiť to, čo k človeku prirodzene patrí, teda získať úroveň pochopenia toho, čo obrázok obsahuje.

Hlavne v počítačovom videní je rozpoznávanie objektov a detekcia objektov ekvivalentná. Oba sa používajú na identifikáciu predmetov na digitálnych fotografiách, ale líšia sa v implementácii. Lokalizácia objektov a klasifikácia obrázkov sú zahrnuté v detekcii objektov a používajú sa na lokalizáciu objektov na obrázku, označenie ich polohy pomocou ohraničujúceho rámčeka a potom predpovedanie triedy objektu na obrázku. Detekcia objektov je podmnožinou rozpoznávania objektov, takže dokáže súčasne lokalizovať objekt a potom ho identifikovať na obrázku.

Metódy rozpoznávania objektov zvyčajne spadajú do prístupov strojového učenia a hlbokého učenia, no tieto dva prístupy majú úplne odlišné pohľady.[17]

Strojové učenie - (Machine Learning)

V prístupe strojového učenia sa musia definovať charakteristiky. Niektoré algoritmy fungujú v tomto prípade lepšie, napr.:

- *SIFT* - Scaling Invariant Feature Transformation
- *SVM* - Support Vector Machine
- *HOG* - Histogram of oriented Gradients[17]

Niektoré z týchto algoritmov ako napríklad *SIFT* je spoplatnené a tak pre ich využívanie je potrebné platiť pravidelný poplatok.

Hlboké učenie - (Deep learning)

Strojové učenie a počítačové videnie zásadne zmenilo hlboké učenie. Tento prístup využíva konvolučné neurónové siete, ktoré pomáhajú dosiahnuť najlepšie výsledky detekcie, aj keď je obraz mierne zmenený.[17]

5.1.1 Histogram orientovaných gradientov

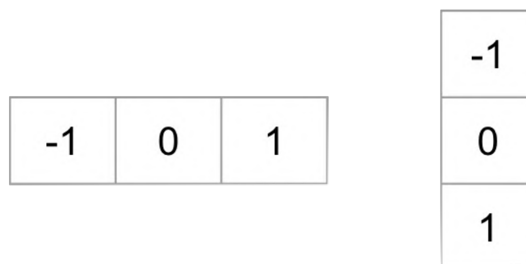
Tiež známy ako **HOG**, je deskriptorom funkcií, ako je napríklad detektor okrajov *Canny* alebo *SIFT*. Používa sa v počítačovom videní a spracovaní obrazu na účely detekcie objektov. Táto technika počíta výskyty orientácie gradientu v lokalizovanej časti. Deskriptor HOG sa zameriava na štruktúru alebo tvar objektu. Je lepší ako ktorýkoľvek deskriptor hrán, pretože na výpočet prvkov používa veľkosť, ako aj uhol gradientu. Pre oblasti snímky generuje histogramy pomocou veľkosti a orientácií gradientu.[38] Výpočet tohto deskriptora vieme rozdeliť do piatich krokov:

Predspracovanie

Obrázok môže mať akúkoľvek veľkosť. Jediným obmedzením je, že analyzované časti majú pevný pomer strán, väčšinou 1:2. Z obrázka sa teda vyberie časť, v ktorej bude ďalej prebiehať detekcia. Táto časť je orezaná a jej veľkosť sa z pravidla zmení na 64×128. Teraz je časť pripravená vypočítať deskriptor *HOG*.[32]

Vypočítanie gradientov

Na výpočet deskriptora *HOG* sa musia najprv vypočítať horizontálne a vertikálne gradienty. Sú potrebné aby bolo možné vypočítať histogram gradientov. To sa dá ľahko dosiahnuť filtrovaním obrazu pomocou jadier nachádzajúcich sa na obrázku 5.3.



Obr. 5.3: Ukážka jadier použitých na filtrovanie obrazu.

Gradient x (na obrázku v ľavo) sa vytvára primárne na zvislých čiarach, zatiaľ čo gradient y (na obrázku v pravo) na vodorovných čiarach. Veľkosť gradientu sa zväčší vždy, keď dôjde k prudkej zmene intenzity. Obrázok s vyobrazením gradientov odstráni veľa nepodstatných informácií ale zvýrazní obrysy.

Pre každý pixel má gradient veľkosť a smer. Pre farebné obrázky sa vyhodnotia gradienty troch kanálov. Veľkosť gradientu v konkrétnom pixely je maximum z veľkosti gradientov troch kanálov, a uhol je uhol zodpovedajúci maximálnemu gradientu.[32]

Vypočítanie histogramu gradientov

V tomto kroku sa obrázok rozdelí na 8×8 buniek a pre každých 8×8 buniek sa vypočíta histogram gradientov. Jedným z dôležitých dôvodov na použitie deskriptora funkcie na opis časti obrázka je to, že poskytuje kompaktnú reprezentáciu. Časť obrázka 8×8 obsahuje 8x8x3 = 192 hodnôt. Gradient tejto časti tak obsahuje 2 hodnoty (veľkosť a smer) na pixel,

čo dáva dohromady 8x8x2 = 128 čísel. Nielenže je reprezentácia kompaktnejšia, ale výpočet histogramu na ploche robí túto reprezentáciu odolnejšou voči šumu. Jednotlivé gradienty môžu mať šum, ale histogram s veľkosťou 8×8 robí zobrazenie oveľa menej citlivé na šum.

Ale prečo časť o veľkosti 8×8? Prečo nie 32×32? Je to výber dizajnu, ktorý vychádza z rozsahu funkcií, ktoré sú hľadané. *HOG* sa pôvodne používal na detekciu chodcov. Časť o veľkosti 8×8 je dostatočne veľká na to aby zachytila zaujímavé črty (napr. tvár, temeno hlavy atď.) na fotografii chodca o veľkosti 64×128.

Histogram je v podstate vektor (alebo pole) 9 čísel zodpovedajúcich uhlom 0, 20, 40, 60, . . . , 160.[32]

Normalizácia bloku

Gradienty obrazu sú citlivé na celkové osvetlenie. Ak obrázok stmavne vydelením všetkých hodnôt pixelov číslom 2, veľkosť gradientu sa zmení o polovicu a preto sa hodnoty histogramu zmenia o polovicu. V ideálnom prípade, by bol deskriptor nezávislý od variácii osvetlenia. Inými slovami histogram by bol „normalizovaný“ tak, aby nebol ovplyvnený zmenami osvetlenia.

Normalizovanie vektoru o dĺžke 3 prebieha následovne:

- Farebný vektor RGB má hodnotu [128, 64, 32]
- Dĺžka tohto vektora je $\sqrt{128^2 + 64^2 + 32^2} = 146,64$
- Vydelením každého prvku tohto vektora číslom 146,64 dostaneme normalizovaný vektor [0,87, 0,43, 0,22]

Následne je dobré normalizovať blok o veľkosti 16x16, ktorý obsahuje 4 histogramy o veľkosti 9x1. Tieto histogramy je možné zreťaziť do vektoru 36x1. Vektor je potom možné normalizovať podobne ako vektor o dĺžke 3. Následne okno o veľkosti 16x16 sa posúva o 8 pixelov po celom obrázku a výpočet normalizácie sa opakuje pre každé okno.[32]

Vypočítanie vektoru histogramu orientovaných gradientov

Na výpočet konečného vektora pre celú časť vyrezaného obrázku, sa vektory o veľkosti 36×1 zreťazia do jedného obrovského vektora. Veľkosť výsledného vektora je možné jednoducho spočítať.

Obrázok obsahuje 105 vektorov (7 horizontálnych a 15 vertikálnych, 7x15=105) o veľkosti 36x1. Keď všetky spojíme do jedného veľkého vektora, dostaneme 3780 rozmerný vektor.[32]

5.2 Keras

Keras je rozhranie pre hlboké učenie, napísané v programovacom jazyku Python, ktoré beží na platforme strojového učenia TensorFlow. Bol vyvinutý so zameraním na umožnenie rýchleho experimentovania. Schopnosť prejsť od myšlienky k výsledku čo najrýchlejšie je kľúčom k dobrému výskumu.

Ako hlavné body *Keras-u* môžme definovať:

- *Jednoduchosť* - ale nie zjednodušenie. Keras znižuje kognitívnu záťaž na vývojára, aby sa mohol viac sústrediť na tie časti problému, na ktorých skutočne záleží.

- *Flexibilnosť* – Keras si osvojuje princíp progresívneho odhaľovania zložitosti. Jednoduché pracovné postupy by mali byť rýchle a jednoduché, zatiaľ čo pokročilé pracovné postupy by mali mať jasne stanovený postup ako dosiahnuť požadovaného výsledku.
- *Výkonný* – Keras poskytuje špičkový výkon a škálovateľnosť. Používajú ho organizácie a spoločnosti vrátane NASA, YouTube alebo Waymo.[1]

5.3 Core ML

Core ML poskytuje jednotnú reprezentáciu pre všetky modely. Aplikácia využíva rozhrania Core ML API a používateľské údaje na vytváranie predpovedí a na dolaďovanie modelov, a to všetko na zariadení používateľa. Spustenie modelu výlučne na zariadení používateľa odstraňuje akúkoľvek potrebu sieťového pripojenia, čo pomáha udržiavať údaje používateľa súkromné a zároveň aplikáciu reaktívnu.

Core ML optimalizuje výkon na zariadení využívaním CPU, GPU a ANE (Apple Neural Engine) a zároveň minimalizuje nároky na pamäť a spotrebu energie.

Balík *Core ML Tools* je možné využívať na konverziu modelov natrénovaných z knižníc tretích strán, ako sú TensorFlow, PyTorch a Keras, do formátu balíka modelov *Core ML*. Potom je možné použiť *Core ML* na integráciu modelov do aplikácie. [3]

5.4 CreateML

CreateML je vhodné na vytváranie a trénovanie vlastných modelov strojového učenia na zariadení MAC od spoločnosti Apple. Modely je možné trénovať na vykonávanie úloh, ako je rozpoznávanie obrázkov, extrahovanie významu z textu alebo hľadanie vzťahov medzi číselnými hodnotami.

Model je možné trénovať na rozpoznávanie vzorov tak, že mu budú poskytnuté reprezentatívne vzorky. Je možné napríklad trénovať model na rozpoznávanie psov tak, že mu je ukázaných veľa obrázkov rôznych psov. Po natrénovaní modelu je možné ho otestovať na údajoch, ktoré predtým nevidel a tak vyhodnotiť, ako dobre plnil danú úlohu. Keď model funguje dostatočne dobre, je pripravený na integrovanie do aplikácie pomocou rámca *Core ML*.

Create ML využíva infraštruktúru strojového učenia zabudovanú do produktov Apple, ako sú Photos a Siri. To znamená, že klasifikácia obrázkov a modely prirodzeného jazyka sú menšie a tréning trvá oveľa menej času. [26]

Kapitola 6

Návrh

Návrh výslednej aplikácie je rozdelený do dvoch hlavných bodov.

Prvým bodom a zároveň aj prvou rozvrhnutou časťou bol server. V tejto časti je opísaná špecifikácia požiadavkou na navrhovaný server, databáza a koncové body za pomoci ktorých sa bude dať so serverom komunikovať.

Druhým bodom je samotná aplikácia, ktorá bude využívať server na ukladanie a získavanie dát. Návrh aplikácie obsahuje iteratívne diagramy použitia, ktoré zobrazujú zamýšľaný vývoj aplikácie. Ďalej obrázky makety užívateľského rozhrania a lokálnu databázu, ktorá bude na zariadení.

Cieľová skupina

Pre návrh cieľovej aplikácie je dôležité zistiť cieľovú skupinu užívateľov. Nakoľko má aplikácia výrazne zrýchliť a uľahčiť identifikáciu zabitej divokej zveri, je stanovenie cieľovej skupiny celkom jednoduché, poľovníci prípadne lesná stráž.

6.1 Server

Návrh serverovej časti prebiehal v dvoch logických iteráciách. V prvej iterácii sa špecifikovali základné požiadavky na výsledný server. Z týchto požiadaviek boli následne odvodené entity databázy, výsledné koncové body, ktoré budú slúžiť na komunikáciu so serverom a v neposlednom rade rámce a knižnice, ktoré čo najviac uľahčia vývoj no zároveň nie sú príliš robustné a tak zbytočne nespomalia samotný chod výsledného servera.

6.1.1 Špecifikácia požiadavkou

Špecifikácia požiadavkou prebiehala primárne po konzultáciách s vedúcim práce, nakoľko mal informácie od cieľovej skupiny užívateľov. Vďaka týmto informáciám bolo jednoduchšie vytvoriť návrh tried obsiahnutých v databáze.

Špecifikácia, ktorá najviac ovplyvnila výber rámcov a knižníc využitých na vývoj servera, bol *dôraz na dokumentáciu*. Na základe tejto špecifikácie bol teda zvolený rámec **Flask-RESTX**, o ktorom je viac v sekcii 4.5.

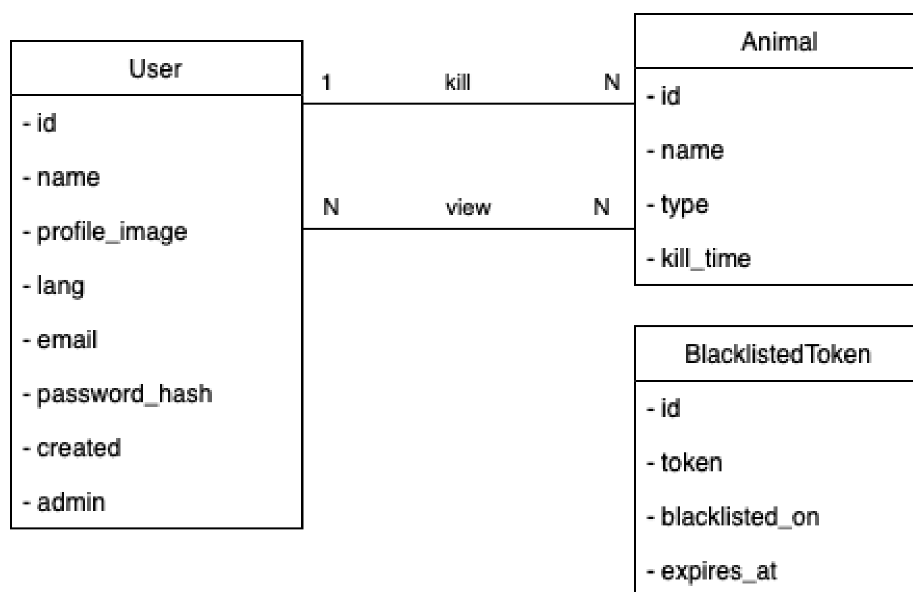
Celkový prehľad špecifikácii požiadavkou, ktoré ovplyvnili návrh servera:

- REST API
- Dôraz na dokumentáciu

- Možnosť prihlásenia
  - Admin
  - Užívateľ
- Ukladanie fotografií
- Úprava databázy aj za pomoci príkazového riadku

6.1.2 Databáza

Serverová databáza slúži na ukladanie informácií o registrovaných užívateľoch, ulovených / zabitých zvieratách a tokenoch, ktoré boli vyradené (zaradené na čiernu listinu) a užívatelia sa pomocou nich už nemôžu verifikovať. Počas návrhu databázy bol zvolený súbor nástrojov SQL nazývaný *SQLAlchemy*, pre vývoj tejto databázy.



Obr. 6.1: Doménový model serverovej databázy

Na základe doménového modelu boli namodelované tri triedy: **User**, **Animal** a **BlacklistedToken**, ktoré už obsahujú dátové typy a stručné popisy svojich atribútov. Výsledný diagram týchto tried sa nachádza na obrázku 6.2.

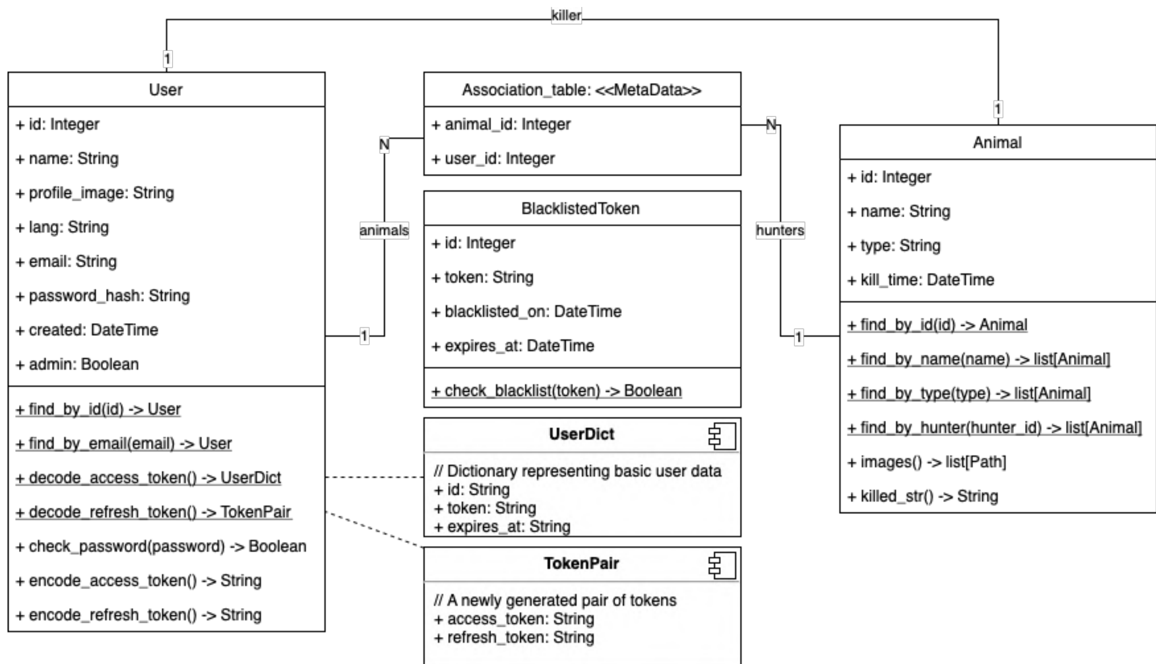
- Trieda **BlacklistedToken** obsahuje informácie o tokene, ktorý už ďalej nemožno využívať na verifikáciu užívateľa.
  - *id* [Integer] - je to primárny kľúč, ktorý označuje token jedinečným číslom, je nastavený na automatickú inkrementáciu pri pridaní nového tokenu do databázy [2]
  - *token* [String] - je reťazec reprezentujúci token, ktorý už naďalej nemôže slúžiť k verifikácií užívateľa

[2]Štruktúra riadku: *Názov atribútu* [dátový typ atribútu] - krátky popis

- *blacklisted_on* [DateTime] - označuje pomocou UTC hodnoty čas, v ktorom bol token zaregistrovaný na čiernu listinu
- *expires_at* [DateTime] - obsahuje časovú hodnotu, ktorá označuje dokedy bol daný token validným

- Trieda **User** obsahuje prihlasovacie údaje a ďalšie informácie o užívateľovi.
  - *id* [Integer] - je to primárny kľúč, ktorý označuje užívateľa jedinečným číslom je nastavený na automatickú inkrementáciu pri pridaní nového užívateľa do databázy
  - *name* [String] - je reťazec reprezentujúci meno (prezývku) užívateľa
  - *profile_image* [String] - je reťazec obsahujúci cestu k profilovej fotke užívateľa, každému novému užívateľovi je udelená predvolená profilová fotka
  - *lang* [String] - definuje užívateľom zvolený jazyk (v akom jazyku sa mu majú zobrazovať texty v rámci aplikácie)
  - *email* [String] - je unikátny textový reťazec obsahujúci hodnotu reprezentujúcu užívateľov kontaktný email
  - *password_hash* [String] - je užívateľove heslo, ktoré bolo pomocou hašovacej funkcie prevedené do reťazca znakov
  - *created* [DateTime] - označuje pomocou UTC hodnoty čas, v ktorom bol užívateľov účet vytvorený
  - *admin* [String] - je príznak [3] signalizujúci, či daný užívateľ nadobúda administrátorské práva
- Trieda **Animal** obsahuje informácie o zabitom zvierati.
  - *id* [Integer] - je to primárny kľúč, ktorý označuje zviera jedinečným číslom, je nastavený na automatickú inkrementáciu pri pridaní nového zvieraťa do databázy
  - *name* [String] - je reťazec reprezentujúci meno zvieraťa, táto hodnota je voliteľná a navrhnutá iba pre uľahčenie hľadania konkrétneho zvieraťa užívateľovi
  - *type* [String] - textový reťazec označujúci druh zabitej zveri
  - *identif* [Double] - hodnota označujúca identifikačné číslo zvieraťa, premenná pripravuje miesto na uloženie hodnoty identifikačného algoritmu
  - *kill_time* [DateTime] - hodnota označujúca čas zabitia zvieraťa
  - *killer* [User] - je referencia na konkrétnu inštanciu užívateľa, ktorý dané zviera zabil
  - *hunters* [Array<User>] - je pole užívateľov, ktorí majú prístup k získaniu informácií o danom zvierati

[3] príznak (flag) - hodnota signalizujúca určitý stav objektu, zväčša nadobúdajúca maximálne dvoch hodnôt



Obr. 6.2: Diagram databázových tried

6.1.3 Koncové body

Kľúčovým atribútom, ktorý odlišuje *REST API* od iných rozhraní je jednotné rozhranie. Určuje štandardizovaný spôsob komunikácie s daným serverom bez ohľadu na klientsku aplikáciu alebo zariadenie, na ktorom je spustený. [10]

Ďalšou veľmi dôležitou časťou *REST-API* servera sú dobre navrhnuté koncové body. Koncové body by mali byť rozvrhnuté do logických celkov, ktoré po skupinách dávajú význam, zároveň aby jednotlivé koncové body využívali správne *HTTP metódy* 4.2.

Navrhnuté koncové body vieme rozdeliť do dvoch skupín: **Account** a **Animal**, každá z týchto skupín bude určovať prefix v rámci URI.

Koncové body v skupine **Account**, sa budú nachádzať za prefixom */Account/*[4] a budú obsluhovať všetky operácie, ktoré sa vzťahujú na správu účtu užívateľa. Ako je napríklad registrácia, prihlásenie, odhlásenie, zmena hesla, a iné.

Koncové body v skupine **Animal**, sa budú nachádzať za prefixom */Animal/* a budú obsluhovať všetky operácie, ktoré sa vzťahujú na správu zabitých zvierat. Ako sú napríklad pridanie nového zvieraťa, editácia existujúceho záznamu zabitého zvieraťa, a iné.

Navrhnuté koncové body teda vyzerajú následovne:

- **Account**
  - *changePassword* [POST] - koncový bod slúžiaci na zmenu užívateľovho hesla, prístup k tomu bodu má len prihlásený užívateľ [5]
  - *edit* [POST] - koncový bod, ktorý poskytuje zmenu dát užívateľovho profilu, prístup k tomu bodu má len prihlásený užívateľ
  - *exists* [GET] - koncový bod, ktorý overí či účet už existuje

[4] * - vyjadruje ľubovolný textový reťazec

[5] Štruktúra riadku: *Názov koncového bodu* [HTTP metóda] - krátky popis

  - *login* [POST] - koncový bod slúžiaci na prihlásenie užívateľa
  - *logout* [POST] - koncový bod, ktorý zruší platnosť tokenom, ktoré práve užívateľ využíva a tým reprezentuje odhlásenie užívateľa, prístup k tomu bodu má len prihlásený užívateľ
  - *lostPassword* [POST] - koncový bod slúžiaci na zmenu hesla v prípade že užívateľ zabudol svoje staré heslo
  - *refreshToken* [POST] - koncový bod, ktorý overí platnosť užívateľových tokenov a v prípade že token na predĺženie relácie je validný, vygeneruje nový pár tokenom pre užívateľa a tým predlží platnosť relácie, prístup k tomu bodu má len prihlásený užívateľ
  - *register* [POST] - koncový bod slúžiaci k zaregistrovaniu nového užívateľa

- **Animal** - všetky koncové body v tejto skupine sú prístupné iba pre prihláseného užívateľa

  - *add* [POST] - koncový bod slúžiaci na pridanie novej zabitej zveri na server
  - *detail* [GET] - koncový bod, ktorý nájde v serverovej databáze nájde určitú zver a zobrazí všetky informácie o nej
  - *edit* [POST] - koncový bod slúžiaci na zmenu dát určitej zveri uloženej na serveri
  - *list* [GET] - koncový bod, ktorý informácie o všetkých zvieratách ku ktorým má prístup užívateľ

6.2 Aplikácia

Počas navrhovania aplikácie na mobilné zariadenia s podporou operačného systému *iOS*, bol braný dôraz na možnosť využitia čo najväčšieho množstva vstavaných funkcií. Nakoľko všetky vstavané funkcie od spoločnosti *Apple* sú veľmi dobre vyladené na ich zariadenia a zároveň ich využitie je dobre popísané v ich dokumentáciách.

Súčasťou návrhu je lokálna databáza, ktorá uschováva iba dáta, ku ktorým bude môcť užívateľ pristúpiť aj v prípade, že nebude mať internetové pripojenie. Ďalšou súčasťou je vývoj návrhu užívateľského rozhrania do takej podoby, aby užívateľ v každom bode aplikácie sa nemusel zbytočne zanárať, a tak už pri prvom použití sa vedel jednoducho orientovať v tom, kde sa v aplikácii nachádza.

Počas návrhu aplikácie bol zvolený iteratívny prístup, boli vytvorené 4 iterácie vývoja, kde v každej iterácii sa aplikácii pridáva nová funkcionalita. Po poslednej iterácii by mala aplikácia obsahovať všetky dôležité aspekty pre jej zamýšľané použitie a aj užívateľmi očakávané aspekty novodobých aplikácií ako je napríklad úprava profilu užívateľa.

6.2.1 Špecifikácia požiadavkou

Špecifikácia požiadavkou pre mobilnú aplikáciu, ktorá má slúžiť primárne poľovníkom, musí byť zameraná na funkčnosť aj v prípade, že sa daný užívateľ nachádza v zóne bez internetu, prípadne signálu. Na základe toho je potrebné aby aplikácia obsahovala aj lokálnu databázu, ktorá bude uschovávať užívateľove zmeny, ktoré môže vykonávať práve v zóne mimo internetu a následne ich propagovať na server po pripojení k internetu. Ďalšou požiadavkou bude možnosť rýchleho vytvorenia nového záznamu o zabitom kuse zvieraťa a teda vyvolanie takejto akcie by sa malo nachádzať v úvodných častiach aplikácie hneď po užívateľovom

prihlásení. Pre vytvorenie nového záznamu o zabitom zvierati je potrebné nasnímať fotodokumentáciu a ďalšou požiadavkou bude možnosť využívať kameru telefónu na snímanie. Pre uľahčenie snímania by mala aplikácia obsahovať interaktívne užívateľské rozhranie, ktoré oznamuje užívateľovi, či na zachytávanom obrázku sa nachádza potrebná oblasť zvieraťa - jej nos, prípadne iná oblasť dostatočne jedinečná na identifikáciu. Po zachytení obrázku, je potrebné obrázok vyhodnotiť čo najrýchlejšie aspoň základnou analýzou, aby užívateľ mohol obrázok nasnímať znova, v prípade že obrázok nie je dostatočne kvalitný a teda identifikácia na serveri nemusí viesť k úspechu.

6.2.2 Prípady použitia

Ako je spomenuté v opise tejto sekcie, aplikácia bude vyvíjaná v niekoľkých iteráciách. Pre každú iteráciu bol vytvorený samostatný diagram prípadu použitia. Diagramy boli vytvárané na základe analýzy špecifikácie požiadavkou, ktoré sa nachádzajú v predchádzajúcej sekcii 6.2.1. Na ich základe boli vybrané akcie, ktoré bude možné previesť v rámci aplikácie.

Tieto diagramy sa nachádzajú v sekciách A až 6.3. Diagram v danej sekcii je základným stavebným kameňom pre vývoj, nakoľko na základe neho bolo určované čo sa v danej iterácií bude vytvárať. Výsledný diagram prípadu použitia sa nachádza na obrázku 6.10.

6.2.3 Užívateľské rozhranie

Užívateľské rozhranie je veľmi dôležitým aspektom terajších mobilných aplikácií, nakoľko je na trhu s aplikáciami veľký pretlak aplikácii s podobnými funkcionalitami a ich využitím. Užívateľ si potom môže vyberať medzi rôznymi aplikáciami toho istého druhu a má tendenciu začať pravidelne využívať aplikácie, ktoré obsahujú jednoduchý, no zato vkusný dizajn.

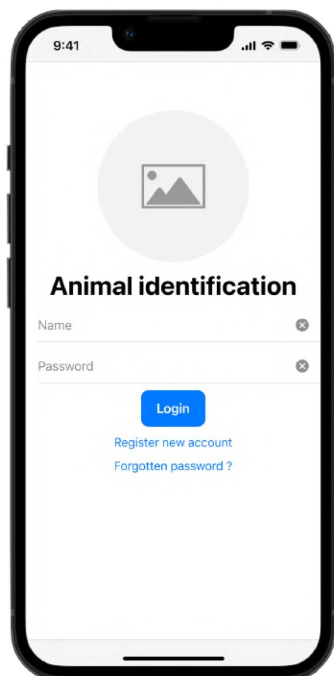
Návrh užívateľského rozhrania vyplýval z poslednej iterácie, no jeho vývoj bude prebiehať počas každej iterácie. Pre každú iteráciu bude pridaná alebo upravovaná časť užívateľského rozhrania. Vďaka tomu bude možné aplikáciu testovať v každom bode vývoja aj na reálnom zariadení prípadne simulátore. Návrh užívateľského rozhrania a zároveň tak aj interaktívna maketa tejto aplikácie, boli vytvorené pomocou aplikácie **Figma**. Výsledný návrh niektorých obrazoviek z užívateľského rozhrania sa nachádza na obrázkoch B.2, B.3, 6.5 a 6.6.

6.2.4 Databáza

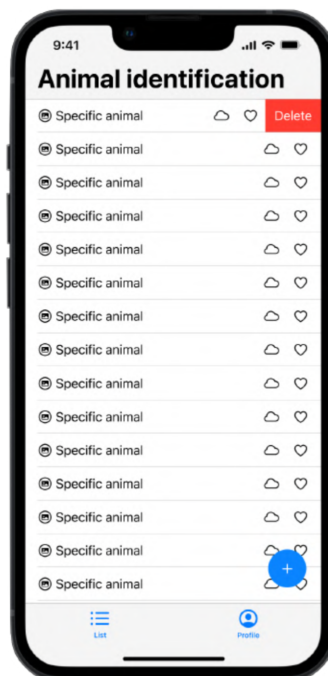
Na zabezpečenie perzistentného uloženia dát o zabitej zveri v mobilnom telefóne, bola navrhnutá databáza uchovávajúca dáta, ktoré bude možné užívateľovi prezentovať aj v prípade, že nebude mať pripojenie na internet. Počas návrhu lokálnej databázy bol braný dôraz na to, aby sa v mobilnom zariadení zbytočne neukladalo veľké množstvo dát a zároveň aby lokálna databáza bola jednoduchá a rýchla.

Okrem tejto databázy bude na uchovanie kľúčov, ktoré umožňujú komunikáciu so serverom využítá aj **Kľúčenka** 3.3. Nakoľko tokeny poskytujú možnosť verifikovať užívateľa na strane servera, je lepšie využiť práve Kľúčenku, nakoľko tá je vyvíjaná priamo spoločnosťou *Apple* a ten ju odporúča na ukladanie citlivých informácií ako sú heslá, tajomstvá, informácie o kreditných kartách a podobne.

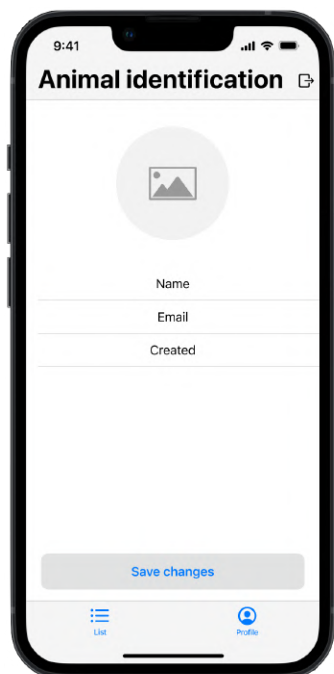
Návrh doménového modelu vyplýval z poslednej iterácie návrhu aplikácie, no jeho vývoj bude prebiehať počas každej iterácie. V niektorých iteráciách budú pridávané nové časti



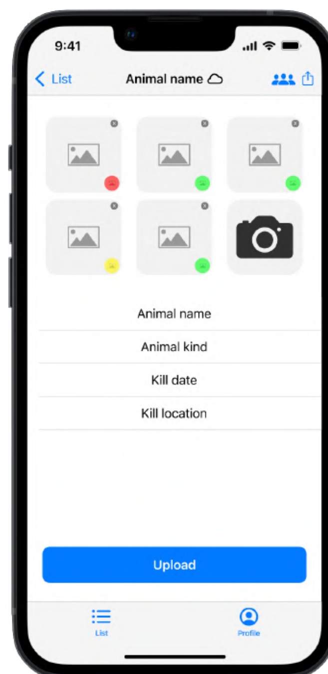
Obr. 6.3: Prihlasovacia obrazovka



Obr. 6.4: Prehľad zvierat

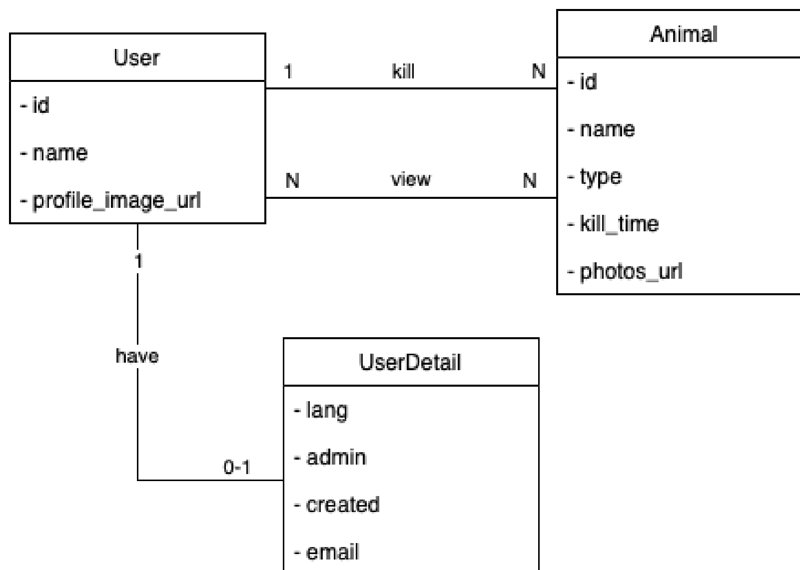


Obr. 6.5: Profil užívateľa



Obr. 6.6: Detail zvieraťa

doménového modelu. Toto chovanie zabezpečí takzvaná migrácia (správa verzií) databázy. Výsledný doménový model lokálnej databázy sa nachádza na obrázku 6.7.



Obr. 6.7: Výsledný doménový model lokálnej databázy

Na základe doménového modelu boli namodelované tri triedy lokálnej databázy: **User**, **UserDetail** a **Animal**. Trieda **UserDetail** bola vytvorená tak, aby boli oddelené základné informácie o užívateľovi od rozširujúcich. Takéto rozdelenie, poskytuje ukladanie menšieho objemu dát, nakoľko pri možnosti zobrazenia užívateľov, ktorí majú prístup ku konkrétnemu zvieraťu je postačujúce zobraziť ich základné informácie.

- Trieda **User** obsahuje informácie o užívateľovi
  - *id* [Integer] - je to primárny kľúč vygenerovaný serverom, ktorý označuje užívateľa jedinečným číslom [1]
  - *name* [String?[2]] - je textový reťazec reprezentujúci prezývku užívateľa
  - *profileImageURL* [String?] - je textový reťazec obsahujúci URL adresu na profilový obrázok užívateľa, tento obrázok je uložený na serveri a bude ukladaný iba do cache pamäte mobilného zariadenia
- Trieda **UserDetail** obsahuje rozširujúce informácie o užívateľovi
- *lang* [String?] - je textový reťazec reprezentujúci jazyk užívateľského rozhrania v rámci aplikácie
- *admin* [Bool?] - je príznak reprezentujúci, či užívateľ nadobúda administrátorské práva
- *created* [Date?] - je dátum reprezentujúci vznik užívateľského profilu
- *email* [String?] - je textový reťazec reprezentujúci emailovú adresu

[1]Štruktúra riadku: *Názov atribútu* [dátový typ atribútu] - krátky popis
[2]? - v jazyku Swift označuje hodnotu, ktorá môže byť nulová

- Trieda **Animal** obsahuje informácie o zabitej zvery
  - *id* [Integer?] - je to kľúč vygenerovaný serverom, ktorý označuje zviera jedinečným číslom
  - *appID* [UUID[6]] - je to primárny kľúč pre lokálnu databázu, vďaka nemu je možné vytvárať nové záznamy v offline režime a následne ich nahrať na server po pripojení na internet
  - *name* [String?] - je textový reťazec označujúci prezývku zvieraťa, zvolenú užívateľom, ktorý dané zviera nahral do databázy
  - *type* [AnimalType?] - je výčtový typ reprezentujúci druh zvieraťa
  - *killTime* [Date?] - je dátum reprezentujúci čas zabitia zvieraťa
  - *photosURL* [Array<String>?] - je pole textových hodnôt, kde každá hodnota predstavuje samostatnú URL adresu fotky, na ktorej sa nachádza nos daného zvieraťa
  - *localPhotos* [Array<Data>?] - je pole dát, kde každý prvok poľa je fotografia, ktorá bola uložená v offline režime a ešte nebola nahratá na server, po nahratí záznamu na server je toto pole vyprázdnené nakoľko, všetky dáta sa nachádzajú na serveri a sú prístupné cez prvky poľa *photosURL*

6.2.5 Vytvorenie fotografie

Pre poskytnutie dostatočne ostrej fotografie v okolí oblasti záujmu, je dobré túto oblasť identifikovať už pri vytváraní fotografie. Vďaka tomuto prístupu je možné automaticky zaostriť na danú oblasť a užívateľovi prezentovať na obrazovku štvoruholník, ktorý danú oblasť záujmu ohraničuje. Týmto spôsobom je užívateľ informovaný či zachytenie fotografie v danom momente bude s väčšou pravdepodobnosťou viesť k fotografii, ktorá sa môže využiť pre jedinečnú identifikáciu zvieraťa.

Nakoľko frekvencia obrázkov v zachytávacej relácii fotoaparátu, je niekoľko desiatok obrázkov za sekundu, je potrebné aby znázorňovanie oblasti záujmu bolo rýchle a nezaberalo veľa výpočtového času, nakoľko v opačnom prípade to môže viesť k obrazu, ktorý sa zasekáva, čo zapríčiní nechuť užívateľa k vyhotoveniu fotografie pomocou tejto aplikácie.

Preto počas zachytávacej relácie by mal natrénovaný model vyhodnocovať prítomnosť oblasti záujmu a následne na výstupe poskytovať súradnice oblasti pre vykreslenie štvoruholníka a možnosti zaostrenia naň. Na obrázku 6.8 je znázornené ako by malo fungovať vyobrazenie zachytávacej relácie.

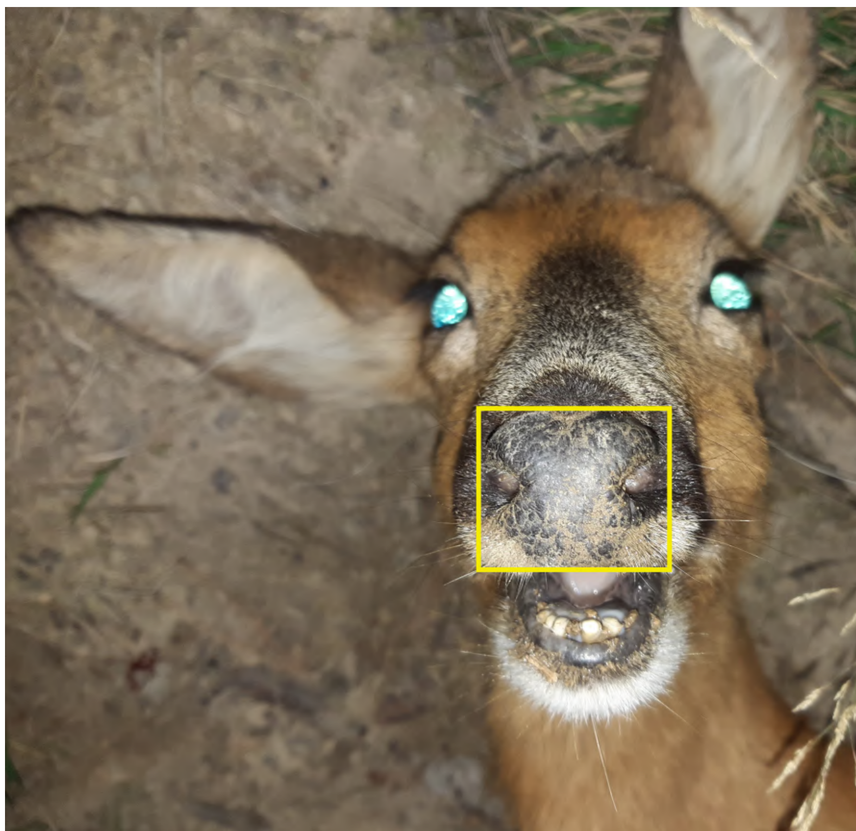
Kvalita zachytenej fotografie

Rozpoznanie kvality fotografie je pre vyvíjanú aplikáciu špecifické, nakoľko nejde iba o základné rozpoznanie či je daná fotografia dostatočne ostrá, ale aj či fotografia obsahuje oblasť záujmu. Oblasťou záujmu pre výslednú aplikáciu je **nos** (rypák) **zvieraťa**.

Na rozpoznanie či daná fotografia obsahuje dostatočnú kvalitu je potrebné natrénovať niekoľko modelov neurónovej siete. Každý model bude trénovaný na určitý druh zveri, prípadne na množinu druhov, kde medzitriedna variabilita nosov je nízka.

V prípade, že prvý model nedokáže vyhodnotiť kvalitu fotografie v reálnom čase, môže byť fotografia odoslaná na spracovanie druhému modelu, ktorý by v takomto prípade mal

[6] Univerzálny unikátny identifikátor je 128bitové číslo používané k identifikácii.



Obr. 6.8: Znázornenie oblasti záujmu vo fotografii

viac času na vyhodnotenie fotografie nakoľko by mohol bežať v druhom vlákne na pozadí aplikácie.

Výstupom modelu bude fotografia spolu s vyhodnotením, či obsahuje dostatočnú kvalitu. Nakoľko ešte neexistuje žiadne spoľahlivé riešenie tohto problému je možné túto funkcionalitu dosahovať rôznymi spôsobmi implementácie.

Jednou z možností je finálne rozhodnutie kvality obrázku na základe hodnôt poskytnutých z modelu neurónovej siete, ktorý bol použitý počas relácie snímania fotografie.

Ďalšou možnosťou je previesť obrázok na binárny, pomocou algoritmov na detekciu hrán. Na takto prevedenom obrázku sa môže detekovať počet hrán a na základe tejto hodnoty sa rozhodnúť, či výsledný obrázok obsahuje dostatočný počet hrán a teda dostatočný počet viditeľných línii na nose zvieraťa pre nasledovnú identifikáciu zvieraťa na serveri. Okrem možnosti detekovať počet hrán je možnosť detekovať mnohouholníky, nakoľko detail nosa divokej zveri pripomína skôr spleť mnohouholníkov (príklad detailnej fotografie nosa sa nachádza na obrázku 6.9).

Ďalšou možnosťou po prevedení obrázka na binárny, je použitie Gáborovej transformácie, ktorá nám poskytne jedinečné pole, ktoré ako celé môže byť použité ako jedinečný identifikátor. Tento prístup bol zvolený napríklad v štúdii 2.2, ktorá sa zaoberala dokazovaním, že nosy psov môžu slúžiť ako jedinečný identifikátor. Použitie takéhoto prístupu u divokej zveri nesie aj problémy, nakoľko psom z prednej časti nosa, ktorá bola práve skúmaná nevyrastá žiadne ochlpenie, ktoré by mohlo skresľovať výstup. Okrem toho bol vytvorený dataset čisto za účelom tejto štúdie a tak pri použití tejto metódy je možné naraziť na ďalšie problémy plynúce z použitia v nekontrolovanom prostredí.

Obr. 6.9: Detail oblasti záujmu vo fotografií

6.3 Iterácie vývoja

V tejto sekcii je priblížený iteratívny vývoj iOS aplikácie. Každá podsekcia obsahuje diagram prípadov použitia, z ktorého sa bude odvíjať vývoj aplikácie v danej iterácii. Okrem tohto diagramu podsekcie obsahujú stručný slovný popis, v ktorom je bližšie opísané na čo sa v danej iterácii zamerať a čo by malo byť výsledkom danej iterácie.

Prvá iterácia

V prvej iterácii vývoja je pred začatím vývoja potrebné inicializovať projekt a databázu, pripraviť potrebné rámce, nastaviť určité pravidlá na štýl zápisu kódu. Po úspešnom vykonaní týchto krokov sa môže začať vyvíjať aplikácia. Hierarchia aplikácie by mala dodržiavať pravidlá návrhového vzorcu **MVVM-C**, ktorý je opísaný v sekcii 3.5.3.

Nakoľko už v prvej iterácii bude potrebné pracovať s databázou a serverom, je potrebné vytvoriť triedy **DatabaseService**, **APIService** a základy tried databázových objektov, ktoré boli navrhnuté v sekcii 6.2.4. **DatabaseService** sa bude starať o **CRUD** operácie nad databázovými objektami. **APIService** sa bude starať o odosielanie požiadaviek na server a dekódovanie ich odpovedí.

Ako posledným krokom v tejto iterácii bude tvorba užívateľských rozhraní, ktoré budú interagovať na užívateľove akcie a reagovať na ne.

Výstupom tejto iterácie bude aplikácia, ktorá umožňuje užívateľovi sa zaregistrovať alebo prihlásiť. Vytvoriť nový záznam o zvierati a nahrať ho na server. Prípadne zobraziť prehľad zvierat, ktoré užívateľ nahral na server alebo vytvoril v offline režime a zatiaľ na server nenahral. Graf prvej iterácie sa nachádza v prílohe A.

Druhá iterácia

V druhej iterácii je kladený dôraz na užívateľom vytvorené záznamy o zvieratách. Tieto záznamy bude môcť užívateľ spravovať a prehliadať. Vytvorenie nového záznamu už bude poskytovať, možnosť nasnímať fotografiu zvieraťa a tak v tejto iterácii je potrebné vytvoriť a natrénovať modely neurónovej siete. Jeden druh modelu bude vykonávať identifikáciu oblasti záujmu vo fotografii a tak podávať užívateľovi spätnú väzbu už pri snímaní fotografie, či fotografia obsahuje potrebnú informáciu. Druhý druh modelu bude následne aplikovaný na získanú fotografiu a bude zisťovať kvalitu tejto fotografie. Viac o návrhu týchto modeloch sa nachádza v sekcii 6.2.5.

Výsledkom tejto iterácie bude, rozšírenie aplikácie o možnosť vyhodnotenia nasnímaných fotografií. A tým zabezpečenie pred zbytočným nahrávaním nekvalitných fotografií na server. Graf druhej iterácie sa nachádza v prílohe A.

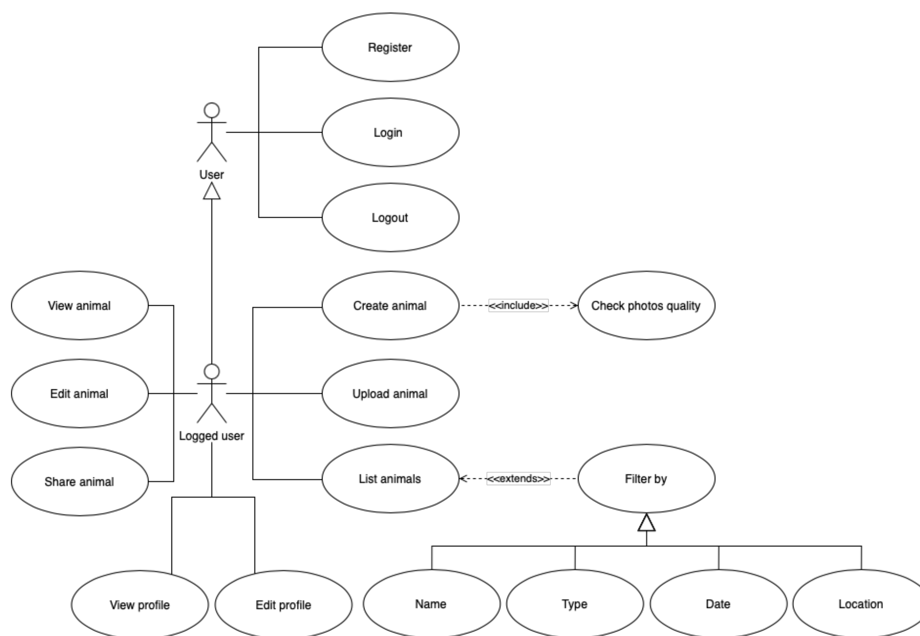
Tretia iterácia

Tretia iterácia vývoja iOS aplikácie sa bude zameriavať na dôkladné otestovanie aplikácie vzniknutej v prvých dvoch iteráciách. Počas tejto iterácie pribudnú do aplikácie funkcie, ktoré by mali užívateľovi uľahčiť hľadanie v záznamoch a tak mu spríjemniť používanie.

Po tejto iterácii bude aplikácia plne funkčná a pripravená na prípadné prevedenie do výrobného prostredia. Graf tretej iterácie sa nachádza v prílohe A.

Štvrtá iterácia

V štvrtej iterácii pribudne možnosť užívateľovi spravovať dáta o svojom profile ako je napríklad jeho profilová fotka, prípadne užívateľské meno. Štvrtá iterácia pojednáva o možnom rozšírení, kedy by jej užívatelia mohli začať aplikáciu využívať ako sociálnu sieť zameranú na menšiu skupinu ľudí. Aplikácii by mala byť pridaná možnosť zdielať záznamy o užívateľových úlovkoch.



Obr. 6.10: Diagram použitia štvrtej iterácie aplikácie

Kapitola 7

Implementácia

V tejto kapitole je opísaný vývoj servera a aplikácie. Bližšie sú rozpísané zaujímavé časti zdrojových kódov, odôvodnenie použitých nástrojov a postupov. Taktiež sa v tejto kapitole nachádza aj postup využitý pri trénovaní modelu neurónovej siete, pre účely rozpoznania nosa zvieraťa na fotografii a začlenenie tohto modelu do aplikácie.

7.1 Server

Pre implementáciu servera bolo využité vývojové prostredie **Visual Studio Code**[7] verzie 1.77. Server bol implementovaný pomocou jazyka Python. Tento jazyk bol zvolený pre jeho podporu skrz rôzne platformy a veľkú knižnicu rámcov, do ktorej komunita pravidelne prispieva. Súčasťou implementácie je rámec Flask-RESTX 4.5, ktorý pridáva podporu pre rýchle a prehľadné vytvorenie REST API servera.

Inštalácia

Pre inštaláciu na rôznych platformách bol vytvorený súbor *setup.py*, ktorý uľahčuje tento proces a to tak, že definuje všetky rámce a ich príslušné verzie, na ktorých je server závislý.

Za predpokladu, že na stroji je nainštalovaný Python s verziou 3.9.6 a vyššou, stačí zadať do terminálu `python3 setup.py install` prípadne `pip install`.

Verzovanie

Nakoľko server sa počas produkčnej fázy môže vyvíjať a meniť, je dobré byť od začiatku na takéto situácie pripravený. Preto v implementácii sú využívané takzvané „blueprints", v podstate to môže byť chápané ako návrhy verzií, ktoré určujú práve prefix jednotlivých koncových bodov. Kód zobrazujúci ukážku implementácie sa nachádza na útržku zdrojového kódu 1.

7.1.1 Architektúra

Rozdelenie súborov a zdrojových kódov odpovedá architekúre pre webové rozhrania a RESTful API. To znamená, že projekt obsahuje nasledujúcu hierarchiu:

- **api**

[7]https://code.visualstudio.com

  - **animals**
    * **dto** - súbor obsahujúci zdrojový kód, ktorý definuje parsovanie argumentov z koncových bodov a návratové modely, ktoré dané koncové body vracajú
    * **business** - súbor obsahujúci zdrojový kód, ktorý sa stará o výpočty a manipuláciu dát
    * **endpoints** - súbor obsahujúci zdrojový kód, ktorý definuje konkrétne koncové body a ich špecifikácie
  - **auth**
    * obdobne ako v priečinku *animals*

- **models** - priečinok obsahuje súbory so zdrojovými kódmi definujúcimi objekty databázy

- **utils** - priečinok obsahuje súbory so zdrojovými kódmi definujúcimi pomocné triedy a funkcie

7.1.2 Autentifikácia

Pre zabezpečenie autentifikácie užívateľa je využívaný JWT Token. Token je vygenerovaný po každom úspešnom prihlásení, prípadne registrácii užívateľa, token je následne užívateľovi zaslaný. Užívateľ je povinný mať uložený tento token nakoľko v prípade, že bude žiadať o prístup ku koncovým bodom, ktoré sú prístupné len pre registrovaných užívateľov musí token pridať do hlavičky žiadosti.

Po obdržaní tokenu na serveri, je token dekódovaný a overený či nebol expirovaný, prípadne pridaný na zoznam zakázaných tokenov. Na zoznam zakázaných tokenov sa automaticky pridavajú všetky tokeny po tom, čo ich vlastník vykonal operáciu odhlásenie.

7.2 Aplikácia

Mobilná aplikácia bola vyvíjaná pomocou vývojového prostredia XCode verzie 14.2. Aplikácia je implementovaná v jazyku *Swift*, užívateľské rozhranie je implementované pomocou rámca *SwiftUI*. Okrem rámcov vyvíjaných spoločnosťou Apple boli využité viaceré rámce od spoločnosti *Futured* ako napríklad *FTAPIKit*, ktorý zapúzdruje prácu s URL reláciami, *FuturedKit*, ktorý pridáva podporu jednoduchej navigácie medzi obrazovkami a umožňuje

```
api_bp = Blueprint("api", __name__, url_prefix="/api/v1")
api = Api(
    api_bp,
    version="1.0",
    title="Wild life identification",
    description="REST API server for wild life identification thesis.",
    doc="/documentation",
)
api.add_namespace(account, path="/Account")
```

Kód 1: Ukážka kódu vytvárajúci blueprint

vytvoriť čistejšiu architektúru, nakoľko je možné jednoduchšie oddeliť *View* a *ViewModel*. Ďalšími využitými rámcami od tejto spoločnosti boli *FormStateKit*, *BindingKit*,... Veľmi dôležitým rámcom je aj *Realm*, ktorý umožnuje vytvorenie objektovo orientovanej databázy v jazyku swift.

Spustenie

Pri prvom spustení aplikácie je inicializovaný databázový súbor, ktorý bude slúžiť na ukladanie lokálnych dát. V prípade, že aplikácia bola aktualizovaná na novšiu verziu, v ktorej sa menili definície databázových objektov, tak počas prvého spustenia novšej verzie aplikácie sú vykonávané migračné postupy zo starých databázových schém na nové. O vygenerovanie týchto migračných postupov sa stará práve rámec *Realm*.

Po úvodnom štarte aplikácie a vytvorení databázového súboru prebieha kontrola či užívateľ v aplikácii je prihlásený alebo nie. Táto kontrola prebieha overením, či sa v kľúčenke 3.3 nachádza užívateľov autentifikačný token. V prípade, že áno, je tento token využitý pri načítaní dát zo servera. Vďaka tomu, že načítanie dát prebieha z koncových bodov ktoré sú zabezpečené, je automaticky serverom overená platnosť uloženého tokenu. Ak platnosť tokenu vypršala, prebieha pokus o vygenerovanie nového tokenového páru pomocou takzvaného obnovovacieho tokenu, ktorý je tiež uložený v kľúčenke a obdržaný zo servera. Môžu nastať iba dve situácie buď obnovovací proces prebehne úspešne a serverom je vygenerovaný nový tokenový pár, ktorý je uložený do kľúčenky. Alebo platnosť oboch tokenov vypršala a užívateľ sa musí znova prihlásiť do aplikácie.

7.2.1 Užívateľské rozhranie

Užívateľské rozhranie bolo implementované pomocou rámca *SwitUI*. Je to deklaratívny prístup vytvárania užívateľského rozhrania. Programátor deklaruje z akých elementov má rozhranie pozostávať a zároveň môže aj deklarovať akciu pre každý element. Celé rozhranie je teda možné jednoducho vytvoriť pomocou kódu.

Výsledné užívateľské rozhranie vyplývalo z návrhu, ktorý sa nachádza na obrázkoch B.2 až 6.6. No najväčšou zmenou oproti návrhu je panel kariet v spodnej časti obrazovky. V návrhu a v konečnej aplikácii, obsahuje tento panel dve tlačítka, pomocou ktorých sa užívateľ môže prepínať medzi obrazovkami. Medzi nimi je pridané tlačidlo, ktoré po stlačení vykoná akciu podľa toho, ktorá obrazovka je práve vyobrazená na displeji. Toto tlačítko má samostatnú triedu s názvom *TabBarCircleObservableObject*, ktorá sa stará o jeho zobrazenie a zobrazenie správneho obrázku v ňom. Prípady akcií tlačidla na rôznych obrazovkách sú:

- **List všetkých zvierat**
  - zobrazenie obrazovky pre pridanie nového zvieraťa
- **Vytvorenie nového záznamu zvieraťa**
  - pridanie nového zvieraťa do lokálnej databázy
- **Detail uloženého zvieraťa**
  - uloženie zmien zvieraťa lokálne
  - nahratie zmien zvieraťa na server
- **Profil užívateľa**

- uloženie zmien užívateľa lokálne
- nahratie zmien užívateľa na server
- odhlásenie užívateľa

Ak obrazovka obsahuje viacero akcií, ako napríklad detail uloženého zvieraťa, je možné vždy vykonať akciu s vyššou prioritou. Nakoľko propagácia lokálnych zmien na sever je závislá od toho či vôbec sú lokálne zmeny v dátach. Uloženie dát lokálne má vyššiu prioritu.

Každý element užívateľského rozhrania, ktorý vyvoláva asynchrónnu akciu obsahuje možnosť zobrazenia načítavania. Počas tohto zobrazenia je daný element zablokovaný pre užívateľove interakcie, pokiaľ sa daná akcia nedokončí či úspešne alebo neúspešne. V prípade že sa daná akcia vykonala neúspešne, je užívateľovi zobrazené upozornenie s lokalizovaným chybovým oznámením. Všetky chyby, ktoré sa v aplikácii vyskytnú, sú odosielané do triedy s názvom *NavigationObservableObject*, ktorá inicializuje model upozornenia. Po úspešnej inicializácii modelu je oboznámená hlavná obrazovka o jeho existencii a tá tento model prezentuje užívateľovi. Vďaka takémuto prístupu je možné všetky chyby vzniknuté počas chodu aplikácie ukladať, pre prípad ak by bolo potrebné opakovať akcie, ktoré tieto chyby vyvolali.

NavigationObservableObject okrem prezentovania chybových oznámení, udržuje aj povedomie o tom, aká obrazovka je práve zobrazená na displeji. Na základe toho každá akcia, ktorá chce vyvolať zobrazenie inej obrazovky na displej ako je aktuálna, musí túto zmenu propagovať do tejto triedy. Po uložení novej hodnoty je začína animácia zobrazenia príslušnej obrazovky.

7.2.2 Ukladanie lokálnych dát

Po úspešnom obdržaní dát zo servera, sú tieto dáta uložené do lokálnej databázy pre ich možné zobrazenie užívateľovi aj v prípade, že nebude mať pripojenie k internetu. Nakoľko je malá pravdepodobnosť, že sa dáta budú meniť v krátkych časových horizontoch, tak okrem uloženia dát do lokálnej databázy, ktorá udržuje dáta aj po ukončení aplikácie, sú dáta ukladané do vyrovnávacej pamäte. Po každom uložení dát je vygenerovaná časová značka, ktorá označuje čas, v ktorom boli dáta stiahnuté zo servera. V prípade, že užívateľ chce zobraziť dáta, ktoré majú svoju lokálnu kópiu vo vyrovnávacej pamäti a zároveň čas od ich posledného stiahnutia nie je väčší ako 15 minút, zobrazia sa už uložené dáta. Toto riešenie šetrí spotrebu mobilných dát užívateľovi a zároveň aj odľahčuje server pre prípad, že aplikáciu bude používať veľké množstvo užívateľov. V prípade, že užívateľ explicitne vyjadrí, že chce dostať aktuálne dáta, čo sa v aplikáciach prejavuje ako takzvaná akcia „pull to refresh“, je ignorovaná časová značka a je vytvorený dotaz na server pre dané dáta. Príklad funkcie, ktorá zabezpečuje toto chovanie sa nachádza v ukážke kódu 2.

Ukladanie lokálnych zmien

Nakoľko užívateľ nemusí mať vždy prístup k internetu ale môže robiť zmeny v dátach, ktoré by neskôr rád propagoval na server, je potrebné vedieť rozlíšiť, či uložené dáta boli lokálne pozmenené alebo nie. Za týmto účelom bol do lokálnej databázovej schémy pridaný príznak, ktorý indikuje, že dáta boli lokálne zmenené, a teda sa líšia od tých, čo sa vyskytujú na servery. Tento príznak je automaticky nastavený, keď užívateľ zvolí možnosť lokálneho uloženia modifikovaných dát.

Postup pridávania nových fotografií je rozdielny od vytvárania zmien v jednoduchých dátových štruktúrach, ako je napríklad meno, dátum,. . . . Tento postup sa líši v tom, že je

```
@MainActor
func getList(force: Bool) async {
    guard lastReload == nil
    || lastReload!.addingTimeInterval(Constants.reloadTimeout) <= Date()
    || force
    else {
        if !resources.hasContent {
            let objects = realmService.getObjects(of: Object.self)
            resources.content = objects
        }
        return
    }
    Task {
        resources.isLoading = true
        do {
            let data = try await objectService.getList()
            for object in objects {
                let success = try realmService.update(object: object)
                if !success {
                    try realmService.save(object)
                }
            }
            resources.content = realmService.getObjects(of: Object.self)
            self.lastReload = Date()
        } catch {
            delegate?.raiseAlert(.init(error: error))
        }
        resources.isLoading = false
    }
}
```

Kód 2: Ukážka funkcie, ktorá využíva vyrovnávajúcu pamäť a lokálnu databázu

potrebné držať binárne dáta fotografií v zariadení, až do doby nahratia na server. Nakoľko po nahratí dát fotografií na server by bolo zbytočné dáta držať aj v binárnej podobne na zariadení, tak sú tieto dáta vymazané a fotografie sú načítané zo vzdialeného servera. Nakoľko sú všetky fotografie nahrané na serveri, stačí lokálne ukladať iba ich URL adresy a dané fotografie načítať do vyrovnávacej pamäte a vyobraziť na displeji zariadenia.

7.2.3 Práca so vzdialeným serverom

Pripojenie k serveru je rozčlenené do niekoľkých štruktúr a jednej triedy, kde každá štruktúra predstavuje jeden konkrétny koncový bod na serveri a trieda predstavuje samotný server. Trieda definujúca server sa nazýva *APIServer* a implementuje stratégie zašifrovania a dešifrovania správ posielaných a prijímaných zo vzdialeného servera. Ďalej sa stará o vytvorenie samotného dotazu, ktorý bude zaslaný na server. V prípade, že je v kľúčenke 3.3 uložený užívateľov identifikačný token, je automaticky pridávaný do hlavičky dotazu práve

v tejto triede. Ďalej sa tu nachádza definícia URL adresy, ktorá je prefixom pre každý koncový bod, toto umožňuje, že v prípade migrácie servera je postačujúce zmeniť iba jednu konštantu aplikácie.

Nakoľko server podporuje pre konkrétny koncový bod vždy iba jeden typ HTTP dotazu, je potrebné aby tento typ obsahovala štruktúra definujúca samotný koncový bod. Každá štruktúra definuje vlastnú funkciu pomocou ktorej bude vytváraná. Parametrami tejto funkcie sú vždy všetky dáta, ktoré je potrebné zaslať na server. O logiku toho ako budú tieto dáta posielané, či budú súčasťou URL alebo budú obsiahnuté v rámci tela dotazu, prípadne iným spôsobom, sa stará daná štruktúra. V prípade, že koncový bod vracia odpoveď zo servera, je potrebné definovať objekt, ktorý má daná štruktúra od servera očakávať. Jedinou podmienkou tohto objektu je, že musí implementovať funkciu na dekódovanie. Toto zabezpečuje protokol *Decodable*, ktorý je vyžadovaný pri definícii štruktúry koncového bodu. Ukážka jednej z využitých štruktúr sa nachádza na útržku kódu 3.

```
struct EditAccountEndpoint: MultipartEndpoint, ResponseEndpoint {
    typealias Response = User

    let method: HTTPMethod = .post
    let path: String = "Account/edit"
    let parts: [MultipartBodyPart]

    init(name: String?, email: String?, image: Data?, lang: String?) {
        var parts = [MultipartBodyPart]()
        if let name { parts.append(.init(name: "name", value: name)) }
        if let email { parts.append(.init(name: "email", value: email)) }
        if let lang { parts.append(.init(name: "lang", value: lang)) }
        if let image {
            parts.append(
                .init(
                    headers: [
                        "Content-Type": "image/jpeg;",
                        "Content-Disposition": "form-data; name=\"image\";
                        ↪  filename=\"profile_image.jpeg\"",
                    ],
                    data: image
                )
            )
        }
        self.parts = parts
    }
}
```

Kód 3: Štruktúra definujúca koncový bod pre úpravu užívateľových dát

7.2.4 Vytváranie fotografie

Pred každou akciou vloženia fotografie do aplikácie, je užívateľovi zobrazené modálne okno upozornenia, na ktorom si môže zvoliť odkiaľ chce danú fotografiu importovať. Užívateľ má na výber dve možnosti, vytvoriť novú fotografiu pomocou kamery priamo v aplikácii alebo nahrať už existujúcu fotografiu zo svojej galérie. V prípade, že si užívateľ zvolí importovať fotografiu z galérie je postup rozpoznávania nosa vo fotografii rovnaký ako v prípade, keď užívateľ fotografiu nasníma pomocou aplikácie.

Ak si užívateľ vybral možnosť vytvorenia novej fotografie, je mu prezentované nové modálne okno, na ktorom je spustená relácia *AVCaptureSession*, ktorá umožňuje využívať vstavanú kameru mobilného zariadenia. Pred prvým zobrazením tejto relácie, je užívateľ požiadaný o udelenie práv aplikácii na používanie fotoaparátu, až následne je možné spustiť reláciu snímania. Bez udelených práv od užívateľa nie je možné reláciu spustiť. Vytvorenie dotazu a nastavenie obrazovky, na ktorej je prezentované snímané okolie pomocou kamery mobilného zariadenia sa nachádza v triede *PhotoPickerViewController*. Táto trieda implementuje protokol *AVCaptureVideoDataOutputSampleBufferDelegate*, ktorý umožňuje reagovať na každú zachytenú snímku pomocou funkcie `func captureOutput(_: AVCaptureOutput, didOutput _: CMSampleBuffer, from _: AVCaptureConnection)`.

Na každej zachytenej snímke je následne aplikované volanie natrénovaného modelu neurónovej siete. Snímke nie je potrebné meniť veľkosť a ani pomer strán, nakoľko na volanie modelu neurónovej siete je využitý rámec *VisionKit*. Tento rámec dokáže automaticky upraviť snímku na požadovanú požadovanú veľkosť modelu. Tento krok môže byť automaticky vykonaný, iba ak má využívaný model neurónovej siete správne definovaný očakávaný vstup.

Ak model dokáže rozpoznať na zachytenej snímke nos zvieraťa, na obrazovke je vykreslený štvorec na súradniciach poskytnutých výstupom modelu. Okrem označenia nosa na obrazovke, užívateľské rozhranie oznamuje užívateľovi „dôveru“ v označenie pomocou sfarbenia tlačidla na vytvorenie fotografie.

V momente, keď sa užívateľ rozhodne vytvoriť fotografiu, stlačí príslušné tlačidlo. Pred uložením zachytenej fotografie je možné ju orezať na súradniciach poskytnutých modelom neurónovej siete. Vďaka tomu je možné zmenšiť veľkosť uložených dát na minimum. Kód implementujúci detekciu nosa na snímke sa nachádza na útržku kódu 4.

Následné uloženie vytvorenej fotografie prebieha zápisom do lokálnej databázy a nahratím na server, tento postup je priblížený v podkapitole **Ukladanie lokálnych zmien** 7.2.2. Okrem fotografie, sa ukladá aj hodnota dôvery rozpoznaného nosa, táto hodnota je produkovaná modelom neurónovej siete a ako jej názov napovedá indikuje dôveru, že v poskytnutých koordinátoch rozpoznaného objektu sa nachádza práve nos zvieraťa. Táto hodnota je využívaná ako vyhodnotenie kvality vytvorenej fotografie.

```
func captureOutput(_: AVCaptureOutput, didOutput sampleBuffer:
↪ CMSampleBuffer, from _: AVCaptureConnection) {
    guard let pixelBuffer: CVPixelBuffer =
    ↪ CMSampleBufferGetImageBuffer(sampleBuffer) else { return }
    let ciimage = CIImage(cvPixelBuffer: pixelBuffer)
    let request = VNCoreMLRequest(model: model) { finishedReq, _ in
        guard let results = finishedReq.results as?
        ↪ [VNRectangleObservation] else { return }
        guard let observation = results.first else { return }
        self.updateRectanglePosition(ciimage: ciimage, boundingBox:
        ↪ observation.boundingBox)
    }

    try? VNImageRequestHandler(ciImage: ciimage).perform([request])
}

private func updateRectanglePosition(ciimage: CIImage, boundingBox: CGRect)
↪ {
    let rect = boundingBox.applying(
        CGAffineTransform(scaleX: ciimage.extent.size.width, y:
        ↪ ciimage.extent.size.height)
    )
    let path = UIBezierPath(rect: rect)
    DispatchQueue.main.async {
        self.rectangleLayer.path = path.cgPath
    }
}
```

Kód 4: Ukážka vykreslenia ohraničenia nosa na snímke počas relácie snímania

7.3 Model neurónovej siete

Pre účely trénovania modelu neurónovej siete, ktorá ma za úlohu detekovať nos zvieraťa vo fotografii bolo potrebné roztriediť poskytnutý dataset a vybrať iba fotografie, ktoré spĺňajú požiadavky. Týmito požiadavkami boli:

- Fotografia obsahuje nos zvieraťa a je jasne viditeľný
- Ostrosť fotografie v mieste nosa
- Viditeľné detaily na nose zvieraťa

Anotovanie dát

Nakoľko neexistuje žiadny verejne dostupný model, ktorý by dokázal automatizovane vybrať iba tie fotografie, ktoré spĺňajú požiadavky a následne by postačovala validácia vybraných fotografií, bolo potrebné ručne vykonať odstránenie nevhodných obrázkov z datasetu. Po odfiltrovaní fotografií, ktoré nespĺňali požiadavky sa museli vytvoriť anotácie k týmto

dátam. Tento krok musel prebiehať bez pomoci automatizácie. Za týmto účelom bol využitý program **LabelStudio**[8], ktorý umožňuje importovať dáta rôzneho druhu a nastaviť si užívateľské rozhranie so štítkami, ktoré sa budú pridávať k anotáciám dát a exportovať vytvorené anotácie do viacerých preddefinovaných štruktúr.

Do poskytnutého datasetu sa po častiach pridávali nové fotografie nasnímané od rôznych poľovníkov, kvôli tomu boli vytvorené pomocné skripty v jazyku *Python*, ktoré porovnávali starý dataset s aktualizovaným a vytvorili súbor s preddefinovanou štruktúrou, ktorý je možné importovať do programu **LabelStudio** a tak sa do tohto prostredia propagovali aktualizácie datasetu.

Program *LabelStudio* síce poskytuje viaceré druhy štruktúr, do ktorých je možné exportovať anotované dáta, ale nepodporuje formát, ktorý využíva program **CreateML** 5.4, ktorý bol vyžitý na trénovanie modelu neurónovej siete. Aby bolo možné využívať program **CreateML**, bol vytvorený skript, ktorý prejde všetky vytvorené anotácie a ich záznamy pretransformuje a uloží do súboru typu *json* s ktorým vie pracovať aj program **CreateML**.

7.3.1 Trénovanie modelov

Pri detekcii nosa zvierat existuje množstvo prístupov, ako napríklad využitie predtrénovaných modelov alebo natrénovanie vlastného modelu neurónovej siete. V tejto sekcii je rozobrané skúmanie oboch prístupov.

Predspracovanie obrázka

Pre dosiahnutie najlepších výsledkov pri trénovaní modelov neurónových sietí je potrebné aby trénovacie dáta boli normalizované. V tomto prípade išlo o zmenu veľkostí fotografií na fixnú výšku a šírku rovnú **416 pixelov**. Poskytnuté fotografie boli od viacerých poľovníkov a zachytené pomocou viacerých zariadení, ktoré vyprodukovali fotografie s rôznymi pomermi strán.

Pre normalizáciu takýchto fotografii bola zvolená technika, ktorá sa označuje ako „letterboxing“. Pri použití tejto techniky sa najskôr normalizuje veľkosť obrázka na požadovanú výšku a šírku pri zachovaní rovnakého pomeru strán ako bol pôvodný obrázok. Následne sa pridajú čierne okraje ako výplň do požadovanej veľkosti. Takto bolo umožnené zachovať pomer veľkosti objektu záujmu vo fotografii.

Tento postup bol vykonávaný na už anotovaných dátach. Anotácie dát udávali súradnice nosa na fotografii pomocou relatívnych súradníc. Všetky anotácie boli overené a prípadne pozmenené na správne súradnice v normalizovaných fotografiách. Po úprave anotácií bol využitý skript, ktorý relatívne súradnice anotácií previedol do reálnych súradníc. Vzorec výpočtu pre každú súradnicu vyzeral nasledovne `real = relative / 100.0 * 416`

YOLOv2

YOLOv2 (You Only Look Once v2) je rýchly a efektívny prístup na detekciu objektov v reálnom čase, ktorý používa konvolučnú neurónovú sieť a upravený algoritmus non-maximum suppression (NMS) na filtrovanie výstupov.

Dotrénovávanie modelu *YOLOv2*, bolo riešené pomocou programu **CreateML**, ktorý je vyvíjaný spoločnosťou Apple. Tento program podporuje trénovanie viacerých druhov neurónových sietí. Je potrebné programu poskytnúť cestu k priečinku, v ktorom sú uložené

[8]https://labelstud.io

všetky fotografie, na ktorých bude trénovať neurónovú sieť a jeden anotačný súbor vo formáte *JSON*, ktorý obsahuje anotácie všetkých fotografií v danom priečinku. Poskytnutý dataset si automaticky rozdelí a jeho časť použije aj na účely validácie.

Počas trénovania je možné sledovať vývoj „loss funkcie“, ktorá udáva chybovosť modelu v určitej iterácii, čím je táto funckia menšia, tým by mal model lepšie dokázať vykonávať akcie, na ktoré je trénovaný. Okrem sledovania vývoja tejto funkcie je možné si vytvárať takzvané „snapshots“. Tie umožňujú vytvoriť obraz modelu neurónovej siete z danej iterácie. Tieto obrazy ale nie sú plnohodnotnými modelmi. Ich význam je skôr na validáciu toho, či trénovanie modelu postupuje tak ako by sa od neho očakávalo. Túto možnosť je možné overiť napríklad pomocou vloženia dát do obrazu a ten poskytne svoj výstup.

V programe je možné určovať hodnoty hyper parametrov *batch*, *grid size* a *iterations*. Parameter *batch* určuje počet vstupných fotografií, ktoré sú spracované súčasne v jednom kroku trénovania siete. To znamená, že namiesto toho, aby sa sieť trénovala na jednej fotografii za jednu iteráciu, sú do siete načítané a spracované naraz viaceré fotografie v rovnakom kroku.

Parameter *grid size* určuje počet regiónov v každej fotografii, ktorá je skenovaná na detekciu objektov. Pri trénovaní sa používa tzv. posuvné okno. Toto okno sa postupne aplikuje na každú oblasť fotografie, aby sa zistilo, či sa v nej nachádza objekt. „Grid“ je mriežka oblastí, ktoré sa postupne skenujú. Veľkosť mriežky určuje počet regiónov na fotografii, môže mať vplyv na presnosť detekcie objektov a rýchlosť trénovania siete.

Od veľkostí týchto hyper parametrov závisí dosiahnutá úspešnosť siete. Správne nastavenie týchto parametrov môže zlepšiť rýchlosť trénovania a zlepšiť presnosť detekcie objektov.

Veľkosť mriežky by mala byť v pomere 1:1, nakoľko pomer strán fotografií je taktiež rovný 1:1. Veľkosť mriežky bola pokusmi určená na 4x4 až 8x8. Menšia veľkosť ako 4x4 už zaostávala s detekciou objektu, zatiaľ čo mriežky väčšie ako 8x8 taktiež zaostávali s detekciou objektu a zároveň čas na ich trénovanie exponenciálne narastal.

Veľkosť *batch* vykazoval najlepšie výsledky pri hodnote 64, podobne ako u mriežky nižšia hodnota zaostávala s detekciou objektu a vyššia hodnota navyšovala čas trénovania modelu bez pridanej hodnoty.

R-CNN

Region-based Convolutional Neural Network (R-CNN) je jeden z prístupov na detekciu objektov vo fotografiách pomocou neurónových sietí. R-CNN rozdeľuje fotografiu na regióny, ktoré sú následne klasifikované pomocou konvolučnej neurónovej siete.

Boli vykonané poskusy o trénovanie plytkých ale aj hlbokých modelov R-CNN na detekciu nosa zvierat. Hlboké modely sa skladali z približne šestnástich vrstiev, pričom sa očakávalo, že hlbší model dosiahne lepšie výsledky.

Pri tvorbe modelov Region-based Convolutional Neural Network (R-CNN) bola využitá knižnica *TensorFlow*, ktorá je široko používaná pri tvorbe a trénovaní modelov neurónových sietí. Navrhnuté modely sa skladali z rôznych vrstiev, ktoré sa striedali v určitom poradí. Medzi tieto vrstvy patrili *convolution*, *max pooling* a *batch normalization*. Tieto vrstvy boli aplikované niekoľkokrát za sebou, aby sa zvýšila presnosť detekcie objektov v fotografiách. Ukážka použitia týchto vrstiev sa nachádza na útržku kódu 5.

Pre trénovanie týchto modelov sa použil optimalizačný algoritmus *Adam*. Tento algoritmus kombinuje výhody adaptívneho gradientového spúšťania a momentum-based spúšťania, čo vedie k rýchlejšiemu trénovaniu a efektívnejšiemu vyhľadávaniu minima chybovej funkcie.

```
inputs = Input(shape=input_shape)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
```

Kód 5: Ukážka kódu s definíciou vrstiev modelu R-CNN

Výsledky trénovania týchto modelov však nedosiahli lepšiu úspešnosť detekcie nosa ako dotrénovaný model YOLOv2. Dotrénovaný model YOLOv2 sa teda ukázal ako efektívnejší na detekciu nosa zvierat ako trénovanie nových modelov R-CNN.

YOLOv4

Novšia verzia modelu neurónovej siete YOLO vykazuje lepšie výsledky v detekcii objektov a detekcia prebieha rýchlejšie, zároveň tento model patrí medzi modely najviac využívané na detekciu objektov v obraze. Nakoľko modelu natrénovanému na jeho predchádzajúcej verzii sa darí detekovať oblasť nosa v aplikácii, bolo očakávanie že, natrénovanie modelu na jeho novšej verzii spresní detekciu nosa.

Na trénovanie modelu neurónovej siete YOLOv4 bol využitý rámec *darknet*[9], ktorý umožňuje trénovanie viacerých modelov. Tento rámec umožňuje okrem nastavenia vlastných hyper parametrov spomenutých v sekcii 7.3.1 aj nastavenie augumentácie dát, prípadné nastavenie voľby počtu grafických kariet, dokonca aj úpravu vrstiev modelu ale táto úprava nie je odporúčaná.

Využitý model v aplikácii

Najlepšie výsledky v aplikácii dosahoval model neurónovej siete dotrénovávaný na modeli YOLOv2. Modely, ktoré boli trénované ako úplne nové, sa do prostredia aplikácie ani nedostali, nakoľko nedosahovali dobré výsledky už počas fázy trénovania. Model dotrénovávany na modeli YOLOv4, vykazoval horšie výsledky v aplikácii aj napriek tomu, že zdokumentovaná výkonnosť tohto modelu je lepšia ako výkonosť jeho predchádzajúcich verzií. Tento jav mohol byť zapríčinený tým, že model z predchádzajúcej verzie bol trénovaný pomocou programu vyvíjaným spoločnosťou Apple. V programe môžu byť napríklad upravené vrstvy pôvodného modelu alebo ináč upravený proces trénovania, ktorý zabezpečuje lepšiu výkonosť modelu v aplikácii. Ďalším faktorom môže byť to, že model trénovaný na verzii YOLOv4 musel byť konvertovaný pomocou vlastného skriptu na typ *mlmodel*, ktorý podporuje operačný systém iOS.

[9] https://github.com/AlexeyAB/darknet

Kapitola 8

Testovanie

Existuje niekoľko prístupov testovania častí a aj výsledných aplikácií a programov. Každý prístup ma svoje plusy ale aj mínusy.

Postupy, ktoré boli použité na testovanie servera, aplikácie a vyvíjaných modelov neurónovej siete sú opísané v jednotlivých sekciách tejto kapitoly spolu s odôvodnením prečo bol daný prístup testovania použitý. Okrem postupov sa tu nachádzajú aj zistené problémy, ktoré bolo potrebné odstrániť.

8.1 Server

Počas vývoja servera bolo zvažované napísanie jednotkových testov. Ale nakoľko dátové štruktúry vytvorené na serveri nie sú rozsiahle, a tak ani operácie nad nimi veľmi zložité, bol server testovaný až po vytvorení koncových bodov, volaním jednotlivých koncových bodov. Každý koncový bod vyvoláva akcie nad uloženými dátami. Hlavným dôvodom testovania až v tejto fáze, bola možnosť využívať automaticky generovanú interaktívnu dokumentáciu. Teda počas testovania bolo možné naraz overiť správnu funkcionalitu koncového bodu a aj to, či jeho opis vygenerovaný v dokumentácii je dostatočný.

Počas tohto testovania boli zistené drobné nedostatky medzi dokumentáciou a očakávanými parametrami funkcií vyvolanými daným koncovým bodom.

8.2 Aplikácia

Vývoj aplikácie nezodpovedal presne iteráciám, ktoré boli vytvorené počas fázy návrhu, ale aplikácia sa nakoniec vyvíjala vo väčších celkoch. Preto testovanie prebiehalo primárne simulovaním užívateľských testov, kedy simulovaný užívateľ vyvolával akcie svojim používaním aplikácie a očakával určité správanie.

V aplikácii sa využíva veľa knižníc a rámcov tretích strán, ktoré sú aktívne spravované a často majú aj napísané jednotkové testy. Vďaka ich využitiu bolo možné sa vyvarovať písaniu vlastných jednotkových testov na najnižšej úrovni. Ostatné operácie a akcie bolo dostačujúce otestovať využívaním aplikácie.

Počas tejto fázy boli odhalené primárne chyby neinteraktívneho užívateľského rozhrania. Vďaka tomu bolo na veľa miestach pridané napríklad zobrazenie načítania prípadne zmenená ikona, ktorá by mala lepšie prezentovať vyvolávanú akciu.

8.2.1 Prepojenie aplikácie a servera

Počas prepájania aplikácie so serverom bolo zistených viacero chýb. Medzi prvými bolo nesprávne posielanie určitých formátov dát, ako napríklad formát dátumu. Ďalšou chybou odhalenou počas prepájania aplikácie so serverom bolo vytváranie príliš veľa dotazov na server. Aplikácia pred každým zobrazením dát vytvárala nový dotaz bez možnosti využitia už načítaných dát. Toto mohlo mať za následok preťaženie servera v prípade užívania aplikácie väčším počtom užívateľov. Vďaka tomuto zisteniu sa do aplikácie pridalo načítavanie dát do vyrovnávacej pamäte 2.

Po implementácii funkcionality, ktorá umožňuje vytvárať fotografie v rámci aplikácie bol zistený problém, že server nedokázal prijímať dáta fotografie z dotazu, ktorý bol vytváraný v aplikácii. Pri zistení tohto problému boli vyskúšané viaceré možnosti formátovania dát fotografie, tak aby ich server dokázal dekódovať. Žiadny z týchto formátov neviedol k správnemu nahratiu fotografie na server. Nakoniec bolo zistené, že server nemá problém s formátom dát, ale so spôsobom ich umiestnenia v dotaze. Na základe tohto zistenia bolo potrebné tento dotaz prepísať a využiť možnosť vlastného vytvárania hlavičiek dotazu. Príklad tvorby vlastnej hlavičky, ktorý sa využíva v aplikácií sa nachádza na obrázku 3.

8.3 Modely neurónovej siete

Testovanie modelov môžme rozdeliť do troch fáz, prvou fázou bolo overovanie správnosti modelu počas jeho trénovania. Zo začiatku sa využíval väčší dataset s väčšou rozmanitosťou dát. Toto môžme chápať tak, že do datasetu boli začlenené aj fotografie, ktoré spĺňali aspoň dve z troch uvedených kritérií spomenutých v sekcii o implementácii modelov neurónových sietí 7.3. Počas trénovania na takomto datasete hodnoty loss funkcií mali veľký rozsah a nikdy sa neblížili nulovej hodnote. Preto sa už v prvej fáze postupne vyradili z datasetu všetky fotografie, ktoré nespĺňali všetky tri podmienky. Po úprave datasetu sa hodnoty loss funkcie začali vylepšovať.

Druhou fázou testovania bolo využívanie natrénovaného modelu na vytvorenie rámca znázorňujúceho nos zvieraťa na fotografii. V tejto fáze sa dataset ešte zmenšil na výslednú veľkosť *1668 anotovaných fotografií*. Z datasetu boli odstránené fotografie, ktoré by sme mohli označiť ako outlinery[1].

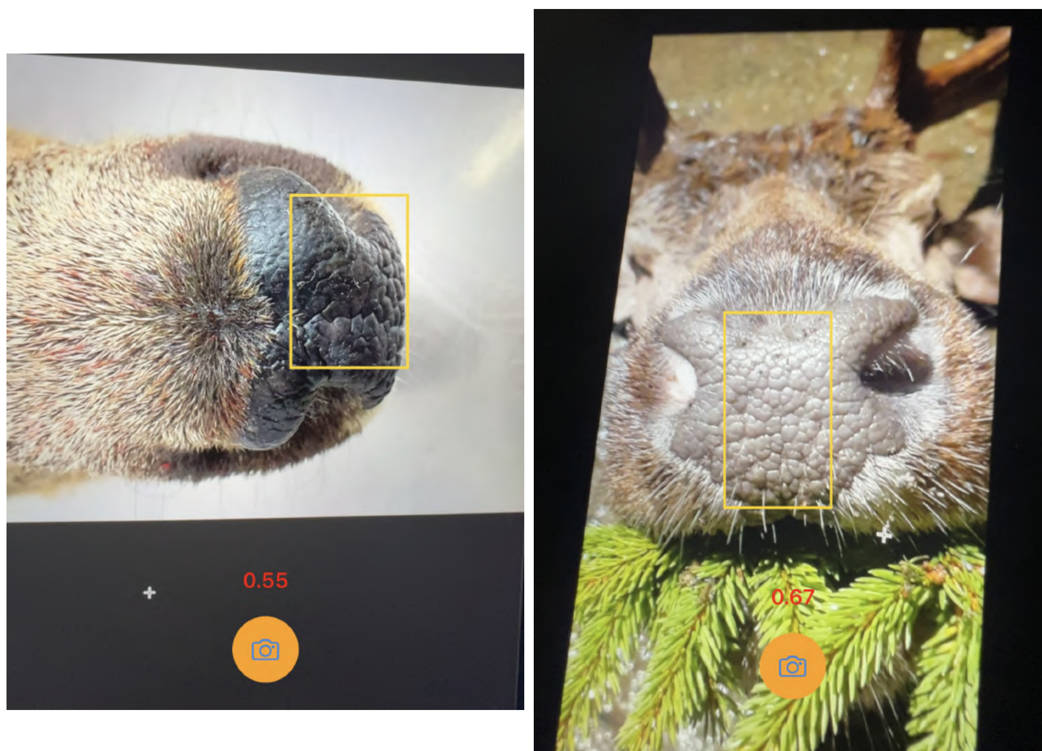
V poslednej fázy testovania boli modely nasadené do aplikácie a testované simulovaním reálneho využívania aplikácie. Nakoľko sa aplikácia má využívať na snímanie zabitej zveri bolo toto simulovanie obmedzené na detekovanie nosa na snímkoch zobrazených na displeji iného zariadenia.

Vyhodnotenie kvality fotografie

Testovanie vyhodnocovania kvality fotografie v aplikácii prebiehalo spoločne s testovaním integrovaných modelov neurónovej siete. Počas testovania niekoľkých modelov v aplikácii bolo zistené, že modely dokážu rozpoznávať s vysokým indexom dôveryhodnosti len nosy, ktoré majú dostačujúcu kvalitu. Tento efekt bol zapríčinený faktom, že modely boli trénované na datasete obsahujúcom len kvalitné fotografie nosa. Preto v prípade, že nos je na fotografii rozmazaný alebo je jeho kvalita inak znehodnotená, model nos nenachádza alebo ho nachádza s malou hodnotou dôveryhodnosti a na základe toho je toto rozpoznanie ig-

[1] Outliner je označenie pre odľahlé dáta. Dáta ktoré sú mimo bežného rozsahu dát použitých v datasete.

norované aplikáciou. Hodnota dôveryhodnosti modelu v rozpoznanú oblasť musí byť väčšia ako *0.5* z celkového rozsahu *0 - 1*.



Obr. 8.1: Snímky obrazovky aplikácie, vytvorené počas testovania detekcie nosa

Kapitola 9

Záver

Cieľom diplomovej práce bolo vytvoriť mobilnú aplikáciu na snímanie fotografií zabitej zveri za účelom nasledujúcej identifikácie danej zveri na serveri. Aplikácia užívateľa navádza pomocou užívateľského rozhrania na vytvorenie kvalitnejšej fotografie nosa zvieraťa, tento jav by mal zabezpečiť väčšiu pravdepodobnosť správnej identifikácie zveri na vzdialenom serveri a zároveň zamedzenie nahrávania fotografií na server, na ktorých nie je potrebná oblasť určená na identifikáciu. Okrem vytvorenia fotografie a priradenia jej k určitému zvieraťu, aplikácia umožňuje prehľad všetkých zabitých zvierat užívateľom a tak vytvára históriu daného poľovníka.

Diplomová práca sa okrem aplikácie zaoberala vytvorením vzdialeného servera, ktorý je využívaný na perzistentné ukladanie a evidovanie zabitej zveri od všetkých užívateľov. Server podporuje vytváranie nových užívateľov a ich verifikáciu, tou zabezpečuje oddelenie vytvorených záznamov zvierat. Na zabezpečenie verifikácie bola využitá technológia *JWT token*.

V mobilnej aplikácii sa využíva aj model umelej inteligencie, ktorý bol poslednou súčasťou diplomovej práce. Nakoľko neexistuje žiadny voľne prístupný model umelej inteligencie, ktorý by slúžil na nájdenie nosa zvieraťa vo fotografii, bolo najskôr potrebné overiť hypotézu, že nos zvierat je dostatočný identifikátor na rozlíšenie jedincov od seba. Touto problematikou sa zaoberajú viaceré práce, ktoré potvrdzujú, že vzory na nosoch zvierat sú od seba na toľko odlišné, že je možné nos použiť ako identifikátor jedinca. Po potvrdení tejto hypotézy sa natrénovali viaceré modely neurónových sietí za účelom rozpoznania nosa na fotografii. Modely boli trénované na datasete fotografií poskytnutých od viacerých poľovníkov.

Dataset bol upravený tak, aby obsahoval len fotografie vysokej zveri vyskytujúcej sa v českých lesoch. Na základe tohto faktu sú modely neurónovej siete natrénované len na rozpoznávanie nosa vysokej zveri. Úprava datasetu bola potrebná nakoľko fotografie iných druhov zvierat boli v nedostatočnom počte.

Na vytvorenie modelov boli zvolené dva prístupy. Prvým prístupom bolo dotrénovávanie modelov, ktoré sú využívané na detekciu rôznych objektov v obraze. Druhým prístupom bolo vytvorenie úplne nových modelov, ktoré budú od začiatku trénované za účelom detekcie nosa zvierat vo fotografii. Výsledný model využívaný v aplikácii je model vytvorený prvým prístupom, nakoľko výsledky z tohto prístupu boli lepšie.

Vytvorená aplikácia spĺňa všetky funkcie, ktoré boli pôvodne naplánované a tak sa v budúcnosti naskytuje možnosť aplikáciu rozšíriť o nové možnosti. Jedným z takýchto rozšírení, môže byť podpora nie len fotografií za účelom identifikácie ale za účelom prezentácie samotného užívateľa. Užívateľ by mohol vyfotografovať trofejnú zver ako celé zviera nie len

jej časť určenú na identifikáciu a zdielať tento úlovok v aplikácií medzi svojich kamarátov a do rôznych skupín. Toto rozšírenie je už z časti implementované na serveri, každá vytvorená inštancia zvieraťa má okrem referencie na svojho majiteľa aj možnosť pridania referencií na iných užívateľov.

Ďalším vylepšením, ktoré by zlepšilo aj pôvodnú zamýšľanú funkcionalitu aplikácie je vytvorenie nového modelu neurónovej siete, tento model by bol trénovaný na uložených dátach vytvorených práve za pomoci aplikácie. Týmto by sa zabezpečilo zvýšenie kvality a presnosti rozpoznávania nosa počas snímania fotografie, nový model by mal dokazovať lepšie výsledky kvôli rozšírenému datasetu.

Posledným vylepšením môže byť natrénovanie modelov neurónových sietí aj na iné druhy zvierat. V prípade ak by tieto modely mali dobrú úspešnosť detekcie nosa vo fotografii, ich začlenenie do aplikácie by bolo jednoduché. Implementovaná relácia snímania poskytuje jednoduché začlenenie modelu, dokonca by nebola ani potrebná užívateľova interakcia na zvolenia toho, ktorý model má byť práve využívaný.

Literatúra

[1] *About Keras* [online]. [cit. 2023-1-2]. Dostupné z: https://keras.io/about/.

[2] *Cacheable* [online]. [cit. 2022-11-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/cacheable.

[3] *Core ML Tools Overview* [online]. [cit. 2023-1-2]. Dostupné z: https://coremltools.readme.io/docs.

[4] *HTTP request methods* [online]. [cit. 2022-11-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[5] *Idempotent* [online]. [cit. 2022-11-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Idempotent.

[6] *Introduction - JSON Web Tokens*. Dostupné z: https://jwt.io/introduction.

[7] *JSON - JavaScript Object Notation*. Dostupné z: https://www.json.org/json-en.html.

[8] *Model–view–viewmodel* [online]. [cit. 2022-10-12]. Dostupné z: https://en.wikipedia.org/wiki/ModelâĂŞviewâĂŞviewmodel.

[9] *Natural Language Processing*. Dostupné z: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803100529166.

[10] *REST API: Key Concepts, Best Practices, and Benefits* [online]. [cit. 2022-11-13]. Dostupné z: https://www.altexsoft.com/blog/rest-api-design/.

[11] *Safe (HTTP Methods)* [online]. [cit. 2022-11-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTTP.

[12] *SwiftUI* [online]. [cit. 2022-10-10]. Dostupné z: https://developer.apple.com/documentation/swiftui.

[13] *SwiftUI in Production? 6 Pros and Cons You Need to Consider* [online]. [cit. 2022-10-10]. Dostupné z: https://betterprogramming.pub/swiftui-in-production-6-pros-and-cons-you-need-to-consider-69ace40a1b46.

[14] *Web API* [online]. [cit. 2022-11-12]. Dostupné z: https://en.wikipedia.org/wiki/Web_API.

[15] *Welcome to Flask-RESTX's documentation!* [online]. [cit. 2022-11-12]. Dostupné z: https://flask-restx.readthedocs.io/en/latest/.

[16] *Xcode* [online]. [cit. 2022-10-10]. Dostupné z: https://en.wikipedia.org/wiki/Xcode.

[17] Alamdari, M. *How to do Object Recognition with TensorFlow(Keras) the Easiest way* [online]. [cit. 2022-12-30]. Dostupné z: https://medium.com/@mralamdari/imagehow-to-do-object-recognition-with-tensorflow-keras-the-easiest-way-23c7ab9604c7.

[18] AppleInsider. *Inside Apple's new Swift programming language*. Dostupné z: https://appleinsider.com/inside/swift.

[19] Choi, H. I., Kim, M.-Y., Yoon, H.-Y., Lee, S., Choi, S. S. et al. Study on the Viability of Canine Nose Pattern as a Unique Biometric Marker. *Animals*. 2021, zv. 11, č. 12. DOI: 10.3390/ani11123372. ISSN 2076-2615. Dostupné z: https://www.mdpi.com/2076-2615/11/12/3372.

[20] Codecademy. *What is REST?* Dostupné z: https://www.codecademy.com/article/what-is-rest.

[21] Concilio, I. *Image Processing and ComputerVision for iOS Applications: An Introduction* [online]. [cit. 2022-10-13]. Dostupné z: https://medium.com/academy-eldoradocps/image-processing-and-computervision-for-ios-applications-an-introduction-fe3171e10be7.

[22] DigitalOcean. *What is a Server?* Dostupné z: https://www.digitalocean.com/community/tutorials/what-is-a-server.

[23] Dixon, B. E., Broyles, D., Crichton, R., Biondich, P. G. a Grannis, S. J. Chapter 8 - Architectures and approaches to manage the evolving health information infrastructure. In: Dixon, B. E., ed. *Health Information Exchange (Second Edition)*. Second Edition. Academic Press, 2023, s. 199–215. DOI: https://doi.org/10.1016/B978-0-323-90802-3.00001-0. ISBN 978-0-323-90802-3. Dostupné z: https://www.sciencedirect.com/science/article/pii/B9780323908023000010.

[24] IBM. *Neural Networks*. Dostupné z: https://www.ibm.com/topics/neural-networks.

[25] Inc, . A. *Create ML* [online]. [cit. 2023-1-19]. Dostupné z: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_keychain.

[26] Inc, . A. *Create ML* [online]. [cit. 2023-1-2]. Dostupné z: https://developer.apple.com/documentation/createml.

[27] Inc., A. *Model-View-Controller* [online]. [cit. 2022-10-12]. Dostupné z: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

[28] Inc., A. *Package Manager* [online]. [cit. 2022-10-12]. Dostupné z: https://swift.org/package-manager/.

[29] Inc., A. *UIKit* [online]. [cit. 2022-10-10]. Dostupné z: https://developer.apple.com/documentation/uikit/.

[30] Inc., A. *UserDefaults* [online]. [cit. 2022-10-11]. Dostupné z: https://developer.apple.com/documentation/foundation/userdefaults.

[31] Inc., A. *What Is Core Data?* [online]. [cit. 2022-10-11]. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/index.html#//apple_ref/doc/uid/TP40001075-CH2-SW1.

[32] Mallick, S. *Histogram of Oriented Gradients explained using OpenCV* [online]. [cit. 2023-1-2]. Dostupné z: https://learnopencv.com/histogram-of-oriented-gradients/.

[33] Murakami, T., Uraguchi, K. a Abe, G. Individual Identification of Mesocarnivores Using Photos of Noseprint Patterns. *Mammal Study*. Mammal Society of Japan. 2016, zv. 41, č. 1, s. 9 – 15. DOI: 10.3106/041.041.0103. Dostupné z: https://doi.org/10.3106/041.041.0103.

[34] Ognjanovski, G. *Everything you need to know about Neural Networks and Backpropagation — Machine Learning Easy and Fun* [online]. [cit. 2022-12-30]. Dostupné z: https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

[35] Pehtrus, B. *MVVM+Coordinators IOS Architecture Tutorial* [online]. [cit. 2022-10-12]. Dostupné z: https://medium.com/nerd-for-tech/mvvm-coordinators-ios-architecture-tutorial-fb27eaa36470.

[36] Petersen, W. The Identification of the Bovine by Means of Nose-Prints1. *Journal of Dairy Science*. 1922, zv. 5, č. 3, s. 249–258. DOI: https://doi.org/10.3168/jds.S0022-0302(22)94150-5. ISSN 0022-0302. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0022030222941505.

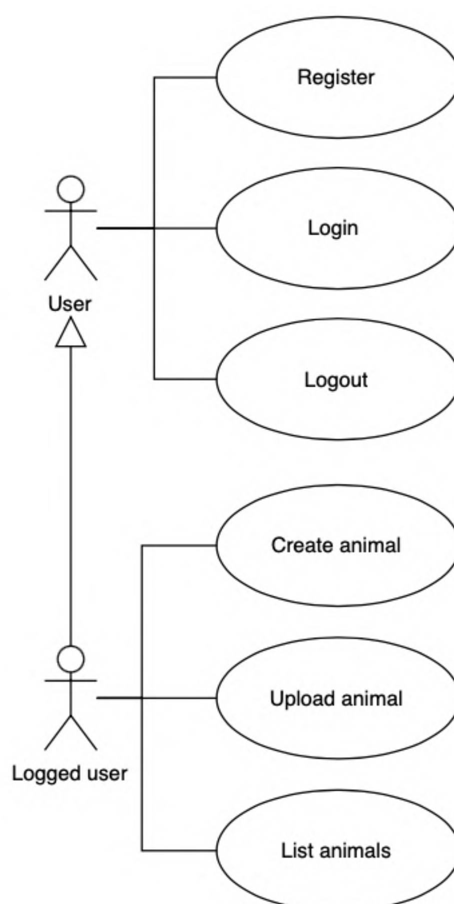
[37] Shivang. *What is Swagger? And Why Do You Need it for your Project?* [online]. [cit. 2022-11-12]. Dostupné z: https://scaleyourapp.com/what-is-swagger-and-why-do-you-need-it-for-your-project/.

[38] Tyagi, M. *HOG (Histogram of Oriented Gradients): An Overview* [online]. [cit. 2023-1-2]. Dostupné z: https://towardsdatascience.com/hog-histogram-of-oriented-gradients-67ecd887675f.

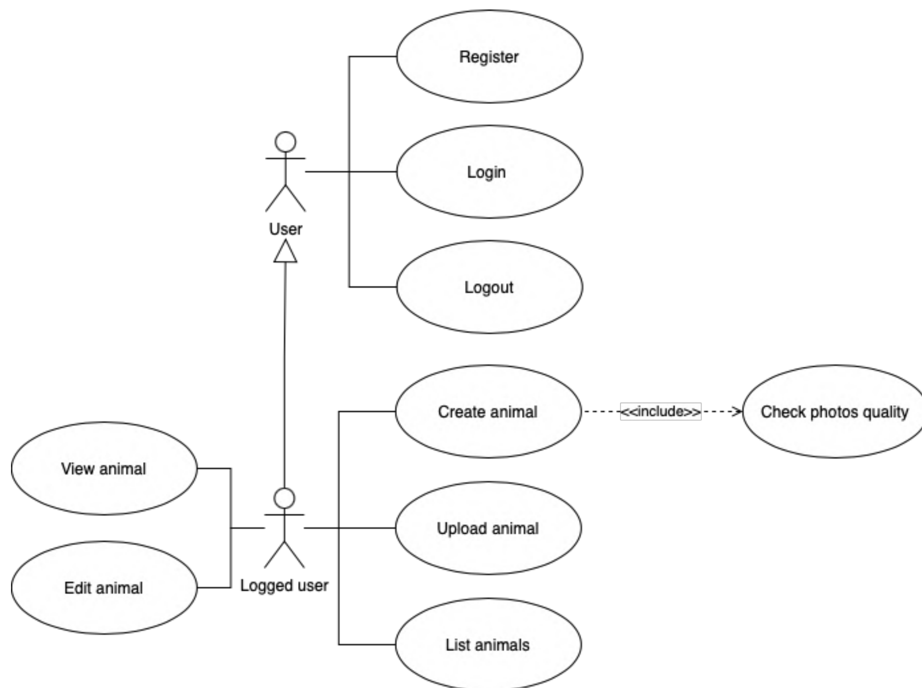
[39] Valdés, D. L. *IOS Architecture: MVVM-C, Coordinators* [online]. [cit. 2022-10-12]. Dostupné z: https://medium.com/sudo-by-icalia-labs/ios-architecture-mvvm-c-coordinators-3-6-3960ad9a6d85.

Príloha A

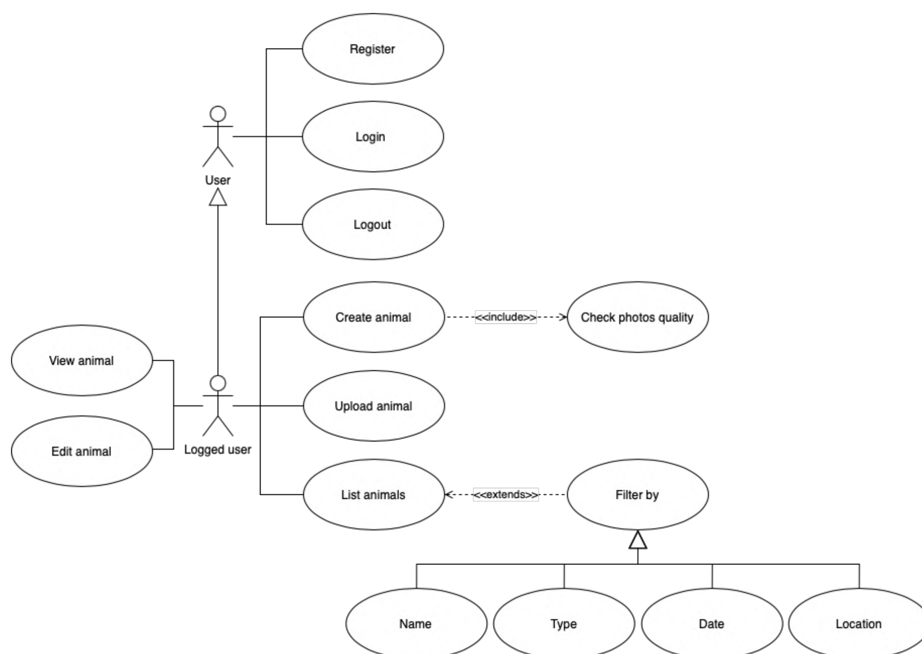
Diagramy návrhov iterácií



Obr. A.1: Diagram použitia prvej iterácie aplikácie



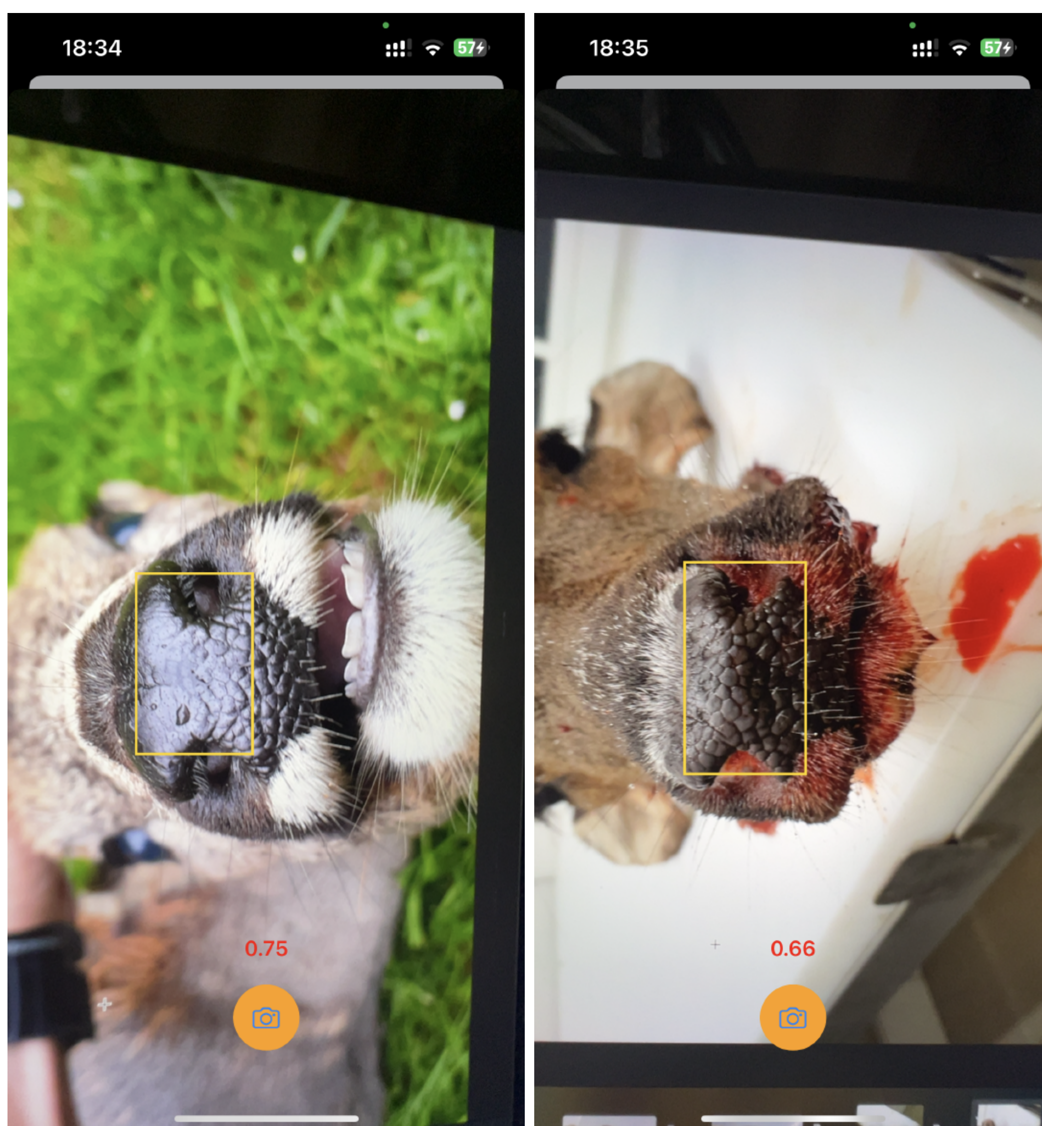
Obr. A.2: Diagram použitia druhej iterácie aplikácie



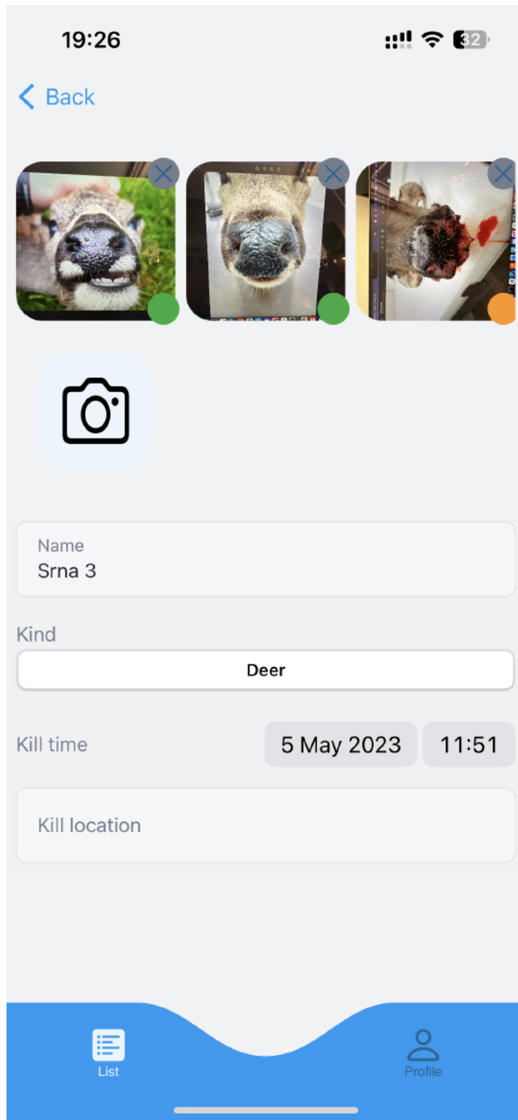
Obr. A.3: Diagram použitia tretej iterácie aplikácie

Príloha B

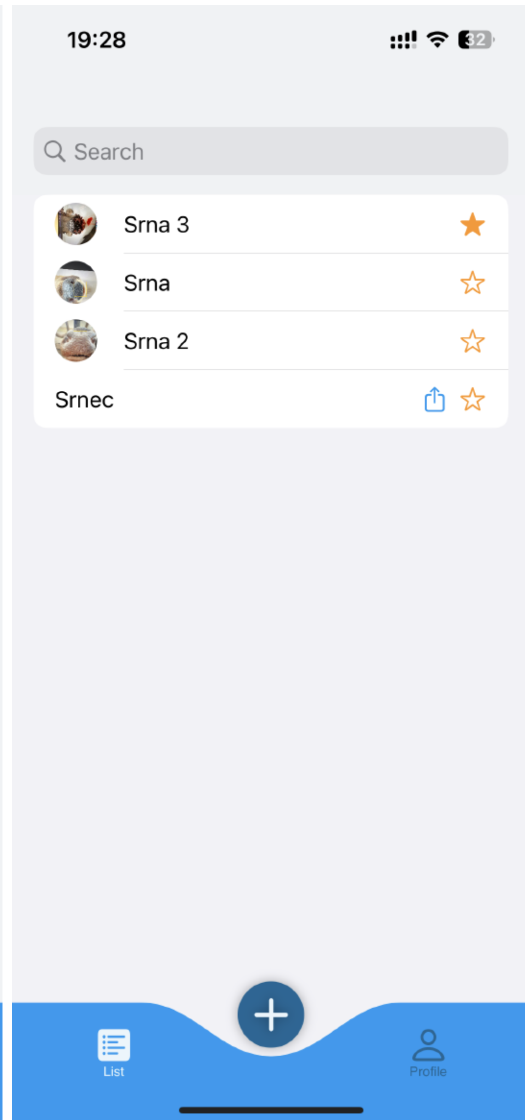
Snímky obrazovky



Obr. B.1: Zachytávanie fotografie nosov zvierat



Obr. B.2: Obrazovka znázorňujúca detail zvieraťa



Obr. B.3: Obrazovka znázorňujúca list všetkých zvierat