



**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ  
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

## **ДИПЛОМНА РАБОТА**

по професия код 481020 „Системен програмист“  
специалност код 4810201 „Системно програмиране“

Тема: Мобилно приложение за мониторинг на защитени видове

Дипломант:

*Ивайло Ивайлов Георгиев*

Дипломен ръководител:

*Ивайло Абаджиев*

СОФИЯ

2023



**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ  
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 22.11.2022 г.  
Дата на предаване: 22.02.2023 г.

Утвърждавам:.....  
/проф. д-р инж. П. Якимов/

## **ЗАДАНИЕ за дипломна работа**

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ  
по професия код 481020 „Системен програмист“  
специалност код 4810201 „Системно програмиране“

на ученика Ивайло Ивайлов Георгиев от 12 Б клас

1. Тема: Мобилно приложение за мониторинг на защитени видове
2. Изисквания:
  - Вход и регистрация с потребителско име и парола
  - Достъп и управление на потребителски профил
  - Достъп и управление на списък с наблюдения
  - Добавяне и редактиране на наблюдение
  - Добавяне и управление на снимки в наблюдението
  - Филтриране и търсене на наблюдения
  - Карта на наблюдения с включен режим “покажи най-близките до мен”
3. Съдържание
  - 3.1 Теоретична част
  - 3.2 Практическа част
  - 3.3 Приложение

Дипломант :.....  
/ Ивайло Георгиев /

Ръководител:.....  
/ Ивайло Абаджиев /

Директор:.....  
/ доц. д-р инж. Ст. Стефанова /

# Становище на дипломен ръководител

Дипломантът е представил завършен дипломен проект. Въпреки готовия проект той не покрива напълно функционалните изисквания спрямо заданието за дипломна работа.

Трябва да се отбележи, че дипломната работа включва усвояване на нова платформа - Android и нов език за програмиране - Kotlin. Въпреки това дипломантът е съумял да се вмести в краткия срок и да разработи приложение, използвайки съвременните практики и конвенции за разработка на мобилен софтуер.

Препоръчвам дипломантът да бъде допуснат до защита.

Ръководител:.....

/ Ивайло Абаджиев/

# Увод

С разрастването на населените места и индустриалните зони, все повече естествени хабитати биват експлоатирани за природни ресурси, строителни и земеделски пространства. От производството на синтетични материали и енергия от полезни изкопаеми остават много неразградими отпадъци, които рано или късно се озовават в природата. Това причинява замърсяване на водите, почвите и въздуха, което освен за нас, се оказва пагубно и за много от другите организми, с които споделяме нашата планета. В много държави се поставят вятърни турбини, които въпреки че предоставят зелена енергия, се явяват фатално препятствие за много мигриращи видове птици. Голям процент и от по-старите водноелектрически централи не предоставят обезопасени канали, по които водните обитатели да се придвижват по течението. Това пречи на стотици видове риба да се размножават ефективно, което от своя страна ограничава количеството храна за всички рибоядни животни.

От началото на Първата индустриална революция (1750 г.) са изчезнали над 560 животински и растителни вида, а в момента хиляди популации са на ръба на изчезването. За да се запазят застрашените видове, правителства и организации от цял свят изграждат резервати, спасителни центрове, съставят планове за развъждане в плен и реинтродукция в дивото. Ефективността на тези усилия може да се проверява основно с проекти за екологичен мониторинг, в които изследователи и специалисти периодично следят броя и състоянието на отделните популации. Намирането на достатъчен брой кадри и финансиране за множеството програми е непосилно за голяма част от организациите, което води до появата на така наречените “природонаучни платформи”. Това най-често представляват приложения, които позволяват на природолюбители и хора без биологично образование да споделят с научните среди важна информация за свои наблюдения на определени видове.

В последни години, тази практика се е установила в много държави и е доказала своя положителен ефект. През 2014 г. в България се създава подобна система, която вече няколко години помага за опазването на множество видове грабливи и водолюбиви птици, бозайници, влечуги и насекоми.

Целта на настоящата дипломна работа е да предостави решение на проблема с липсата на достатъчно кадри за провеждане на биологичен мониторинг, чрез мобилно приложение, което да позволи на природолюбители и активисти да съставят и споделят списъци с наблюдения на различни видове. Чрез тези наблюдения, хората ще имат възможност да научат повече за околната среда, както и да намерят повече мотивация да я опазват и изследват. Така приложението ще има пряк образователен ефект върху неговите потребители, защото ще им позволи да видят къде и в какъв период от годината се срещат различните видове, както и да видят техни снимки.

Освен за природозащитни цели, събраните данни ще са полезни и при създаването на научни трудове и публикации. Учените ще могат да виждат кога и къде са наблюдавани техните обекти на изучаване и да включват необходимата информация в собствени проекти.

# Първа глава

## Преглед на подобни приложения и съществуващи технологии за реализация на мобилни приложения

### 1.1. Преглед на подобни приложения

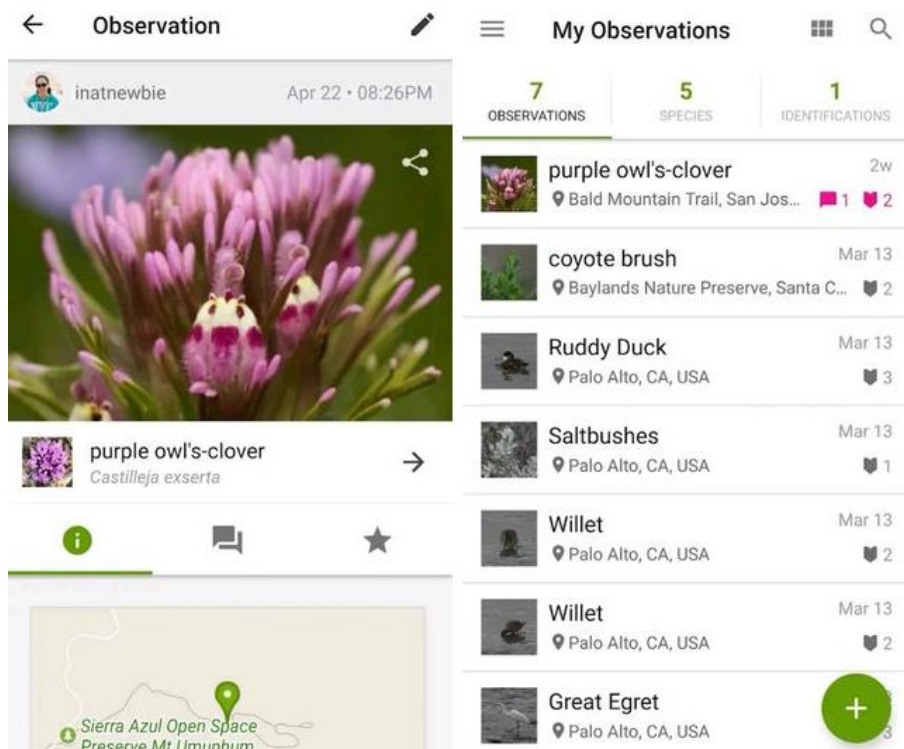
В последните 10 години на пазара се появяват множество платформи и приложения, целящи да следят популациите и разпространението на различни животински и растителни видове. Повечето вече имат хиляди редовни потребители и публични общности, които не спират да се разрастват. Интересът към опазването на природата определено се увеличава, което дава поле за развитие на проекти като тази дипломна работа.

Сега следва да разгледаме три от по-популярните мобилни приложения за биологичен мониторинг.

#### 1.1.1. iNaturalist

Едно от първите мобилни приложения в тази сфера, пуснато през 2011 година. Налично е за Android и iOS.

В приложението има така наречените “наблюдения”, които задължително трябва да съдържат снимка на организъм, дата и локация (*Фигура 1.1*). Едно наблюдение може да съдържа само един биологичен вид. Ако потребителят е способен сам да определи организма, може да избере неговото наименование от списък. Всяко наблюдение може да получи таг “Изследователско качество”, ако 3 или повече души се съгласят, че видът на снимката е определен правилно. Това цели да покаже, че въведената информация е възможно най-достоверна. Наблюденията могат да се търсят по видове и локации.



Фиг. 1.1 – Два екрана от iNaturalist приложението

### 1.1.2. SmartBirds Pro

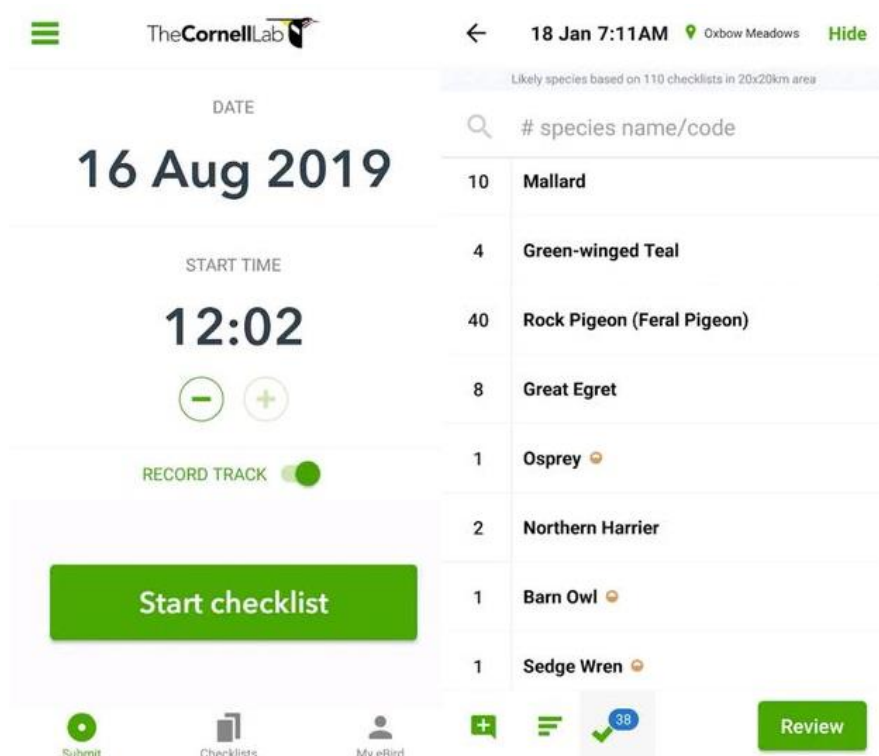
Първото българско приложение за мониторинг на биологични видове, пуснато през 2014 г. от БДЗП. Налично само за Android.

Информацията за проведения мониторинг се въвежда във формуляри, като потребителят може да избере от няколко различни според типа организми, които наблюдава, или даден проект, в който участва. Трябва да въведе дата, час, локация, климатични условия, видове и детайли за състоянието на видените екземпляри. Могат да се вписват всички растения, безгръбначни и гръбначни животни, които са част от Закон за биологичното разнообразие (ЗБР).

### 1.1.3. eBird by Cornell Lab

Мобилното приложение е пуснато през 2015. Налично е за операционните системи Android и iOS.

В него фигурират така наречените “списъци” (Фигура 1.2). Всеки списък трябва да съдържа дата, час, локация, колко време потребителят е бил на терен и поне един въведен вид. Опционално – могат да се добавят колко километра са били провървяни по време на мониторинга и да се добавят някакви коментари за района и наблюдаваните птици. Може да се провери какви видове са наблюдавани в даден район при включване на GPS.



Фиг. 1.2 – Снимки на двата от екраните на приложението eBird by Cornell Lab

## 1.2. Съществуващи технологии и развойни среди за реализиране на мобилно приложение

### 1.2.1. Мобилни операционни системи

Мобилните операционни системи<sup>[8]</sup> са софтуерни платформи, които се използват в мобилни устройства, като телефони и таблети. Те осигуряват основните функционалности необходими за управление на устройството и позволяват на потребителя да инсталира и използва приложения.



### **1.2.1.1. Android**

Това е една от най-популярните мобилни операционни системи. Android <sup>[10]</sup> е разработен от Google и има код с отворен достъп (open-source)<sup>[1]</sup>, така че различни производители могат да използват и модифицират платформите си. Това е причина за широката му дистрибуция и повече от 80% от мобилните устройства на световния пазар оперират върху тази система.

### **1.2.1.2. iOS**

Също много разпространена операционна система, разработена от Apple. Има затворен код<sup>[2]</sup>, което означава, че може да се използва само върху устройства на Apple и често няма възможност за модификация от страна на други производители. В резултат на това, iOS е използван главно върху iPhone и iPad устройства и има по-ограничена дистрибуция в сравнение с Android.

### **1.2.1.3. Други**

Други мобилни операционни системи като Windows Phone и BlackBerry OS също са налични, но имат по-малка популярност и достъпност на пазара. Те обикновено се използват в бизнес среда и имат различни допълнителни функционалности, като защита на данни и корпоративна интеграция.

## **1.2.2. Развойни среди за разработка на мобилни приложения**

Развойните среди (Integrated Development Environment – IDE) или още наричани интегрирани среди за разработка<sup>[9]</sup>, са софтуерни приложения с комплекс от инструменти и технологии, които се използват за създаване и интеграция на код и ресурси, като предлагат инструменти за дебъгване и профилиране в единна среда. Подобряват процеса на разработка като го правят по-ефективен.

### **1.2.2.1. Android Studio**

Android Studio е безплатна развойна среда с открит код за разработка на мобилни приложения на Android. Тя е официално поддържана от Google и се използва от множество разработчици и компании по света. Изградена е върху софтуера на JetBrains и IntelliJ IDEA. Един от нейните плюсове е функцията за задължително автоматично запамятаване (auto-save) на кода, която гарантира, че няма да има загуба на прогрес. Характеристики:

- За компилация (compilation support) се използва Gradle
- За интеграция на приложението: ProGuard
- Вградена поддръжка на облачни услуги, което позволява интеграцията на Firebase Cloud Messaging, Google App Engine и други
- Използване на емулятори за стартиране и дебъгване на програмата

### **1.2.2.2. IntelliJ IDEA**

IntelliJ IDEA е развойна среда предназначена за програмиране на Java и други обектно ориентирани езици. Предлага богат избор от инструменти, включително автоматично довършване на код, интегриране със системи за контрол на версиите, инструменти за отстраняване на грешки и тестване на кода. IntelliJ има интуитивен потребителски интерфейс и е добре оптимизиран за голям брой проекти. Също така предоставя поддръжка за изграждане на приложения в множество платформи, включително за Android.

### **1.2.2.3. Xcode**

Xcode е интегрирана среда за разработка на софтуер, създадена за разработване на проекти в Apple платформи като macOS, iOS, iPadOS и watchOS. Позволява на разработчиците да създават, тестват и споделят приложения за различни Apple устройства и включва инструменти за управление на проекти, кодиране, отстраняване на грешки и тестване на приложения. Xcode има богата библиотека от инструменти, API<sup>[3]</sup> и подобно на AndroidStudio има емулятор. С

помощта на средата, програмистите могат да оптимизират своите приложения за по-добро представяне, да ги тестват на различни устройства и да ги публикуват в App Store.

#### **1.2.2.4. Visual Studio Code**

Visual Studio Code е програмен редактор създадена от Microsoft. Наличен е за различни операционни системи като Windows, Linux и macOS. Предлага набор от различни функционалности и поддържа множество езици за програмиране, включително HTML, CSS, Java, JavaScript, Python, Ruby и много други. Редакторът е лек и оптимизиран, което му дава предимство пред други подобни софтуерни приложения.

#### **1.2.3. Kotlin**

Kotlin<sup>[12]</sup> е официален език за разработка на Android приложения и се поддържа от Google. Това гарантира, че винаги има актуални инструменти и документация за работа с него, както и достъп до най-новите функционалности и библиотеки за разработка на приложения. Езикът има компактен и четим синтаксис, което прави проектите с него по-лесни за разбиране и поддръжка. Предлага много удобни функционалности като “null safety”, което помага да се предотвратят много от честите грешки, свързани с работа с нулеви стойности.

Kotlin е пълноценен “заместител” на Java и се компилира до Java байткод, което означава, че много от наличните Java библиотеки могат да бъдат използвани в него.

Към езика са имплементирани много вградени функционалности, като например корутини (coroutines), които предоставят по-лесен и ефективен начин за управление на асинхронен код.

#### **1.2.4. Java**

Java е обектно ориентиран език за програмиране, който използва компилация до байткод и виртуална машина (JVM), за да постигне преносимост

на програмите между различни платформи. Това гарантира лесно разработване на приложения за различни операционни системи и архитектури. Сигурността на езика е висока, тъй като не се работи директно с паметта. Java е един от основните езици за разработване на Android приложения и се използва от популярни платформи като Spotify, Twitter и други.

### **1.2.5. Firebase Cloud Firestore**

Firestore е облачна услуга за мащабируемо съхранение на данни, която позволява бърз и лесен достъп в реално време. Използва нерелационен (NoSQL) модел и предоставя API, което улеснява манипулацията на информация от множество клиентски устройства и платформи. Проектирана е да работи добре с мобилни и уеб приложения, като има и поддръжка за автоматичната репликация на данни в различни региони, което подобрява надеждността и скоростта на достъпа. Firestore също предоставя гъвкавост при работа с права за достъп и роли на потребителите. Може да бъде интегрирана лесно в множество среди за разработка на мобилни приложения, включително в Android Studio и Xcode.

# Втора глава

## Проектиране на структурата на мобилното приложение

### 2.1. Функционални изисквания на мобилното приложение

- Вход и регистрация
- Създаване на профил чрез потребителско име/имейл и парола
  
- Достъп и управление на потребителски профил
- Влизане в приложението със създаден профил
- Промяна на паролата
  
- Достъп и управление на списък с наблюдения
- Достъп до списък с наблюдения, които да могат да се отварят и разглеждат
  
- Добавяне и редактиране на наблюдение
- Добавяне на ново наблюдение, което включва: дата, час, продължителност на мониторинга, име на локацията, коментар (например за хабитата или климата), брой видове и списък с наблюдаваните организми
- Всеки потребител да може да редактира наблюденията, които е въвел
  
- Добавяне и управление на снимки в наблюдението
- Към всяко наблюдение да може да се добавя снимка от галерията
- Да може после снимката да се редактира и смени
  
- Филтриране/търсене на наблюдения
- Да има налична опция за филтриране, например за наблюдения с най-много записани видове

- Карта на всички наблюдения
- Карта, която показва положението на потребителя и наблюденията около него

## **2.2. Избор на технологии за реализацията на мобилното приложение**

### **2.2.1. Избрана операционна система – Android**

Android<sup>[11]</sup> има много предимства пред останалите мобилни операционни системи като например:

- Той е най-разпространената операционна система за мобилни устройства в света. Според статистиките за пазара на мобилни операционни системи, към края на 2022 г. пазарният дял на Android в световен мащаб е над 71%, което означава, че приложения разработени с него ще бъдат достъпни за много потребители.
- Android предоставя голяма гъвкавост при разработката на приложения, като поддържа множество езици за програмиране, включително Java, Kotlin и C++. Това дава по-голям избор от технологии за реализацията на проекта.
- Android предлага богат API, който дава възможност за интеграция с множество функционалности и услуги на устройството, включително камерата, GPS, контактите и други.
- Работи на различни устройства, като телефони, таблети и други, което означава, че е възможно да се създават приложения за много различни целеви групи от потребители.

### **2.2.2. Избрана среда за разработка – Android Studio**

Android Studio е лесен за инсталиране и конфигуриране, като поставя всички инструменти, необходими за разработване на приложението, на едно

място. Заради това се нарежда сред най-използваните интегрирани среди за разработка с Android. Други причини, които го правят препоръчителен, са:

- Поддържа множество езици за програмиране, включително Java и Kotlin, което го прави подходящ за широк кръг от разработчици.
- Има богата документация, която е леснодостъпна и предоставя всички необходими ресурси за разработка на мобилни приложения.
- Android Studio е активно развиващ се инструмент и редовно има актуализации с нови функционалности, които го правят все по-добър и ефективен за разработване на мобилни проекти.
- Предоставя емулатор, който позволява на разработчиците да тестват своите приложения без да трябва да притежават набор от мобилни устройства.

### **2.2.3. Избран програмен език за разработка на приложението – Kotlin**

Kotlin е определен като официален език за разработка на Android приложения от Google и това го прави много чест избор сред програмистите. Причините, поради които той е подходящ за този проект, включват:

- Има съвместимост с Java и възможност за използване на повечето библиотеки и инструменти, създадени за езика.
- Kotlin предоставя много функции, които намаляват вероятността от грешки и неочаквано поведение на приложението, например нулева безопасност, типизация на данни и много други.
- Има по-опростен синтаксис, което улеснява изразяването на различни концепции. Това прави кода по-лесен за четене и поддръжка.
- Генерира код, който е бърз и ефективен за изпълнение при Android устройства.
- Има богата документация.

## 2.2.4. Избор на система за бази данни – Firestore

Firebase Firestore<sup>[13]</sup> е чест избор сред облачните услуги за мобилни приложения. Причините, поради които тази технология е подходяща за дипломната работа, включват:

- Има гъвкава йерархична структура на данните, което улеснява работата с тях. Информацията се съхранява в колекции от документи, а самите документи могат да поддържат различни типове данни (низове, числа и други).
- Осигурява предаване в реално време, като позволява на потребителите да виждат актуални данни от колекциите в базата.
- Firestore предоставя API, което улеснява манипулацията на данните от множество клиентски устройства и платформи, включително мобилни устройства и уеб приложения.
- Предоставя гъвкавост при работа с права за достъп и роли на потребителите, което позволява да се определят какви данни могат да бъдат достъпни от конкретни потребители или групи от потребители.

## 2.2.5. Firebase Cloud Storage

Firebase Cloud Storage<sup>[14]</sup> е облачна услуга за съхранение на файлове, която позволява бърз и лесен достъп до данните за мобилни и уеб приложения. Услугата е подходяща за качване и изтегляне на снимки в приложението. Други нейни предимства включват:

- Може да се мащабира лесно според нуждите на приложението, позволявайки му да се справя с големи обеми от данни.
- Има висока степен на защита на данните чрез автоматичното им шифриране в покой и по време на трансфер.
- Може да се интегрира лесно с други услуги на Firebase като Firebase Authentication, Firebase Cloud Store и други.
- Firebase Cloud Storage предоставя SDK<sup>[4]</sup> за множество езици за програмиране, включително Java и Kotlin.



## 2.2.6. Firebase Authentication

Firebase Authentication<sup>[15]</sup> предоставя backend услуга, лесни за употреба SDK и готови UI библиотеки, за да позволи на потребителя да се удостовери в дадено приложение. Може да се интегрира лесно в много различни мобилни, веб приложения и платформи. Други причини, които правят тази услуга подходяща за тази дипломна работа, включват:

- Предлага много възможности за добавяне на допълнителни функционалности като двуфакторна автентикация, синхронизация със социални мрежи и други.
- Има висока степен на защита на данните за потребителите, като може да използва много различни методи за потвърждение на идентичността (електронна поща, телефонен номер, социални мрежи и други).
- Основният план на Firebase Authentication може да се използва безплатно. Той включва базовите функционалности на системата за идентификация и управление на потребители.

## 2.2.7. Google Maps SDK за Android

Google Maps SDK<sup>[16]</sup> за Android позволява на разработчиците да добавят функционалности на Google Maps в техните Android приложения. Той включва библиотеки за карти, местоположение и създаване на маршрути, които могат да се използват за създаване на различни видове приложения, свързани с географски данни. В разработването на дипломната работа ще се използва главно, за да може потребителят да види на картата своето местоположение. Други предимства на този пакет за разработка включват:

- Лесно се интегрира в приложенията, благодарение на богатата документация, предоставена от Google.
- Предлага набор от функции за работа с карти като например създаване на интерактивни карти, добяване на маркери, търсене на местоположения и други.

## 2.3. Основни компоненти и структура на мобилното приложение

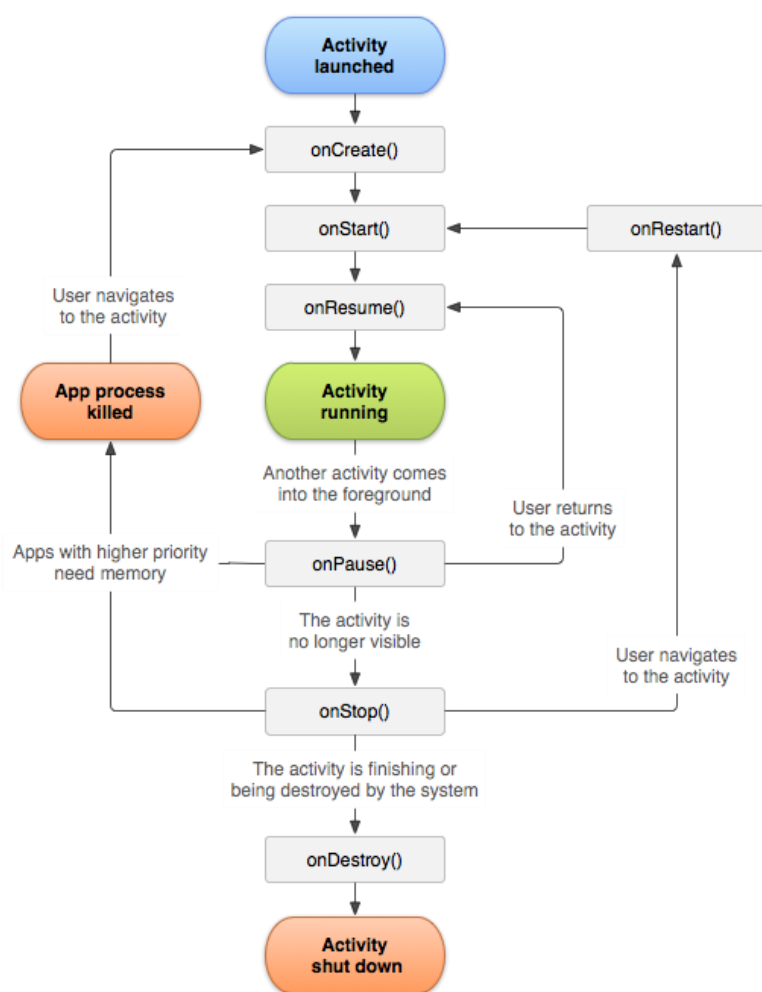
### 2.3.1. Activity

Activity е входната точка на взаимодействие между потребителя и мобилното приложение. Този основен компонент на Android приложенията често представлява екран или прозорец на потребителския интерфейс. Има жизнен цикъл (Фигура 2.1), който се състои от няколко състояния, като onCreate(), onStart(), onResume(), onPause(), onStop() и onDestroy(). Всеки един от тези методи се изпълнява при определено събитие в жизнения цикъл на даденото activity.

Едно activity може да комуникира с други компоненти на приложението чрез интененти (intents). Интенентите позволяват да се стартира друго activity и да се

получават или изпращат данни.

В него могат да се използват множество различни layout ресурси, които да съдържат различен дизайн и функционалност, за да представят екрана на потребителя. По този начин приложението може да променя дизайна на екрана динамично в зависимост от действията на потребителя.



Фиг. 2.1 – Жизнен цикъл на Activity

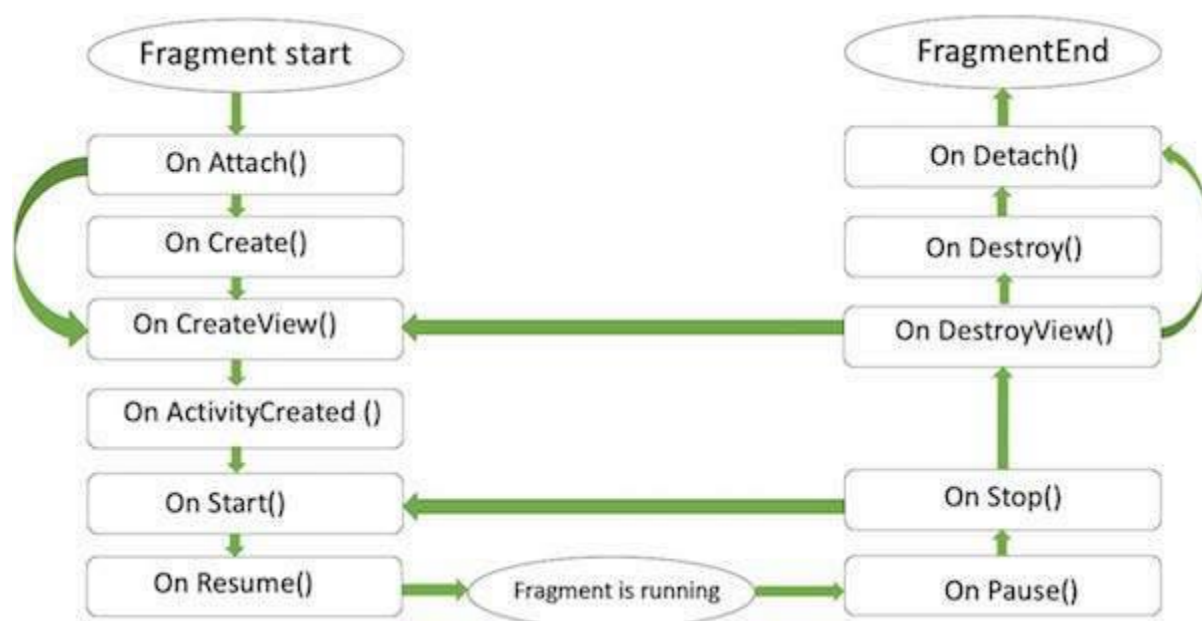
### 2.3.2. Fragment

Fragment е компонент, който позволява да се преизползва потребителският интерфейс. Това означава, че UI компонент може да бъде добавен, премахнат или заменен в екрана на потребителя без да се променя структурата на останалите елементи в този екрана.

Често фрагментите се използват за постигане на многопанелен дизайн, в който екранът на устройството е разделен на две или повече части, които могат да бъдат обработвани отделно. Всеки fragment може да управлява оформлението си самостоятелно и сам да се справя с обработването на събития. Това обаче не означава, че могат да съществуват напълно самостоятелно, и всеки fragment трябва да бъде приютен от дадено activity.

Жизненият цикъл на този вид компоненти (Фигура 2.2) се различава (но пак зависи) от цикъла на едно activity и им позволява да се справят със събития като създаване, промяна на състоянието, визуализация и деактивиране.

Основният им плюс е, че са лесни за поддръжка и разширяване. Това позволява да се създават малки и опростени фрагменти, които могат да бъдат комбинирани и реорганизирани, за да се постигне желаният потребителски интерфейс.

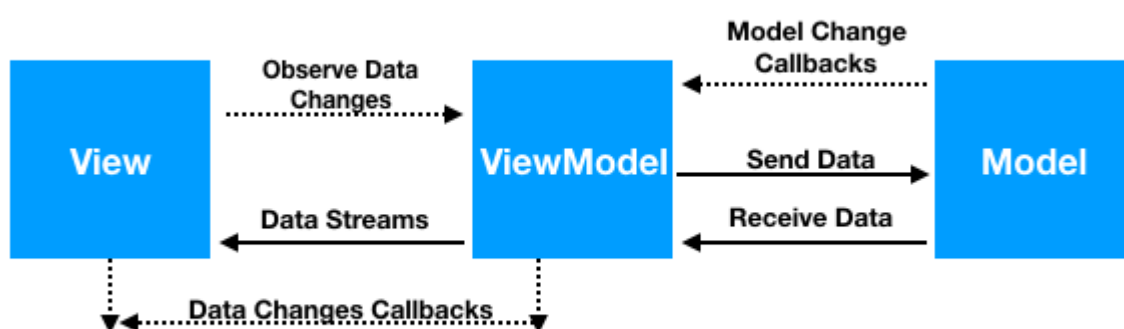


Фиг. 2.2 – Жизнен цикъл на Fragment

### 2.3.3. Model – View -ViewModel архитектура

Model-View-ViewModel (MVVM) е архитектурен модел за построяване на приложения. Цели да опрости процеса на разработка и да улесни разбирането на проектите като разделя потребителския интерфейс от бизнес логиката в три структурни елемента (*Фигура 2.3*). Това са:

- **Model** – Предоставя данните и бизнес логиката, които са необходими за приложението. Това включва взаимодействието с външни източници на данни и манипулирането на данни за обработка във ViewModel.
- **View** – Представя информацията на потребителя и приема входните му данни. Съдържа код за визуализация и управление на събитията от потребителя.
- **ViewModel** – Служи като посредник между Model и View и управлява потока на данни. Компонентът съдържа логиката за манипулиране на данните и изпращане на актуализации към View, когато данните в Model се променят. Това често се случва с помощта на LiveData данни. ViewModel също така улеснява комуникацията между View и Model, като използва Observer шаблон.



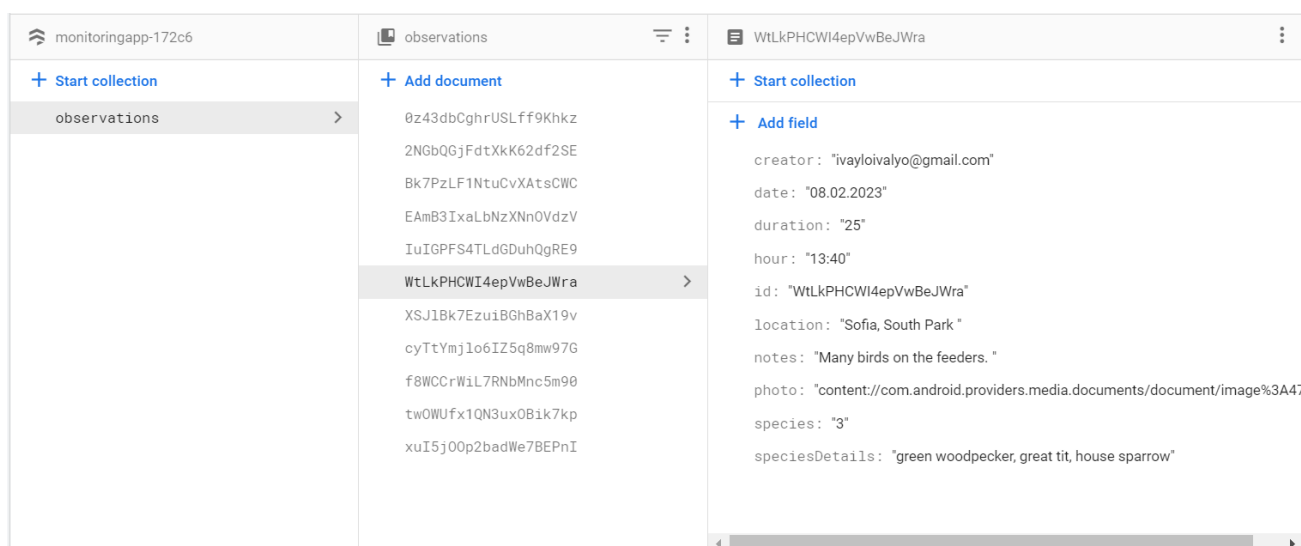
Фиг. 2.3 – Структура на MVVM

## 2.3.4. Структура на базата данни

За реализация на дипломната работа съм създал една колекция във Firestore и една папка със снимки във Firebase Storage.

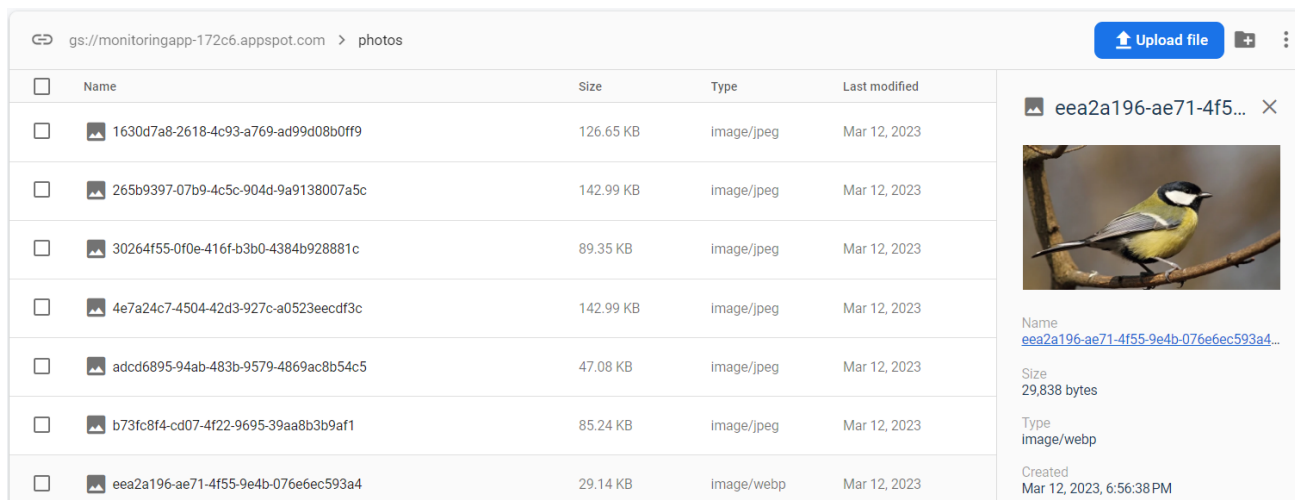
Колекцията се казва “observations” (Фигура 2.4) и представя въведените в мобилното приложение наблюдения. Полетата, които съдържа, са:








- id – уникалният идентификатор на документа
- creator – имейл на потребителя, който е въвел наблюдението
- date – датата, която е вписана при въвеждане на наблюдението
- duration – представлява колко минути потребителят е провеждал мониторинга. Вписва се ръчно.
- hour – начален час на мониторинга, който се вписва при въвеждане на наблюдението
- location – името на локацията, което се въвежда от потребителя
- notes – коментари или записки за наблюдението, които потребителят въвежда
- photo – адрес към Firebase Storage с въведеното изображение
- species – броят видове, които са били наблюдавани по време на мониторинга
- speciesDetails – описание на наблюдаваните видове





Фиг. 2.4 – Структура на базата данни във Firestore

Когато към наблюдението се добави снимка, във Firebase Storage се запазва UUID ключа на тази снимки в папка “photos” (Фигура 2.5)



<input type="checkbox"/>	Name	Size	Type	Last modified
<input type="checkbox"/>	 1630d7a8-2618-4c93-a769-ad99d08b0ff9	126.65 KB	image/jpeg	Mar 12, 2023
<input type="checkbox"/>	 265b9397-07b9-4c5c-904d-9a9138007a5c	142.99 KB	image/jpeg	Mar 12, 2023
<input type="checkbox"/>	 30264f55-0f0e-416f-b3b0-4384b928881c	89.35 KB	image/jpeg	Mar 12, 2023
<input type="checkbox"/>	 4e7a24c7-4504-42d3-927c-a0523eecd3c	142.99 KB	image/jpeg	Mar 12, 2023
<input type="checkbox"/>	 adcd6895-94ab-483b-9579-4869ac8b54c5	47.08 KB	image/jpeg	Mar 12, 2023
<input type="checkbox"/>	 b73fc8f4-cd07-4f22-9695-39aa8b3b9af1	85.24 KB	image/jpeg	Mar 12, 2023
<input type="checkbox"/>	 eea2a196-ae71-4f55-9e4b-076e6ec593a4	29.14 KB	image/webp	Mar 12, 2023

 eea2a196-ae71-4f55-9e4b-076e6ec593a4 ×



Name  
[eea2a196-ae71-4f55-9e4b-076e6ec593a4...](#)

Size  
29,838 bytes

Type  
image/webp

Created  
Mar 12, 2023, 6:56:38 PM

Фиг. 2.5 – Структура на Firebase Storage

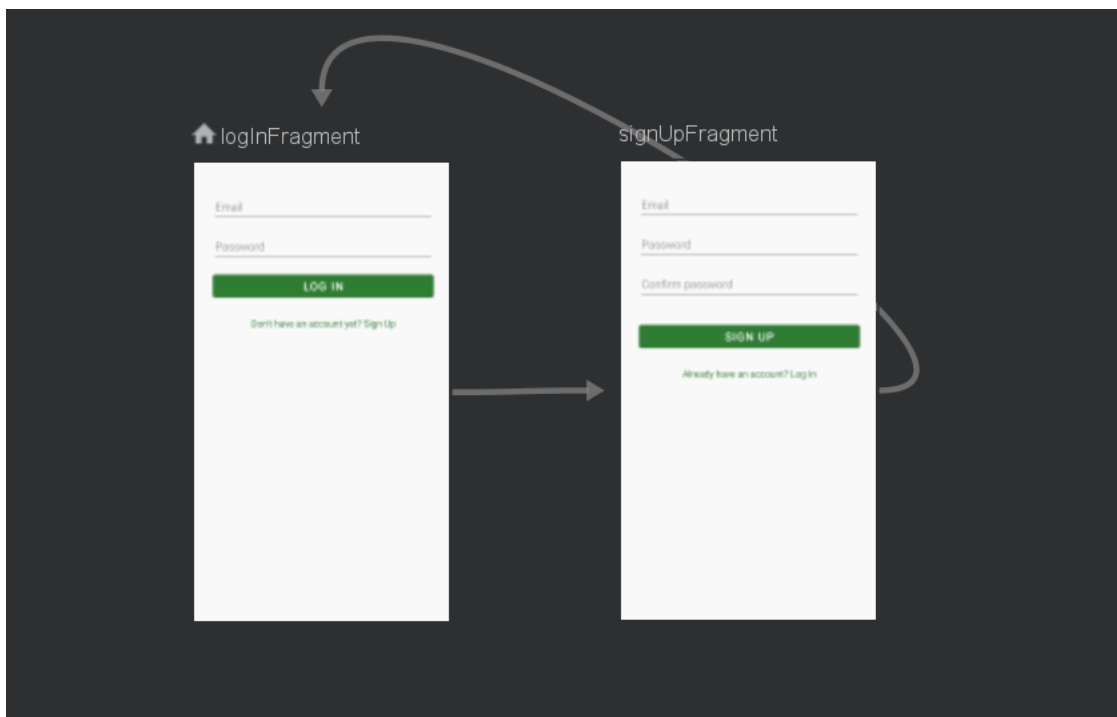
## 2.3.5. Структура на екраните

Структурата на екраните представлява съвкупност от различни фрагменти. Всеки fragment има различна роля и репрезентира отделен екран в приложението. Имат различни потребителски интерфейси и имплементирани функционалности.

### 2.3.5.1. Фрагменти в RegistrationActivity

Това activity отговаря за входа и регистрацията на потребителите. В него са разположени следните фрагменти (Фигура 2.6):

- LogInFragment – първият екран, който се появява при пускане на мобилното приложение. В него потребителите могат да въвед своя имейл и парола, за да достъпят създаден акаунт.
- SignUpFragment – потребителите, които не са се регистрирали, могат да достъпят този екран чрез бутон в LogInFragment. За да се регистрират въвеждат имейл, парола и потвърждение на паролата. След успешна регистрация пак се препращат към LogIn фрагмента.

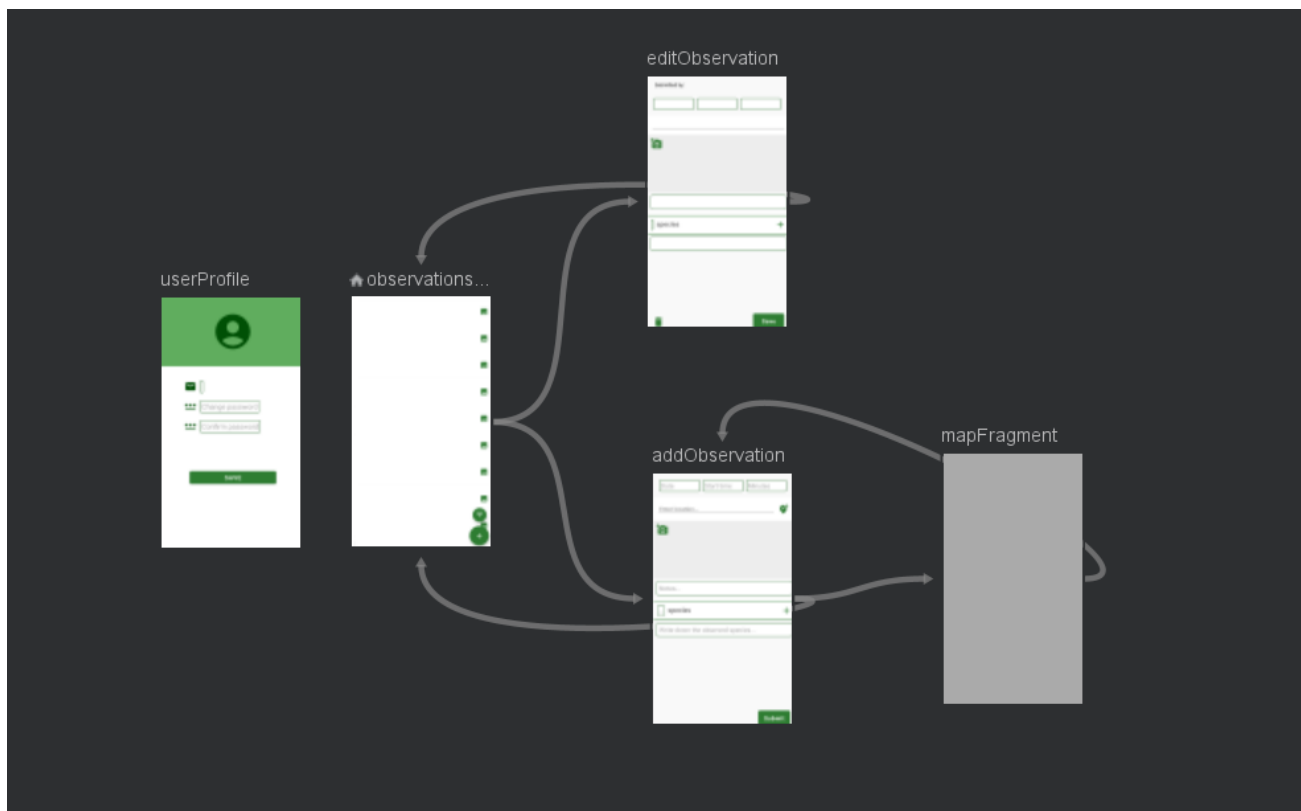


Фиг. 2.6 – Структура на екраните в RegistrationActivity

### 2.3.5.2. Фрагменти в UserActivity

UserActivity се стартира, след като потребителят се автентикира успешно. Дава достъп до основните функционалности и екрани. В него са разположение следните фрагменти (Фигура 2.7):

- UserProfile – екран, който позволява на потребителя да види акаунта, с който е влязъл в приложението, и да редактира паролата си.
- ObservationsFragment – в този екран потребителите могат да видят списък (с recycler view) на всички качени наблюдения.
- EditObservation – представя данните, снимката и автора на всяко едно наблюдение. Потребителите имат право да редактират данните и изображението в своите собствени записи.
- AddObservation – екран с формуляр за въвеждане на ново наблюдение
- MapFragment – Google Maps карта с настоящото местоположение на потребителя



Фиг. 2.7 – Структура на екраните в UserActivity



# Трета глава

## Програмна реализация на мобилно приложение за мониторинг на защитени видове

### 3.1 Използавни практики и библиотеки

#### 3.1.1. Navigation –

Navigation graph е компонент на Android Jetpack Navigation, който позволява да се създаде потребителският поток в приложението. Има графично представяне на потребителските интерфейси на различните екрани (фрагменти) в приложението, както и начина на тяхното свързване и преход между тях. В navigation graph всяко activity или fragment се представя със своя дестинация (destination), която може да бъде свързана с други дестинации чрез навигационни “действия” (actions). Всяко действие дефинира начина на преминаване от един екран към друг.

Компонетът включва всички екрани (фрагменти) и действия (actions) в една цялостна дървовидна структура.

Navigation menu е компонент в Android, който позволява на потребителя да се навигира между различни части на приложението. Често се представя като списък с опции, които могат да се изберат за преминиване в съответния екран. Навигационното меню може да се създаде чрез XML<sup>[5]</sup> файлове, които определят опциите и връзките между тях. Подобрява потребителското изживяване, като дава повече опции за навигация между екраните. Може да се добави към приложението чрез DrawerLayout, BottomNavigationView и други.

В разработката на мобилното приложение изпозвам меню (*Фигура 3.1*) за достъпване на картата, профила, екрана за добавяне на наблюдения и списъка със записи.

```

private fun onNavigationItemSelected(menuItem: MenuItem) {
    when (menuItem.itemId) {
        R.id.nav_profile -> {
            navController.navigate(R.id.userProfile)
            drawerLayout.closeDrawer(GravityCompat.START)
        }
        R.id.nav_observations -> {
            navController.navigate(R.id.observationsFragment)
            drawerLayout.closeDrawer(GravityCompat.START)
        }
        R.id.nav_addObservation -> {
            navController.navigate(R.id.addObservation)
            drawerLayout.closeDrawer(GravityCompat.START)
        }
        R.id.nav_map -> {
            navController.navigate(R.id.mapFragment)
            drawerLayout.closeDrawer(GravityCompat.START)
        }
        else -> {
            NavigationUI.onNavDestinationSelected(menuItem, navController)
            drawerLayout.closeDrawer(GravityCompat.START)
        }
    }
}
}

```

Фиг. 3.1 – Функция за навигация с меню

### 3.1.2. LiveData –

LiveData е клас, който може да наблюдава за промени в данните на приложението. Той е предназначен да улесни имплементацията на архитектурни модели като MVVM. Съобразява се с жизнените цикли (lifecycle-aware) на сървиси и компоненти като activity и fragment. Основната му полза за проекта е, че може на базата на неговите промени да се изпълняват различни събития. Например да следи за промени в базата данни и да ги отразява автоматично, което да се използва за изкарване на новите наблюдения в ObservationsFragment и премахване на изтрити такива, без да има нужда потребителят да презарежда екрана (Фигура 3.2).

```

private val _myObservations = MutableLiveData<List<ObservationData>>()
val myObservations: LiveData<List<ObservationData>>
    get() = _myObservations

init {
    //Initial loading of data
    observationsCollection.get().addOnSuccessListener { snapshot ->
        val observations = snapshot.documents.mapNotNull { document ->
            document.toObject(ObservationData::class.java)?.copy(id = document.id)
        }
        _myObservations.value = observations
    }.addOnFailureListener { e ->
        Log.e( tag: "ObservationsVM", msg: "Error getting documents: ", e)
    }
}

```

Фиг. 3.2 – Тук се инициализиране обект на LiveData, който може да се наблюдава от други компоненти, и се зареждат данните на \_myObservations от базата данни при създаването на ViewModel-a.

### 3.1.3. Glide

Glide е библиотека за управление на изображенията в Android проекти. Предоставя лесен начин за зареждане на изображения от различни източници и тяхното визуализиране в приложението. Има и механизми за кеширане на снимките и оптимизиране на техния размер, което позволява да бъдат зареждани бързо дори при по-бавна връзка. Това, плюс оптимизирането на използваната памет, прави технологията изключително добра за проекта.

Използвам библиотеката във фрагменти AddObservation и EditObservation, за да визуализирам снимките (Фигура 3.3), които са били качени.

```

val uploadTask = storageRef.putFile(imageUri)
uploadTask.continueWithTask {...}.addOnCompleteListener { task ->
    if (task.isSuccessful) {
        val downloadUri = task.result
        Glide.with(requireContext()) RequestManager
            .load(downloadUri.toString()) RequestBuilder<Drawable!>
            .into(binding.imageView)
    } else {...}
}

```

Фиг. 3.3 – Визуализиране на снимката във фрагмент AddObservation, след като е била качена успешно

## 3.2. Регистрация и автентикация на потребителите

### 3.2.1. Регистрация

При пускане на приложението се стартира RegistrationActivity, което има два фрагмента – LoginFragment (автентикация) и SignUpFragment (регистрация). Това activity има собствен navigation graph, който се казва registration\_nav\_graph (Фигура 3.4). В него като стартова дестинация е зададен LoginFragment, но има action за навигиране, който позволява да се достъпи екрана за регистрация.

```

<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/registration_nav_graph"
    app:startDestination="@id/loginFragment">

    <fragment
        android:id="@+id/loginFragment"
        android:name="com.example.monitoringapp.LoginFragment"
        android:label="Log In"
        tools:layout="@layout/fragment_log_in" >
        <action
            android:id="@+id/action_loginFragment_to_signUpFragment"
            app:destination="@id/signUpFragment"
            app:popUpToInclusive="true" />
    </fragment>

```

Фиг. 3.4 – registration\_nav\_graph и action\_loginFragment\_to\_signUpFragment

След като нерегистриран потребител достъпи SignUpFragment екрана, ще види три EditText полета и бутон “Sign Up”, който при натискане трябва да създаде нов профил във Firebase Authentication. Преди да се създаде новият профил, трябва да се направи проверка, по различни критерии, на въведените данни (Фигура 3.5).

```
if (email.isBlank()) {  
    Toast.makeText(context, text: "You must enter email", Toast.LENGTH_SHORT).show()  
    return@setOnClickListener  
}  
  
if (!emailPattern.matcher(email).matches()) {  
    Toast.makeText(context, text: "Invalid email format", Toast.LENGTH_SHORT).show()  
    return@setOnClickListener  
}  
  
if (password.isBlank() || confirmPassword.isBlank()) {  
    Toast.makeText(context, text: "You must enter password", Toast.LENGTH_SHORT).show()  
    return@setOnClickListener  
}  
  
if (password.length < 8){  
    Toast.makeText(context, text: "Password must be at least 8 characters long", Toast.LENGTH_SHORT).show()  
    return@setOnClickListener  
}  
  
if (password != confirmPassword) {  
    Toast.makeText(context, text: "Passwords don't match", Toast.LENGTH_SHORT).show()  
    return@setOnClickListener  
}
```

Фиг. 3.5 – Проверка на въведените данни

В кода е конфигуриран `setOnClickListener` за бутона, който да следи за натискания. При неговото кликуване се извличат въведените данни за имейл, парола и потвърждение на паролата (`confirm password`) от текстовите полета. След това се проверява дали всички полета са попълнени, дали имейлът е във валиден формат, дали паролите съвпадат и имат поне 8 символа. Ако някои от тези условия не са изпълнени, се извежда съобщение за грешка във формата на Toast.

При успешно и правилно въвеждане на необходимите данни, се извършва създаване на нов потребител в системата (Фигура 3.6).




```

authentication.createUserWithEmailAndPassword(email, password).addOnCompleteListener{ it: Task<AuthResult!>
    if (it.isSuccessful) {
        Toast.makeText(context, text: "Successfully Signed Up", Toast.LENGTH_SHORT).show()
        Navigation.findNavController(binding.root).navigate(R.id.action_signUpFragment_to_LogInFragment)
    } else {
        Log.e( tag: "SIGN_UP", msg: "Failed to sign up", it.exception)
        Toast.makeText(context, text: "Sign Up Failed!", Toast.LENGTH_SHORT).show()
    }
}
}

```

Фиг. 3.6 – Създаване на нов акаунт с authentication.  
createUserWithEmailAndPassword()

Методът createUserWithEmailAndPassword() е от клас FirebaseAuth() и се използва за генериране на потребителски акаунти във Firebase Authentication (Фигура 3.7). Като параметри подаваме въведените имейл и парола. След това използвам addOnCompleteListener, който да слуша за завършването на операцията по създаване на потребителя. Ако операцията е успешна, потребителят се препраща към страницата за вход в приложението.

denis@gmail.com		Mar 7, 2023	Mar 7, 2023	hM2toYytTNab2a3b4mCbMxWRG...
ivaylo@gmail.com		Mar 7, 2023	Mar 7, 2023	C8KnEHN5biPrCid40KiaaskJuC3
milko@gmail.com		Mar 5, 2023	Mar 5, 2023	unL2NNggd8fAHlITwXGxrXu1q6l2
anatolfrenski@abv.bg		Mar 5, 2023	Mar 13, 2023	pva0bdlDztPEzDcPYfjBRDTu2Lj2
ivayloivalyo@gmail.com		Feb 26, 2023	Mar 14, 2023	YIW7TZqVtvcbOaa61n8gNN32IdR2

Фиг. 3.7 – Панел с регистрирани потребители във Firebase Authentication

### 3.2.2. Автентикация

След като потребителят успешно е създал своя акаунт, се изисква да въведе в LogInFragment паролата и електронната поща, с които се е регистрирал, за да достъпи UserActivity-то в приложението и да получи достъп да неговите функционалности. Подобно на SignUpFragment, този фрагмент също има проверки за празни полета и формат на имейла, които връщат setOnClickListener при грешно въведена информация.

Методът `signInWithEmailAndPassword()` приема като параметри въведените имейл и парола (Фигура 3.8) и връща обект от тип `AuthResult`. После въвежда данните за автентикация на потребителя в системата. След като този task бъде приключен успешно, се връща Toast съобщение и се стартира `UserActivity`, а при неуспешен опит се изписва грешка.

```
authentication.signInWithEmailAndPassword(email, password).addOnCompleteListener{ it: Task<AuthResult!>
    if (it.isSuccessful) {
        Toast.makeText(context, text: "Successfully Logged in", Toast.LENGTH_SHORT).show()
        startActivity(Intent(requireActivity(), UserActivity::class.java))
        requireActivity().finish()
    } else {
        Log.e( tag: "SIGN_UP", msg: "Failed to sign up", it.exception)
        Toast.makeText(context, text: "Wrong email or password!", Toast.LENGTH_SHORT).show()
    }
}
```

Фиг. 3.8 – Употреба на `signInWithEmailAndPassword()` в `LogInFragment`

### 3.3. Управление на потребителски профил

След успешно влизане в своя профил, потребителите могат да се навигират с менюто до `UserFragment`, който предоставя възможност да редактират паролата, с която са се регистрирали. Екранът има две полета за въвеждане на текст, като въведените пароли се проверяват за брой символи и дали са еднакви.

При коректно попълване на всички полета, се извиква методът `updatePassword()`, който взима като параметър новата парола (Фигура 3.9) и я задава за профила на потребителя. При успех се стартира наново `RegistrationActivity`.

```
authentication.currentUser?.updatePassword(newPassword)?.addOnCompleteListener { task ->
    if (task.isSuccessful) {
        Toast.makeText(context, text: "Password updated successfully", Toast.LENGTH_SHORT).show()
        startActivity(Intent(requireActivity(), RegistrationActivity::class.java))
        requireActivity().finish()
    }
}
```

Фиг. 3.9 – Употреба на `updatePassword()` в `UserFragment`

## 3.4. Списък с наблюдения

### 3.4.1. Представяне на списъка от наблюдения

Списъкът с наблюдения се представя в `ObservationsFragment` с помощта на `RecyclerView`. Това е компонент на Android, който ефективно представя списъци от данни и има много функционалности. Работи като контейнер, който може да показва голям брой елементи в списък, без да натоварва паметта на устройството. Ефективен е, защото изгражда само необходимите елементи в момента на показването им и преработва (“рециклира”) останалите, които потребителят скролва нагоре или надолу.

Използва `item_layout` (Фигура 3.10) XML файл, който представя дизайна и полетата на всеки елемент, и адаптер (Фигура 3.11), който да свърже `RecyclerView` с необходимите данните. Има и клас `ViewHolder`, в който се дефинират компонентите, които да се извеждат.

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable name="individualObservation" type="com.example.monitoringapp.data.ObservationData"/>
    </data>

    <androidx.cardview.widget.CardView...>
</layout>
```

Фиг. 3.10 – `item_layout.xml`

В `item_layout` има `binding expression` с променлива от тип `ObservationData`. `ObservationData` е `data` клас, който съдържа информация за променливите, които съответстват на полетата във `Firestore` документите.



```

class ObservationViewHolder(val binding: ItemLayoutBinding) : RecyclerView.ViewHolder(binding.root)

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ObservationViewHolder {
    val view = ItemLayoutBinding.inflate(LayoutInflater.from(parent.context), parent, attachToRoot = false)
    return ObservationViewHolder(view)
}

override fun onBindViewHolder(holder: ObservationViewHolder, position: Int) {
    val individualObservation = observations[position]
    holder.binding.individualObservation = individualObservation

    holder.itemView.setOnClickListener { it: View!
        onItemClick?.invoke(individualObservation)
    }

    Log.d(tag: "ObservationAdapter", msg: "Observations: $observations")
}

override fun getItemCount(): Int {
    return observations.size
}

```

Фиг. 3.11 – Адаптер ObservationRecyclerViewAdapter

В адаптера се използват няколко метода, които отговарят за разпределянето и добяване на данни в RecyclerView. Това са:

- onCreateViewHolder() – Извиква се, когато създаде и инициализира нов ViewHolder (в случая ObservationViewHolder).
- onBindViewHolder() – Изпълва класа ObservationViewHolder с данни, за да може после те да се представят на екрана.
- getItemCount() – Връща големината на списъка с данни (наблюдения).

Има и допълнително добавена функция (Фигура 3.12) – updateObservations(), която взима като параметър списък от тип ObservatioData. Използва се, за да актуализира списъка с наблюдения и да съобщи на приложението, че са настъпили промени в адаптера, заради които recycler-ът ще трябва да се генерира наново.

```

fun updateObservations(newObservations: List<ObservationData>) {
    observations = newObservations
    notifyDataSetChanged()
}

```

Фиг. 3.12 – updateObservations()

Създаването на Recycler View и настройването на адаптера (Фигура 3.13) се изпълнява в метода onViewCreated() на ObservationsFragment. В този метод се прави референция към LiveData обекта от ObservationsViewModel и viewLifecycle се задава като собственик на жизнения цикъл, за да се гарантира, че UI на екрана ще се актуализира при промени в данните. За настройването на адаптера за recyclerView, се прави инстанция на ObservationsRecyclerAdapter, като се подава празен списък на конструктора. Накрая myObservations се наблюдава за промени, а ако има такива се извиква методът updateObservations().

```

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val myObservations = viewModel.myObservations

    binding.lifecycleOwner = viewLifecycleOwner

    val recyclerView = binding.observationRecycler
    adapter = ObservationsRecyclerAdapter(emptyList())
    recyclerView.adapter = adapter
    recyclerView.setHasFixedSize(false)

    adapter.setOnItemClickListener { observation ->
        val bundle = bundleOf( ...pairs: "id" to observation.id)
        Navigation.findNavController(binding.root).navigate(R.id.action_observationsFragment_to_)
    }

    myObservations.observe(viewLifecycleOwner) { observations ->
        adapter.updateObservations(observations)
    }
}

```

Фиг. 3.13 – Създаване на Recycler View и настройване на адаптера в ObservationsFragment

### 3.4.2. Филтриране

В списъка с наблюдения трябва да има възможност потребителят да подрежда отделните записи по даден критерий. За дипломната работа е реализирана опция за сортиране на списъците по брой на въведените видове. Това ще подрежда отделните наблюдения в низходящ ред, като ще позволи на потребителите да видят в кои райони има голямо биоразнообразие.

За активиране на филтъра, в интерфейса на фрагмента ObservationsFragment има създаден плаващ бутон, за който има свързан `setOnClickListener()`. При натискане на бутона се изпълнява заявка (Фигура 3.14) към базата данни, която да сортира полетата от тип “species” в низходящ ред с `.orderBy(„species“, Query.Direction.DESCENDING)`.

Ако заявката е успешна, се връща обект `QuerySnapshot`, който съдържа сортираните наблюдения, и после се конвертира в списък от `ObservationData` обекти. Накрая адаптерът се актуализира с `updateObservations()`.

```
binding.filterFabButton.setOnClickListener { it.View!  
    val query = FirebaseFirestore.getInstance().collection( collectionPath: "observations")  
        .orderBy( field: "species", Query.Direction.DESCENDING)  
    query.get().addOnSuccessListener { querySnapshot ->  
        val sortedObservations = querySnapshot.toObject(ObservationData::class.java)  
            .sortedByDescending { it.species.toIntOrNull() }  
        Log.d( tag: "ObservationsFragment", msg: "Filtered observations: $sortedObservations")  
        adapter.updateObservations(sortedObservations)  
        adapter.notifyDataSetChanged()  
    }.addOnFailureListener { exception ->  
        Log.e( tag: "ObservationsFragment", msg: "Error getting observations", exception)  
    }  
}
```

Фиг 3.14 – Заявка за сортиране по брой видове

## 3.5. Достъп и управление на наблюденията

### 3.5.1. Отваряне на наблюдение

Освен да представя основната информация за записите в списък, приложението трябва да позволява при натискане на някое от наблюденията в `ObservationsFragment` да се отваря нов екран, който да съдържа пълната информация за селектираното наблюдение и неговата снимка, ако има такава.

За да се имплементира това, във фрагмента със списъка е поставен `setOnItemClickListener` (Фигура 3.15), който да следи за натискане на някой `item` от `recycler-a`.

При засечено кликване, се създава `bundle` обект, който има зададена стойност `id`-то на селектираното наблюдение. Накрая класът `Navigation` се използва, за да се намери `NavController` и да се отвори “`editObservation`” дестинацията с подаден аргумент `bundle`. Това позволява при отварянето на фрагмента `EditObservation`, той да е населен с данните на документа с това `Id`.

```
adapter.setOnItemClickListener { observation ->
    val bundle = bundleOf( ...pairs: "id" to observation.id)
    Navigation.findNavController(binding.root)
        .navigate(R.id.action_observationsFragment_to_editObservation, bundle)
}
```

Фиг. 3.15 – `setOnItemClickListener` в `ObservationsFragment`

### 3.5.2. Редактиране на наблюдение

Потребителите на приложението трябва да имат опцията да редактират наблюдения, които са качили, и да могат да ги изтриват. За тази цел е създаден `EditObservation` класа, в който има бутони за изтриване и запазване.

Потребителят не трябва да има право да променя записи, които не е въвел през свой акаунт. Това може да се предотврати по различни начини като две решения са имплементирани в дипломната работа.

Първото решение е да се настройт правилата (Rules) за четене, добавяне и презаписване на документи във Firestore. За колекцията “observations” на проекта съм добавил следните правила (*Фигура 3.16*):

- Всеки потребител има право да чете документите
- Само автентикираните потребители могат да създават документи
- Само автентикиран потребител, който е въвел дадения документ, има право да го променя

```
1 rules_version = '2';
2 service cloud.firestore {
3   match /databases/{database}/documents {
4     match /observations/{id} {
5       allow read: if true;
6       allow create: if request.auth != null;
7       allow update, delete: if request.auth != null && request.auth.token.email == resource.data.creator;
8     }
9   }
10 }
```

*Фиг. 3.16* – Правила на базата данни

Второто решение е да се направи проверка за акаунта на потребителя при натискане на бутоните за изтриване и запазване. Това е изпълнено в `setOnClickListener`-ите на всеки от бутоните на фрагмента (*Фигура 3.17*). Сравнява се дали полето “creator” на документа съответства на имейла на сегашния потребител (`private val user = FirebaseAuth.currentUser`). Ако съвпада се извиква необходимата функция, иначе се връща Toast съобщение.

```
binding.saveButton.setOnClickListener { it: View!
    if (binding.observation?.creator == user?.email) {
        changeObservation()
    } else {
        Toast.makeText(requireContext(), text: "You can't change this observation", Toast.LENGTH_SHORT).
    }
}
```

*Фиг. 3.17* – Проверка за принадлежност на наблюдението при `saveButton`

Ако потребителят е собственик на отвореното наблюдение, може да променя всяко едно от неговите полета и после да запази промените с натискане на бутон `saveButton`, който от своя страна изпълнява метода `changeFragObservation()` в същия фрагмент. В него се проверява дали има празни полета, при наличие на такива се връща Toast съобщение, а ако няма – извиква се `ObservationsViewModel` функцията `changeObservation()`. Тя взима като аргумент нов `ObservationData` обект (Фигура 3.18), който съдържа променените входни данни. Първоначално функцията извиква `observationsCollection.document(observation.id)`, за да получи референция към документа във `Firestore`, който съответства на подадения `observation` обект. След това функцията използва метода `set()`, за да обнови съдържанието на документа.

```
fun changeObservation(observation: ObservationData) {  
    observationsCollection.document(observation.id).set(observation)  
        .addOnSuccessListener { it: Void!  
        Toast.makeText(context, text: "Observation is updated", Toast.LENGTH_SHORT).show()  
        Log.d( tag: "ObservationsVM", msg: "Observation updated successfully")  
        }  
        .addOnFailureListener { e ->  
        Log.e( tag: "ObservationsVM", msg: "Error updating observation", e)  
        }  
}
```

Фиг. 3.18 – Функция `changeObservation()`

Ако потребителят е създал отвореното наблюдение, има право да го изтрие с натискане на бутона `deleteButton` (Фигура 3.19). При продължително натискане на бутона се взима идентификатора на документа и се подава на функцията `deleteObservation` в `ObservationsViewModel`. След което с навигацията отива обратно на `ObservationsFragment`.

Самата функция `deleteObservation` изтрива документа с подаденото `Id` и връща съобщение в лога при успех или грешка.

```

binding.deleteButton.setOnLongClickListener { it: View!
    if (binding.observation?.creator == user?.email) {
        val observationId = arguments?.getString( key: "id")
        if (observationId != null) {
            viewModel.deleteObservation(observationId)
        }
        Navigation.findNavController(binding.root)
            .navigate(R.id.action_editObservation_to_observationsFragment)
        true ^setOnLongClickListener
    } else {
        Toast.makeText(requireContext(), text: "You can delete only your observations", Toast.LENGTH_SHORT)
        false ^setOnLongClickListener
    }
}
}

```

Фиг. 3.19 – Извикване на функция deleteObservation при натискане на deleteButton

### 3.5.3. Добавяне на наблюдение

Всеки автентикиран потребител може да качва в платформата нови наблюдения като попълни формуляра в AddObservation фрагмента. Неговият интерфейс е сходен с този на EditObservation, но липсва бутон за изтриване и поле, което да показва кой е автор на записа.

За да се въведе ново наблюдение се натиска бутон submitButton, който от своя страна изпълнява метода addFragObservation() в същия фрагмент. В него се проверява дали има празни полета (Фигура 3.20), при наличие на такива се връща Toast съобщение, а ако няма – извиква се ObservationsViewModel функцията addObservation(). След това навигацията връща потребителя на ObservationsFragment екрана.

Функцията addObservation() взима като аргумент нов ObservationData обект, който съдържа променените входни данни, и го добавя в “observations”, след което записва неговото id.

```

if (observation.date.isEmpty() ||
    observation.hour.isEmpty() ||
    observation.location.isEmpty() ||
    observation.duration.isEmpty() ||
    observation.species.isEmpty() ||
    observation.speciesDetails.isEmpty()) {
    Toast.makeText(context, text: "You can't have empty fields", Toast.LENGTH_SHORT).show()
    return
}else{
    viewModel.addObservation(observation)
    findNavController().navigateUp()
}

```

Фиг. 3.20 – Проверка за празни полета в AddObservation

### 3.6. Добавяне на снимки

Във всяко наблюдение трябва да е възможно да се качи снимка, която може да изобразява конкретен екземпляр от наблюдаваните видове или хабитата в района.

За тази цел във фрагмента AddObservation е имплементирана функцията `uploadImage()`, която взима като аргумент `URI`<sup>[6]</sup> на селектирано от галерията изображение (Фигура 3.21). При подаване на този `URI`, се генерира уникален идентификатор за изображението и се създава референция към мястото за съхранение (“photos/”) на Firebase Storage. Извиква се методът `putFile()`, за да се качи файлът в хранилището. При приключване на качването се взима препратка към `URL`-а. Ако операцията е успешна, изображението се зарежда в `ImageView` с помощта на `Glide` библиотеката. При грешка се показва съобщение на потребителя с помощта на `Toast`.

Във фрагмента за редактиране на наблюдения (EditObservation) има опция да се качи нова снимка. Това става с помощта на функцията `updateImage()`, която има идентична структура с `uploadImage()`, но при добавянето на новата снимка подsigурява, че старата ще бъде изтрита от облачното хранилище (Фигура 3.22). За тази цел първо прави проверка дали има качено изображение за този документ,



ако има такова – взима референция от хранилището за URL-на старото изображение и го изтрива.

```
private fun uploadImage(imageUri: Uri) {
    val imageName = UUID.randomUUID().toString()
    val storageRef = Firebase.storage.reference.child( pathString: "photos/$imageName")

    val uploadTask = storageRef.putFile(imageUri)
    uploadTask.continueWithTask { task ->
        if (!task.isSuccessful) {
            task.exception?.let { throw it }
        }
        storageRef.downloadUrl ^continueWithTask
    }.addOnCompleteListener { task ->
        if (task.isSuccessful) {
            val downloadUri = task.result
            Glide.with(requireContext()) RequestManager
                .load(downloadUri.toString()) RequestBuilder<Drawable!>
                .into(binding.imageView)
        } else {
            Log.e( tag: "UPLOAD", msg: "Failed to upload image: ${task.exception}")
            Toast.makeText(requireContext(), text: "Failed to upload image", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Фиг. 3.21 – uploadImage() във фрагмента AddObservation

```
val oldImageRef = Firebase.storage.getReferenceFromUrl(oldImageUrl)
oldImageRef.delete().addOnSuccessListener { it: Void!
    Log.d( tag: "UPDATE", msg: "Old image deleted successfully")
}.addOnFailureListener { e ->
    Log.e( tag: "UPDATE", msg: "Failed to delete old image", e)
}
```

Фиг. 3.22 – Изтриване на старо изображение във фрагмента EditObservation

Качените изображения се изтриват по подобен начин и във ObservationsViewModel функцията deleteObservation(). Това е направено с цел да не се заема излишно хранилищно място при изтриване на наблюдение във фрагмента за редактиране.

### 3.7. Визуализиране на карта

В приложението потребителят трябва да е способен да достъпи карта, която да показва неговото местоположение.

За реализацията на тази функционалност в дипломната работа е включен Google Maps SDK за Android, който да позволи да се използват услугите на Maps и да улесни представянето на локации с пинчета.

Създаден е MapFragment, в който се използват “...`gms.location.LocationServices`”, за взимане на локация от устройството, и “...`gms.maps.model.MarkerOptions`” за поставяне на маркер (пинче).

Във фрагмента е дефинирана

`PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION` константа, която се използва като код за заявка за достъп до местоположението на потребителя.

Когато приложението иска да използва локацията на устройството, потребителят трябва да даде необходимото разрешение. Заради това трябва да се изпрати заявка, която се идентифицира с определен код. След като потребителят реагира на заявката за разрешение, приложението може да провери кода, за да разбере дали е била одобрена или отхвърлена.

В метода `onCreateView` се създава инстанция на layout-а за фрагмента (“`fragment_map`”) и се инициализира `MapView` обекта (*Фигура 3.23*). `MapView` се използва за визуализация на картата и в него се поставя API ключът, който се генерира от Google Maps. Накрая на метода се извиква `getMapAsync()` с параметър `this`, който да съобщи на картата да използва `onMapReady()`, когато се генерира напълно.

При `onMapReady` (*Фигура 3.24*) се инициализира `GoogleMap` обектът. Въведени са и настройките на потребителския интерфейс за картата – `isMyLocationButtonEnabled` и `isZoomControlsEnabled`.

```
<com.google.android.gms.maps.MapView
    android:id="@+id/google_map"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:apiKey="AIzaSyAQsgMRDFmjJuJiV6dz4x0zuJcFwjiE0u0"
    android:clickable="true"
    android:enabled="true"
    android:focusable="true" />
```

Фиг. 3.23 – MapView обект в layout-a на MapFragment

```
override fun onMapReady(googleMap: GoogleMap) {
    this.googleMap = googleMap
    googleMap.uiSettings.isMyLocationButtonEnabled = true
    googleMap.uiSettings.isZoomControlsEnabled = true
    enableMyLocation()
}
```

Фиг. 3.24 – Метод OnMapReady в Map Fragment

В края на OnMapReady() се извиква функцията enableMyLocation(), в която се проверява дали приложението има разрешение да достъпва местоположението на потребителя. Ако има, isMyLocationEnabled на обекта GoogleMap се задава на “true”, което показва текущото местоположение на потребителя на картата (Фигура 3.25).

```
private fun enableMyLocation() {
    if (ContextCompat.checkSelfPermission(
        requireContext(),
        Manifest.permission.ACCESS_FINE_LOCATION
    ) == PackageManager.PERMISSION_GRANTED
    ) {
        googleMap.isMyLocationEnabled = true
    }
}
```

Фиг. 3.25 – enableMyLocation() в MapFragment

На картата се добавя маркер (Фигура 3.26) с текущото местоположение на потребителя и “камерата” на картата се премества, за да се центрира върху текущото местоположение.

Ако приложението няма разрешение, се извиква методът

ActivityCompat.requestPermissions() с параметър  
permission.ACCESS\_FINE\_LOCATION.

```
fusedLocationClient.lastLocation
    .addOnSuccessListener { location: Location? ->
        if (location != null) {
            val currentLatLng = LatLng(location.latitude, location.longitude)
            googleMap.addMarker(
                MarkerOptions()
                    .position(currentLatLng)
                    .title("Current location")
            )
            googleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(currentLatLng, zoom: 15f))
        }
    }
```

Фиг. 3.26 – Добавяне на маркер и центриране на камерата

# Четвърта глава

## Ръководство на потребителя

### 4.1. Системни изисквания и начин на инсталация

За стартиране на мобилното приложение е необходимо мобилно устройство с операционна система Android или емулатор/виртуална машина, която поддържа Android.

Подходящ е емулаторът на интегрираната среда за разработка Android Studio, но изисква компютърната конфигурация да има над 4 GB RAM памет.

За да се инсталира и отвори приложението са необходими следните стъпки:

- Отидете на този адрес:  
[https://github.com/IviGeorgiev/MonitoringApp\\_Diplomna](https://github.com/IviGeorgiev/MonitoringApp_Diplomna),  
и клонирайте github хранилището.
- 1) След това директно можете да изтеглите последната версия на apk<sup>[7]</sup> файла и да го заредите на устройството си. Трябва да одобрите “Инсталация на приложения от непознати източници”, ако се появи като опция на екрана Ви.
- 2) Ако стартирате приложението с този метод, не е нужно да следвате останалите стъпки.
- Изтеглете средата Android Studio от официалния сайт. Адрес:  
<https://developer.android.com/studio>
- Заредете папката “MonitoringApp\_Diplomna” в средата
- Може да стартирате проекта по два начина:
  - 1) Използвайте предоставения емулатор и му задайте подходяща версия на Android. След това натиснете “Run app”

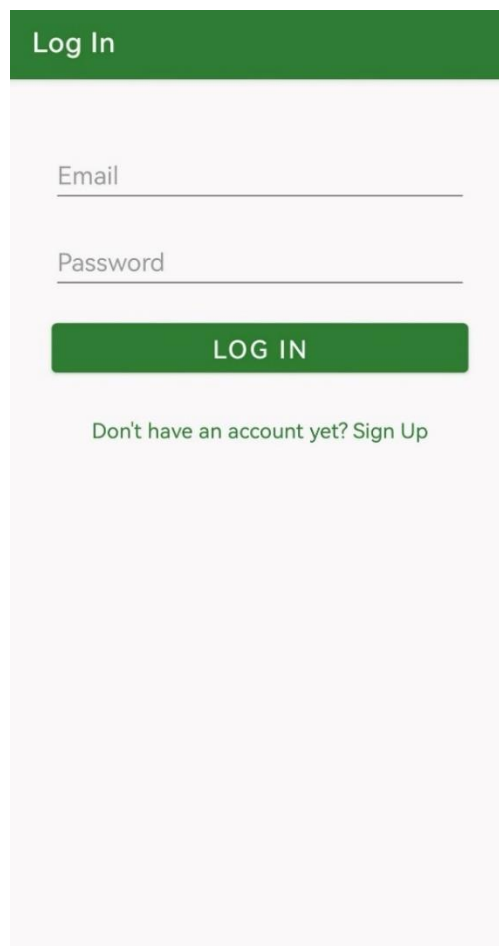
2) Свържете Android устройство с компютър, използвайки подходящ USB адаптер. На устройството Ви трябва да излезе меню с опции “Прехвърляне на снимки”, “Прехвърляне на файлове”, “Само зареждане”. Изберете “Прехвърляне на файлове”. В настройките на телефона/таблета потърсете “Настройки за разработчици” и одобрете “Дебъгване с USB”. След това в Android Studio селектирайте съответстващия идентификатор в “Device Manager” и натиснете “Run app”.

Първите два метода са препоръчителни, защото не изискват добавяне на много допълнителни разрешения към хардуерно устройство.

## 4.2. Ръководство за използване

При зареждане на мобилното приложение, първият екран, който се визуализира, е “Log In” (*Фигура 4.1*), означение в проекта – LogInFragment. В него потребителят може да въведе своите имейл и парола, ако има създаден акаунт. Ако все още не се е регистрирал, може да натисне

“Don’t have an account yet? Sign Up”, което ще го изкара на екран с етикет “Sign Up” (*Фигура 4.2*), означение в проект – SignUpFragment.



Log In

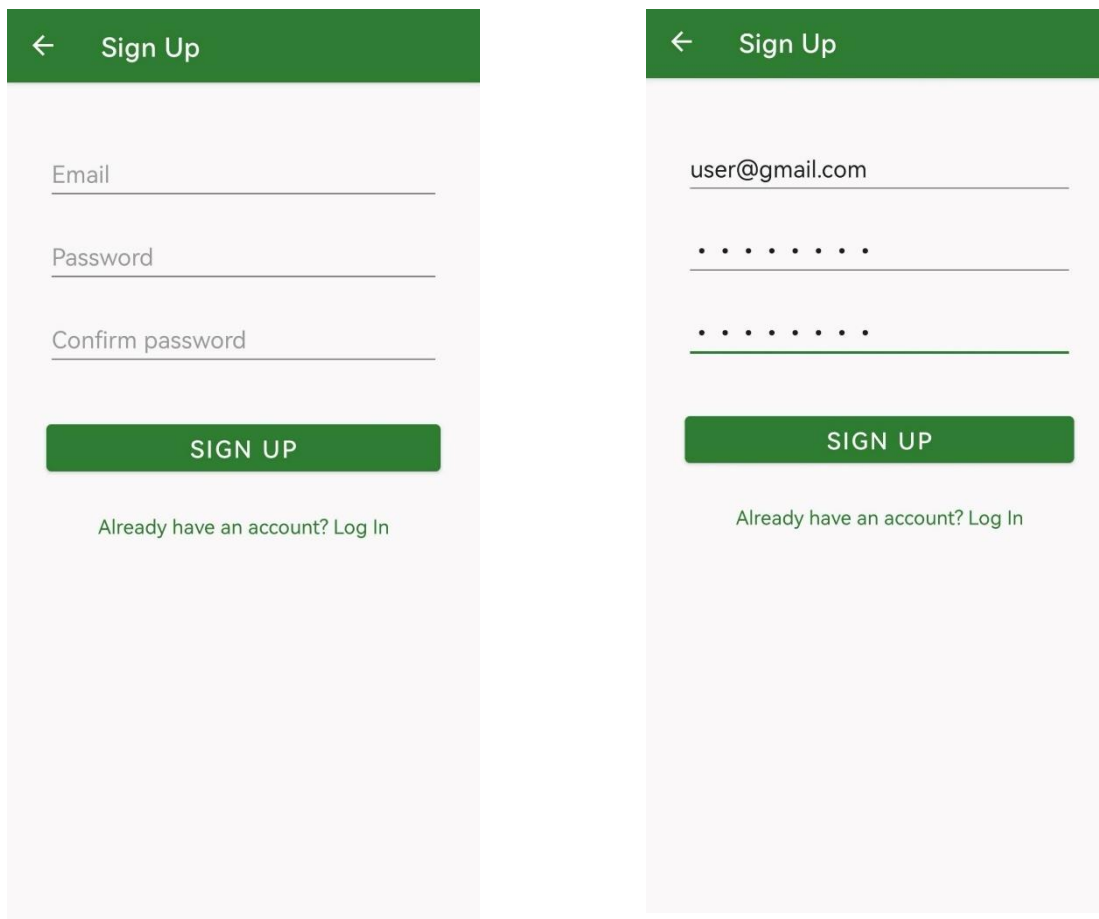
Email

Password

LOG IN

Don't have an account yet? Sign Up

*Фиг. 4.1* – Екран за влизане в профил

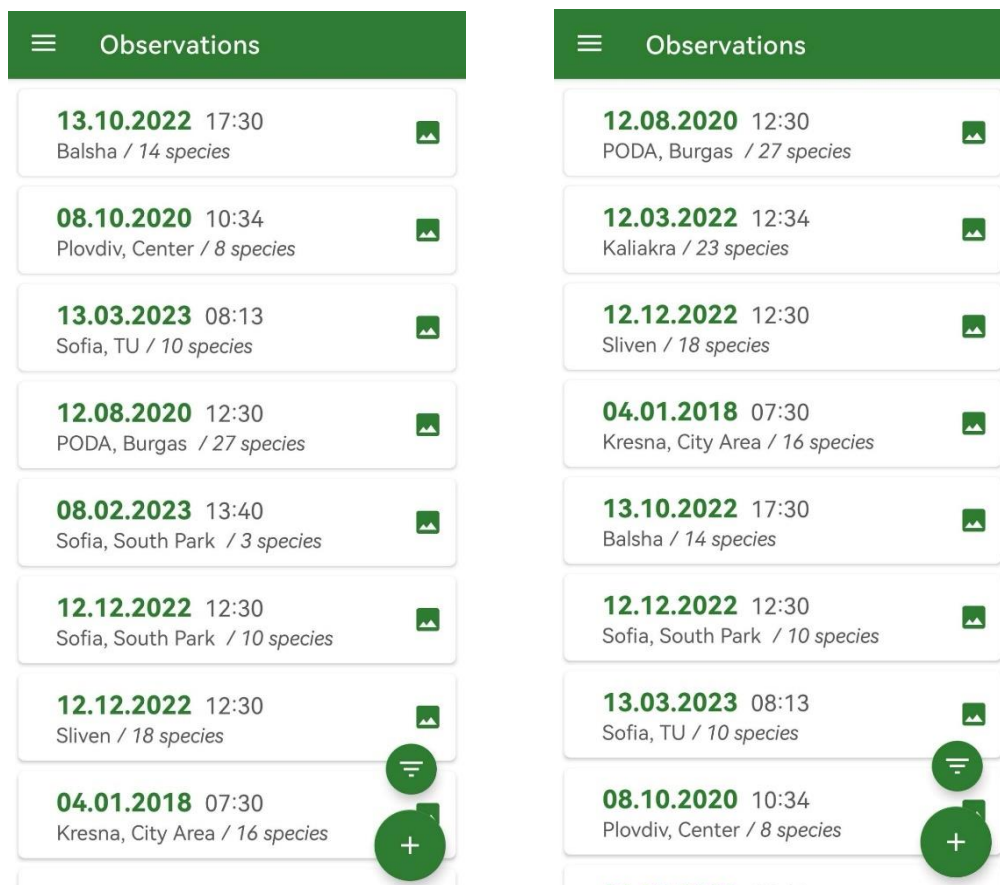


Фиг. 4.2 – Празен и попълнен екран за регистрация

За да се регистрира, потребителят трябва да въведе имейл в подходящ формат и две еднакви пароли, които имат повече от 8 символа. След натискане на “Sign Up”, ако регистрацията мине успешно – пак се показва екранът “Log In”.

Когато се въведат необходимите данни за автентикация, се отваря екран “Observations” (Фигура 4.3), означение в проекта – ObservationsFragment. В него потребителят може да разглежда въведените наблюдения, да ги филтрира по брой на видовете с плаващия бутон за филтрирани (три хоризонтални черти) или да достъпи формуляра за добавяне на наблюдение с “плюс” бутона.





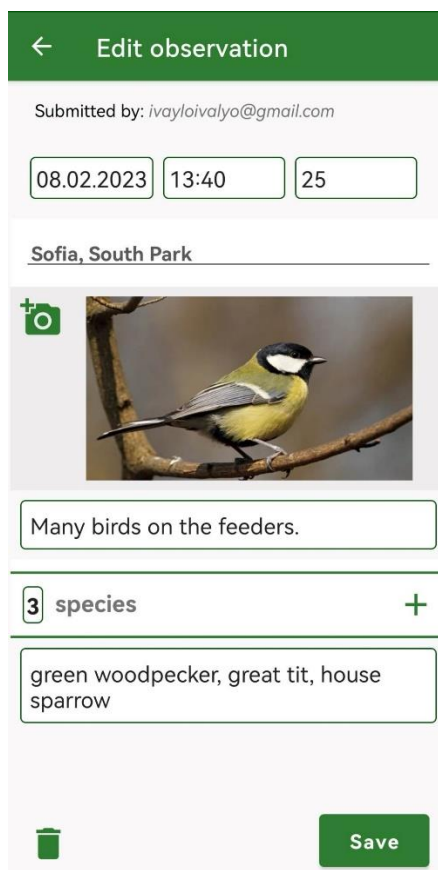
Фиг. 4.3 – Екран за представяне на наблюденията преди и след филтриране

При натискане на някое от наблюденията се отваря екран “Edit Observation” (Фигура 4.4) с данните за съответното наблюдение. Това включва автор на наблюдението, дата, час на започване, продължителност, локация, коментар, брой видове и списък с видовете.

Ако потребителят отвори свое наблюдение, има право да променя данните във всяко едно от полетата и да добави нова снимка. Не може да запазва наблюдения с празни полета.

За да добави ново наблюдение, потребителят трябва да достъпи екрана “Add Observation”. Това може да стане с бутона в “Observations” или с отваряне на падащото меню (Фигура 4.5) и избиране на “Add Observation” (Фигура 4.6). В екрана за добавяне на нов запис трябва да попълни всички полета от формуляра и опционално може да добави снимка. Има и бутон с карта, който му позволява да

отвори Gogle Maps карта. Ако разреши приложението да използва неговото местоположение, може да види къде се намира. Това би било полезно, ако наблюдателят посещава непознат район и не може да се ориентира как да опише локацията.




← Edit observation

Submitted by: ivayloivalyo@gmail.com

08.02.2023 13:40 25


Sofia, South Park



Many birds on the feeders.

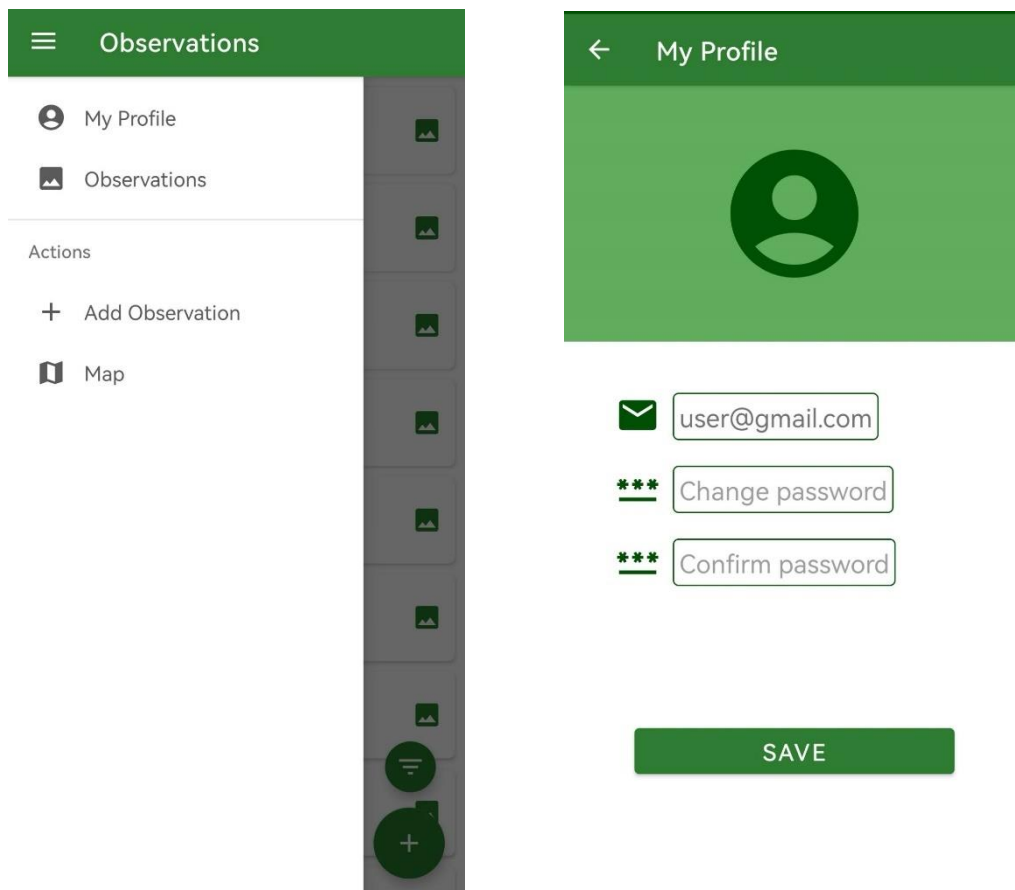
3 species +

green woodpecker, great tit, house sparrow

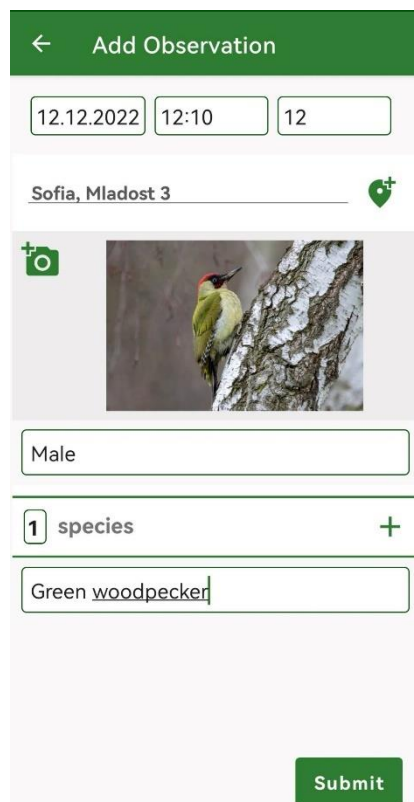
 Save

Фиг. 4.4 – Наблюдение отворено в “Edit Observation”

С падащото меню потребителят може да достъпи и екранът “My Profile”, означение в проекта – UserProfile, в който да види с кой имейл се е регистрирал и да смени паролата си.



Фиг. 4.5 – Падащо меню и екран “My Profile”



Фиг. 4.6 – Попълнен екран “Add Observation”

# Заклучение

Като заключение може да се обобщи, че реализацията на мобилното приложение за мониторинг на защитени видове е успешна. Голяма част от изначално поставените изисквания са изпълнени. Доволен съм от постигнатото и от многото придобити знания.

Приложението е подходящо за природолюбители от всички възрасти и потребителското изживяване е много сходно с това на други подобни платформи. С добавяне на допълнителни функционалности и промени в архитектурата, може да се превърне в един наистина полезен образователен и природозащитен инструмент. Ще е добре да се оптимизира предаването на данни, за да може приложението да работи по-бързо и ефективно. Работата с карта би била подобрена, ако потребителят има повече опции и действия, които може да изпълнява с нея.

Списъкът с функционалности, които би било полезно да се добавят в бъдеще, включва:

- Подобряване на потребителския интерфейс.
- Избиране на видовете от списъци.
- Добавяне на статии и допълнителна информация за различни видове животни и растения.
- Потребителска класация за хората с най-голям принос.

# Използвани термини

1. Отворен код – open-source software – Софтуер, чийто изходен код е достъпен за свободно преглеждане, използване, модифициране и разпространение
2. Затворен код – closed-source software – Софтуер, чийто изходен код не е достъпен за външни потребители, а е под контрола на компанията или лицето, което го е създало.
3. API – Application Programming Interface – Софтуер, който служи като посредник между две приложения и позволява да комуникират помежду си и да обменят информация, услуги и функционалности.
4. SDK – Software Development Kit - Набор от инструменти, библиотеки и ресурси, които позволяват на разработчиците да създават софтуерни приложения за дадена платформа или операционна система.
5. XML – Extensible Markup Language – Език за маркиране на данни, който се използва за съхраняване и прехвърляне на информация. В разработването на Android приложения, XML файловете се използват за създаване на екрани, менюта, списъци и други компоненти на потребителския интерфейс.
6. URI – Uniform Resource Identifier – Низ, който идентифицира уникално даден ресурс (например, уеб страница, файл, изображение и други) в мрежата.
7. APK – Android Package Kit – Файлов формат за инсталация на Android приложения.

## ИЗПОЛЗВАНИ ИЗТОЧНИЦИ

8. Mobile Operating System: <https://www.toppr.com/guides/computer-science/computer-fundamentals/operating-system/mobile-operating-system/>
9. IDEs for Mobile App Development - <https://geekflare.com/best-ide-for-mobile-app-development/>
10. Android operating system: <https://www.javatpoint.com/android-operating-system>
11. Android for developers: <https://developer.android.com/>
12. Kotlin Documentation: <https://kotlinlang.org/>
13. Cloud Firestore: <https://firebase.google.com/docs/firestore>
14. Cloud Storage for Firebase: <https://firebase.google.com/docs/storage>
15. Firebase Authentication: <https://firebase.google.com/docs/auth>
16. Maps SDK for Android overview:  
<https://developers.google.com/maps/documentation/android-sdk/overview>

# Съдържание

Увод.....	4
<b>Първа глава</b>	
Преглед на подобни приложения и съществуващи технологии за реализация на мобилни приложения.....	6
1.1. Преглед на подобни приложения.....	6
1.1.1. iNaturalist.....	6
1.1.2. SmartBirds Pro.....	7
1.1.3. eBird by Cornell Lab.....	7
1.2. Съществуващи технологии и развойни среди за реализиране на мобилно приложение.....	8
1.2.1. Мобилни операционни системи.....	8
1.2.1.1. Android.....	9
1.2.1.2. iOS.....	9
1.2.1.3. Други.....	9
1.2.2. Развойни среди за разработка на мобилни приложения.....	9
1.2.2.1. Android Studio.....	10
1.2.2.2. IntelliJ IDEA.....	10
1.2.2.3. Xcode.....	10
1.2.2.4. Visual Studio Code.....	11
1.2.3. Kotlin.....	11
1.2.4. Java.....	11
1.2.5. Firebase Cloud Firestore.....	12
<b>Втора глава</b>	
Проектиране на структурата на мобилното приложение.....	13
2.1. Функционални изисквания на мобилното приложение.....	13
2.2. Избор на технологии за реализацията на мобилното приложение.....	14
2.2.1. Избрана операционна система – Android.....	14
2.2.2. Избрана среда за разработка – Android Studion.....	14
2.2.3. Избран програмен език за разработка на приложението – Kotlin.....	15
2.2.4. Избор на система за база данни – Firestore.....	16
2.2.5. Firebase Cloud Storage.....	16
2.2.6. Firebase Authentication.....	17
2.2.7. Google Maps SDK за Android.....	17
2.3. Основни компоненти и структура на мобилното приложение.....	18
2.3.1. Activity.....	18
2.3.2. Fragment.....	19

2.3.3. Model-View-ViewModel архитектура.....	20
2.3.4. Структура на базата данни.....	21
2.3.5. Структура на екраните.....	22
2.3.5.1. Фрагменти в RegistrationActivity.....	22
2.3.5.2. Фрагменти в UserActivity.....	23
<b>Трета глава</b>	
Програмна реализация на мобилно приложение за мониторинг на защитени видове.....	25
3.1. Използвани практики и библиотеки.....	25
3.1.1. Navigation.....	25
3.1.2. LiveData.....	26
3.1.3. Glide.....	27
3.2. Регистрация и автентикация на потребителите.....	28
3.2.1. Регистрация.....	28
3.2.2. Автентикация.....	30
3.3. Управление на потребителски профил.....	31
3.4. Списък с наблюдения.....	32
3.4.1. Представяне на списъка с наблюдения.....	32
3.4.2. Филтриране.....	35
3.5. Достъп и управление на наблюденията.....	36
3.5.1. Отваряне на наблюдение.....	36
3.5.2. Редактиране на наблюдение.....	36
3.5.3. Добавяне на наблюдение.....	39
3.6. Добавяне на снимки.....	40
3.7. Визуализиране на карта.....	42
<b>Четвърта глава</b>	
Ръководство на потребителя.....	45
4.1. Системни изисквания и начин на инсталация.....	45
4.2. Ръководство за използване.....	47
Заклучение.....	52
Използвани термини.....	53
Използвани източници.....	54
Съдържание.....	55