Stredná odborná škola strojnícka
Partizánska cesta 76, 957 01 Bánovce nad Bebravou

Ročníkový projekt

Mobilná aplikácia pre evidenciu stromčekov

Dominik Ježík

2021

Stredná odborná škola strojnícka
Partizánska cesta 76, 957 01 Bánovce nad Bebravou

Mobilná aplikácia pre evidenciu stromčekov

Ročníkový projekt

Dominik Ježík

Študijný odbor: 3918 M technické lýceum
Trieda: IV. A
Vedúci ročníkového projektu: Ing. Pavel Hazucha
Dátum odovzdania práce: 4. februára 2021

Bánovce nad Bebravou 2021

Čestné vyhlásenie

Vyhlasujem, že som ročníkový projekt na tému: Mobilná aplikácia pre spravovanie stromov vypracoval samostatne s použitím uvedenej odbornej literatúry.

V Bánovciach nad Bebravou, dňa 4. februára 2021 .....

Dominik Ježík

Poďakovanie

Ďakujem môjmu konzultantovi Ing. Pavlovi Hazuchovi za jeho pomoc, cenné rady a usmernenia, ako aj trpezlivosť a ústretovosť počas vypracovávania môjho ročníkového projektu.

Obsah

Zoznam skratiek

API - Application Programming Interface

UI - User Interface

HTTP - Hypertext Transfer Protocol

JSON - JavaScript Object Notation

DI – Dependency Injection

SDK – Software Development Kit

IDE – Integrated Development Enviroment

0 Úvod

Táto práca prevádza čitateľa návrhom a tvorbou Android mobilnej aplikácie. Aplikácia bude určená pre mesto Bánovce nad Bebravou a jej úlohou je poskytnúť systém na evidenciu stromov v tomto meste. Mobilná aplikácia bude komunikovať s webovým serverom pomocou vlastného API systému. Aplikácia je určená pre dobrovoľníkov, ktorý budú zbierať informácia o stromoch v meste. Každý dobrovoľník si bude môcť vytvoriť účet cez aplikáciu a následne bude môcť pridávať stromy do nášho systému. Ku každému stromu ukladáme GPS súradnice stromu, typ stromu, fotografiu, prípadne ďalšie informácie. Dobrovoľník bude mať možnosť aktualizovať a v prípade potreby odstrániť stromy, ktoré boli pridané z jeho účtu. Nebude mať možnosť zasahovať do stromov, ktoré nepridal on sám. Túto možnosť má len administrátor.

API znamená Application Programming Interface a je to rozhranie pre komunikáciu servera s klientom – aplikáciou. Aplikácia bude vykonávať HTTP požiadavky na server. Server nevráti bežnú HTML stránku ako pri prehliadačoch, ale vráti nám dáta vo formáte JSON.

K tvorbe aplikácie použijeme moderný jazyk Kotlin a vývojové prostredie Android Studio, ktoré je oficiálne vývojové prostredie od spoločnosti Google a JetBrains. Poskytuje všetky potrebné nástroje pre vývoj natívnej aplikácie. Napríklad mobilný emulátor, inteligentný kód editor alebo build systém – Gradle.

Práca sa skladá z viacerých častí, ktoré opisujú teoretické a praktické časti práce. Teoretická časť popisuje použité technológie pri tvorbe aplikácie a tiež vysvetľuje ako fungujú základné stavebné bloky Android aplikácií. Praktická časť sa zaoberá konkrétnou tvorbou mobilnej aplikácie.

1 Problematika a prehľad literatúry

V tejto časti, problematika a prehľad literatúry, budú vysvetlené stavebné bloky Android aplikácií a takisto použité technológie v našom projekte. Základné stavebné bloky sú aktivita a fragment. Použité technológie môžeme v projekte používať vďaka technológií Gradle.

1.1 Android

Android je operačný systém pre mobilné zariadenia založený na upravenej verzií Linux jadra. Je navrhnutý pre zariadenia s dotykovou obrazovkou ako sú napríklad inteligentné telefóny, tablety alebo inteligentné hodinky. Android vyvíja konzorcium Open Handset Alliance a je sponzorovaný spoločnosťou Google. Rozdelený je na 5 vrstiev.

1.2 Android SDK

Android SDK (Software Development Kit) je kolekcia nástrojov a knižníc pre vývoj Android aplikácií. Obsahuje všetky potrebné nástroje k pohodlnému programovaniu aplikácií – debugger, knižnice, emulátor, dokumentáciu a ďalšie. Google vydáva s každou verziou systému Android aj príslušné SDK. Android SDK je kompatibilné s operačnými systémami Windows, macOS a Linux.

1.3 Android Studio

Android Studio je oficialne integrované vývojové prostredie (IDE) určené pre vývoj Android aplikácií. Založené je na IntelliJ IDEA IDE od spoločnosti JetBrains. Spolu s týmto IDE, Android Studio ponúka množstvo nástrojov pre najproduktívnejší vývoj aplikácií. Napríklad: Gradle build system, emulátor, testovacie nástroje a ďalšie.

1.3.1 Gradle

Gradle je open-source multiplatfomový nástroj na automatizáciu a zostavovania projektov a programov rôznych jazykov. Riadi proces vývoju softvéru pomocou úloh pozostávajúcich z kompilácie, testovania, zverejnenia a ďalších. Podporuje jazyky: Java, Kotlin, Groovy, Scala, C/C++ a Javascript. Gradle je vybudovaný na konceptoch Apache Ant a Apache Maven.

1.4 Aktivity

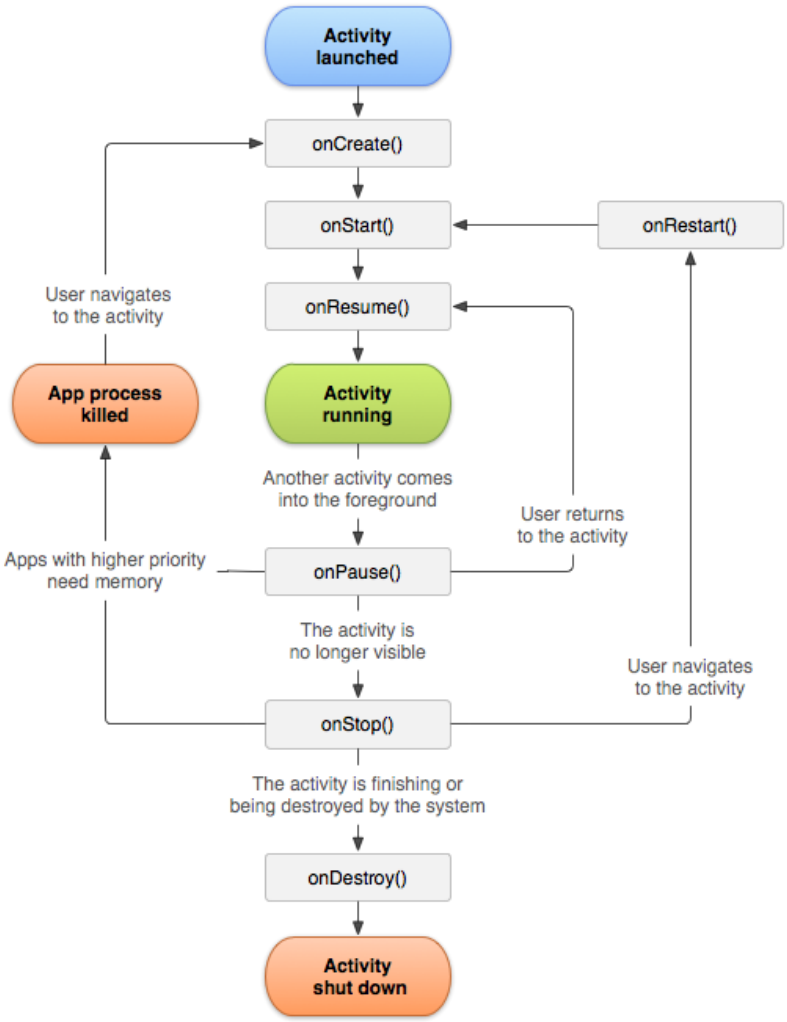
Aktivita patrí k hlavným stavebným konceptom Android aplikácií. Hlavný vstupný bod aplikácie je na rozdiel od väčšiny programov, ktoré sú načítane pomocou main metódy, práve aktivita. Android systém spúšťa v aktivite metódy, ktoré reprezentujú fázy životného cyklu aktivity. Mobilné aplikácie sa nespúšťajú vždy z rovnakého miesta. Aplikácia môže mať viacero aktivít, ktoré môžu sa môžu navzájom spúšťať a ukončovať. Napríklad, ak spustíme aplikáciu telefón z domovskej obrazovky, načíta sa príslušná aktivita, ktorá zobrazuje zoznam nedávne volaných čísel. Toto nemusí byť jediný spôsob ako spustiť túto aplikáciu. Napríklad ak sa nachádzame v mobilnom prehliadači a na webovej stránke, ktorá zobrazuje telefónne číslo, dokážeme sa kliknutím na toto číslo dostať do aplikácie telefónu a uskutočniť hovor na toto číslo, bez potreby manuálne otvoriť aplikáciu z domovskej obrazovky a napísať číslo. Po kliknutí na číslo vo webovom prehliadači, spustí webový prehliadač aktivitu z aplikácie telefón zobrazujúcu klávesnicu a prepošle do tejto aktivity aj dané telefónne číslo.

Aktivita poskytuje obrazovku, na ktorej môžeme vykresľovať UI prvky ako sú napríklad tlačidlo, text, textové pole, obrázky a ďalšie. Jedna aktivita môže zobrazovať napríklad prihlasovací formulár a iná aktivita môže zobrazovať zoznam uložených stromov. Aplikácia väčšinou obsahuje hlavnú aktivitu (MainActivity), ktorá sa zobrazí ako prvá obrazovka po spustení aplikácie z domovskej obrazovky. Aktivita je tvorená z dvoch súborov. Prvý súbor obsahuje triedu, ktorú chápeme ako Aktivita. Táto trieda dedí z triedy s názvom AppCompactActivity. Druhý súbor je vo formáte .xml a obsahuje užívateľské rozhranie.

1.4.1 Životný cyklus aktivity

Od spustenie aktivity, cez prechádzanie medzi ďalšími aktivitami, až po úplné ukončenie aplikácie, inštancia našej aktivity prechádza štádiami jej životného cyklu. Trieda s názvom AppCompactActivity, z ktorej musí každá aktivita dediť nám poskytuje viacero metód. Tieto metódy reprezentujú fázy životného cyklu aktivity, ktoré sú spúšťané podľa toho, v akom štádiu sa aktivita nachádza.

V každej aktivite musíme implementovať metódu onCreate, ktorá je spustená pri vytvorení aktivity systémom. V tejto metóde vykonáme logiku, ktorú chceme spustiť len jeden krát v životnom cykle aktivity. Tiež by sme z tadeto mali zobraziť užívateľské rozhranie, nachádzajúce sa v .xml súbore.



Obr. 1 Diagram životného cyklu aktivity (Android Developers 2020)

1.5 Fragmenty

Fragment chápeme ako časť užívateľského rozhrania, ktorý sa vloží na určené miesto v aktivite. Fragment má vlastný životný cyklus, ale nedokážeme ho používať samostatne bez aktivity alebo bez toho, aby sme ho vložili do iného fragmentu. Fragmenty využívame napríklad pri navigácií v aplikácií. Aktivita obsahuje bočný panel alebo spodný panel s navigáciou a obsahuje miesto v strede obrazovky, na ktoré pomocou navigácie vkladáme príslušný fragment. Napríklad máme 2 položky v navigácií/menu a 2 fragmenty, ktoré chceme zobraziť, keď používateľ klikne na položku v menu. Pri navigácií zobrazíme vždy len jeden fragment, ktorý reprezentuje položku v navigácií. Fragment je tvorený dvoma súbormi. Prvý súbor je trieda, ktorá dedí z triedy Fragment. Do konštruktoru tejto triedy prepošleme odkaz na druhý súbor fragmentu, obsahujúci UI vo formáte .xml.

1.6 Android Jetpack

Jetpack je sada knižníc, vďaka ktorej vývojári píšu menej kódu ako obvykle. Pomáhajú dodržiavať najlepšie praktiky pri vývoji aplikácií. Komplexné veci dokážeme napísať jednoduchšie. Kód napísaný vývojármi pomocou týchto knižníc pracujú správne na veľkom spektre Android verzií. Jetpack rieši problémy, vznikajúce pri spravovaní životného cyklu aktivít v aplikáciách. Dôležité sú pravidelné aktualizácie pre Jetpack, ktoré získava častejšie ako samotná Android platforma. Obsahuje komponenty architektúry, určené pre vývoj moderných, robustných a jednoducho testovateľných aplikácií. Do komponentov architektúry patrí: Data Binding, LiveData, Navigation, ViewModel, Room a ďalšie.

1.7 Kotlin

Kotlin je staticky typovaný programovací jazyk bežiaci nad JVM (Java Virtual Machine). Kotlin je vytvorený na plnú spoluprácu s Javou. Kotlin štandardné knižnice závisia na Java class knižnici. Primárne je určený pre JVM, ale je tiež možné ho skompilovať na Javascript alebo do natívneho kódu iOS aplikácií, ktoré majú spoločnú logiku s Android aplikáciami. Vyvíjaný je spoločnosťou JetBrains. Veľká časť vývojárov používajúcich Javu pre vývoj Android aplikácií prešli na Kotlin, po tom ako Google oznámil, že Kotlin sa stáva preferovaným jazykom pre vývoj Android aplikácií. Používanie Kotlinu poskytuje viacero výhod, napríklad: menej kódu s lepšou čitateľnosťou, Coroutines, rozširujúce funkcie, lambda výrazy, väčšia bezpečnosť v kóde, null bezpečnosť. Kotlin je možné využívať spolu s programovacím jazykom Java bez potreby migrovať celý kód do Kotlinu. Medzi veľké výhody tiež patrí jednoduchosť naučenia, hlavne pre Java vývojárov.

1.7.1 Coroutines

Coroutines umožňujú písať asynchrónny kód. Pri Android aplikáciách nám pomáhajú riešiť dlhotrvajúce úlohy, ako je napríklad komunikácia so serverom cez http požiadavky alebo zápis do lokálnej databáze. Ak by sme tieto akcie vykonávali na hlavnom vlákne mohli by sme ho zablokovať a následne by naša aplikácia mohla spadnúť. Coroutines využívajú viaceré knižnice z Android Jetpacku.

1.8 Dagger Hilt

Hilt je knižnica poskytujúca Dependency Injection (DI) pre Android, redukujúca štandardný kód, ktorý sa tvorí manuálnym tvorením DI. Pri manuálnej tvorbe DI je nutné od vývojára aby každý objekt triedy konštruoval ručne, a taktiež je nutné aby manuálne vytvoril kontajner na ukladanie a znovu použitie objektov/inštancií. Hilt nám poskytuje štandardizovanú cestu ako používať DI v našich projektoch a riadi ich životný cyklus automaticky. Hilt je postavený na populárnej DI knižnici Dagger. Dagger oproti Hilt zjednodušuje potrebnú prípravu DI pred použitím.

1.8.1 Dependency Injection

Dependency Injection (DI) je návrhový vzor, technika, v ktorej objekt príjme iný objekt, na ktorom je závislý – nedokáže bez neho správne fungovať. Tieto objekty sú nazývané dependencies – závislosti. DI je konkrétna technika princípu IoC – Inversion of Control. To znamená, že objekty sú riadené vyšším mechanizmom a samotné objekty získavajú závislosti zvonku a automaticky. Existuje viacero možností ako objekt získa závislosť. V projekte použijeme setter injection – závislosť bude vložená priamo do vlastnosti objektu, a tiež využijeme constructor injection – závislosti budú poskytnuté ako parametre konštruktoru triedy. DI je zodpovedná za vytváranie objektov, zistenie aké triedy vyžaduje daný objekt a poskytnúť tieto objekty. Medzi výhody DI patrí možnosť rozširovať aplikáciu o niečo jednoduchšie, zbytočný kód, ktorý by sme museli písať je zredukovaný a pomôže prípadnému Unit testovaniu. Nevýhodou môže byť fakt, že DI je o niečo komplexnejšia technika na naučenie a na pochopenie.

1.9 Retrofit

Retrofit je HTTP klient pre Android, Javu a Kotlin, poskytujúci typovú bezpečnosť. Poskytuje jednoduchú cestu ako vykonávať HTTP požiadavky na server a získať odpoveď v JSON formáte. Odpoveď z tohto formátu konvertuje na jednoduchý objekt Kotlin triedy, ktorú si sami vytvoríme. K práci s Retrofitom je potrebné vytvoriť Retrofit objekt. Následne máme možnosť rozdeliť naše API požiadavky do viacerých súborov. Napríklad požiadavky na server súvisiace s používateľom, prihlásením a registráciou budú v súbore UserAPI. Iné požiadavky súvisiace s manipuláciou uložených stromov budú v súbore TreesAPI. Každý z týchto súborov obsahuje interface a v ňom definujeme názvy metód. Aby Retrofit vedel akú požiadavku odoslať na

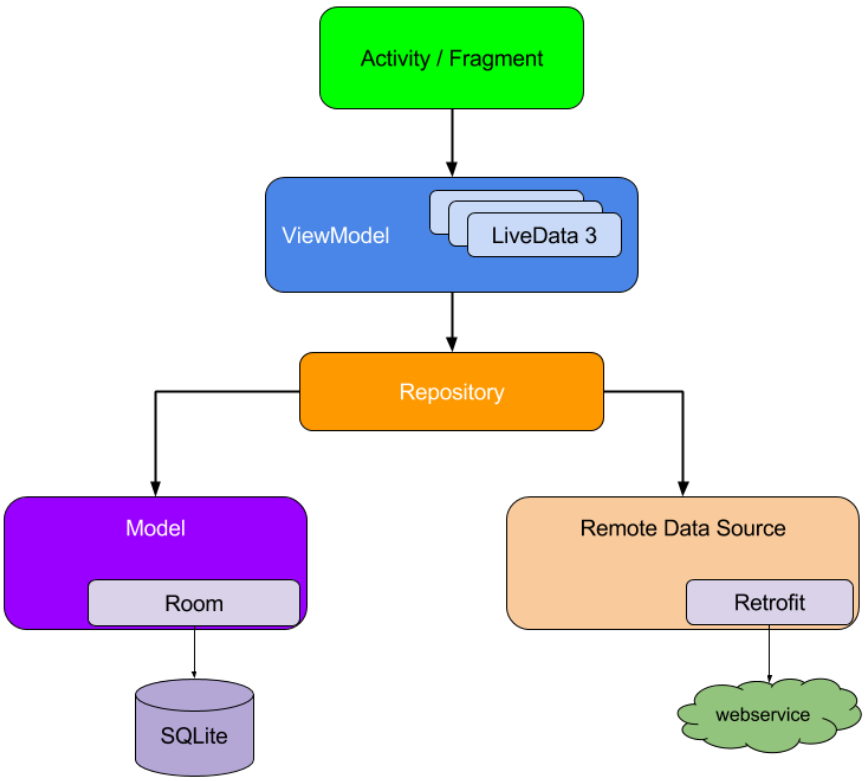
server keď zavoláme metódu, metódam pridáme anotácie. Anotácie určia typ požiadavky, URL adresu, dáta a hlavičky požiadavky. Dáta, ktoré nám server odošle budú priradené objektu, ktorý sme určili návratovým typom metódy. Aby sme boli schopný spúšťať metódy, ktoré sme definovali v tomto súbore, Retrofit za nás vytvorí implementáciu tohto interfacu. Dáta v podobe objektu získame pomocou použitia MutableLiveData, ktoré poskytuje Android Jetpack.

2 Cieľ práce

Cieľom tejto práce je navrhnúť a vypracovať Android natívnu aplikáciu, určenú pre evidenciu stromov mesta Bánovce nad Bebravou. Práca bude riešiť návrh mobilnej natívnej aplikácie, na základne MVVM architektúry. Cieľom je poskytnúť riešenie pre mesto ako evidovať stromy v meste. Mesto získa relevantné dáta o stromoch v meste, ktoré môžu neskôr slúžiť na zlepšenie životného prostredia a zelene v meste. Aplikácia bude určená pre dobrovoľníkov, ktorí prejavia záujem o zber informácií. Taktiež je cieľom vysvetliť vývoj Android aplikácií pomocou pokročilej architektúry a rozdelenia aplikácie na vrstvy. Tiež čitateľovi vysvetli návrhovú techniku Dependency Injection, použitú pri tvorbe.

3 Materiál a metodika

Natívna aplikácia bude postavená na architektúre MVVM a komponentoch architektúry, ktoré nám poskytne Android Jetpack. Architektúra sa bude skladať zo 4 hlavných komponentov: UI (Aktivita alebo Fragment), ViewModel, Repozitár a API.



Obr. 2 Diagram architektúry aplikácie (Android Developers 2020)

Každý komponent je závislí iba od komponentu, ktorý je o úroveň nižšie. Napríklad aktivita je závislá iba od ViewModelu. To znamená, že aktivita nemôže fungovať správne bez ViewModelu. Repozitár môže byť závislí od dvoch komponentov. V našom prípade nebudeme využívať SQL lokálnu databázu v aplikácií ale len API. API bude komunikovať so serverom a server s databázou používateľov a stromov. Pomocou týchto komponentov sme našu aplikáciu rozdelili na tri vrstvy. Prvá vrstva - vrstva používateľského rozhrania zahŕňa aktivity a fragmenty našej aplikácie. Táto vrstva bude má na starosti len vykresľovanie užívateľského rozhrania a nemá priamy prístup k dátam zo servera. Druhá vrstva obsahuje biznis logiku

aplikácie. Sem patrí ViewModel, ktorý nemá možnosť zasahovať priamo do používateľského rozhrania ale je tu napríklad validácia dát. Táto vrstva bude komunikovať s poslednou vrstvou, ktorá je dátový zdroj. Patrí sem repozitár, cez ktorý komunikujeme s API. Rozdelením aplikácie na vrstvy sme docielili oddelenie zodpovednosti jednotlivých komponentov iba na určitú funkcionalitu. Týmto rozdelením sme implementovali najznámejší princíp v programovaní – Separation of Concerns – SoC – Oddelenie zodpovednosti.

3.1 Vytvorenie projektu

Otvoríme si Android Studio a vytvoríme nový projekt. Vo výbere predpripravených šablón si vyberieme „Empty Activity“, ktorá obsahuje iba jednu prázdnu aktivitu. Zvolíme názov aplikácie, programovací jazyk Kotlin a minimálne SDK na API 19. Minimálne SDK určuje aká je potrebná minimálna verzia operačného systému Android k spusteniu aplikácie. API 19 je verzia Android 4.4 KitKat. Staršie verzie nebudú schopné našu aplikáciu spustiť. Pri výbere minimálneho SDK chceme vybrať optimálnu verziu – nechceme najstaršie SDK, pretože by sme nemohli využívať moderné funkcie v našej aplikácií a tiež nechceme najnovšie, pretože by našu aplikáciu vedelo spustiť len málo zariadení. Ako posledné zvolíme názov balíka. Názov by mal byť opačný od našej domény, teda ak máme doménu „bntrees.dominikjezik.sk“ názov balíka je „sk.dominikjezik.bntrees“. Android Studio nám vygeneruje projekt. V ľavej časti programu máme zobrazenú projektovú štruktúru. AndroidManifest obsahuje základnú konfiguráciu aplikácie. Registrujeme tu aktivity, ktoré v projekte vytvoríme, názov aplikácie, ikony, tému a ďalšie. V aplikácií budeme tiež potrebovať mať prístup k internetu, k polohe zariadenia a tiež k zápisu a k čítaniu z úložiska zariadenia. Aby sme mohli využívať permissie, potrebujeme ich v tomto súbore zaregistrovať.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="sk.dominikjezik.bntrees">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <application
        android:requestLegacyExternalStorage="true"
        android:name=".BaseApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher_main"
        android:label="Stromčeky BN"
        android:roundIcon="@mipmap/ic_launcher_main_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.BnTrees.NoActionBar"
        android:usesCleartextTraffic="true"
        tools:targetApi="m">
```

Obr. 3 Kód súboru AndroidManifest.xml (Dominik Ježík 2020)

3.1.1 Registrácia závislostí

V Gradle Scripts, v súboroch build.gradle zaregistrujeme naše závislosti, ktoré budeme v projekte používať. V prvom súbore build.gradle pridáme v časti dependencies len jeden riadok, kvôli balíku Dagger Hilt, ktorý bude na pozadí generovať ďalšie potrebné súbory. V druhom súbore build.gradle v časti plugins pridáme „kotlin-kapt“ a „dagger.hilt.android.plugin“, ktoré sú tiež potrebné pre správnu činnosť Dagger Hilt. V android časti povolíme funkciu View Binding. Na konci súboru zaregistrujeme všetky potrebné závislosti ako sú napríklad: Retrofit, Dagger Hilt, Google Maps, Navigation Components, Coroutines a ďalšie. Sú to balíky, ktoré budeme využívať v našej aplikácií. Po úpravách v týchto súboroch musíme zosynchronizovať Gradle. Vždy keď tu spravíme nejakú zmenu zobrazí sa nám tlačidlo na synchronizáciu v pravom hornom rohu. Pri synchronizácií sa do projektu stiahnu chýbajúce balíky.

Aplikáciu spustíme buď pomocou nášho smartphonu pripojeného do počítača alebo si cez AVD manažéra zabudovaného do Android Studia vytvoríme virtuálny zariadenie, na ktorom môžeme aplikáciu testovať. Aplikáciu následne spustíme pomocou tlačidla na hornej lište.

3.2 Príprava architektúry

Pre našu trojvrstvovú architektúru si musíme v projekte vytvoriť priečinky, do ktorých budeme jednotlivé komponenty ukladať. Celú štruktúru budeme tvoriť v priečinku „java/sk.dominikjezik.bntreen“. Vytvoríme priečinok „ui“ a v ňom „viewmodels“ a „fragments“. V „ui“ budú tiež všetky aktivity. MainActivity vytvorenú Android Studiom presunieme do tohto priečinku. Ďalej vytvoríme v „java/sk.dominikjezik.bntreen“ priečinky „repositories“, „models“ a „api“. Podľa názvu vieme čo budeme do tých priečinkov ukladať.

3.2.1 Príprava Dependency Injection

Aby sme mohli v našej aplikácií používať Dependency Injetion, do Gradle sme pridali balík Dagger Hilt, ktorý nám poskytne jej implementáciu. V „java/sk.dominikjezik.bntreen“ vytvoríme triedu BaseApplication, ktorá dedí zo vstavanej triedy Application. Nad triedu pridáme Hilt anotáciu aby vedel, kde sa nachádza hlavná trieda našej aplikácie. Túto triedu musíme ešte zaregistrovať v AndroidManifest.

BaseApplication.kt

```
1  package sk.dominikjezik.bntrees
2
3  import android.app.Application
4  import dagger.hilt.android.HiltAndroidApp
5
6  @HiltAndroidApp
7  class BaseApplication: Application() {
8  }
```

Obr. 4 Kód triedy BaseApplication (Dominik Ježík 2020)

Ďalej vytvoríme priečinok „di“ určený pre Dagger Hilt moduly. My vytvoríme len jeden s názvom AppModule. Súbor nebude obsahovať klasickú triedu ale objekt, ktorý je špeciálny pre jazyk Kotlin. Funguje podobne ako trieda s tým rozdielom, že nedokážeme vytvoriť jej inštanciu. Miesto toho funguje ako „singleton“, teda vždy existuje len jediná inštancia. Aby Hilt dokázal nájsť tento modul, objektu nad názov pridáme anotáciu „@Module“ a „@InstallIn(ApplicationComponent::class)“. Súbor je určený na registráciu závislosti pre naše triedy. Ak napríklad potrebujeme používať inštanciu nejakej triedy na viacerých miestach, museli by sme vždy vytvárať novú inštanciu. Namiesto toho nám Dagger Hilt automaticky vloží potrebné závislosti do našich tried.

3.2.2 Príprava http klienta

Pomocou Retrofit balíka budeme vykonávať http požiadavky na náš server. K používaniu balíka musíme vytvoriť inštanciu tejto triedy. Práve pre vytvorenie inštancie použijeme Dependency Injection a náš modul. V module vytvoríme metódu provideRetrofit. Nad metódu pridáme anotácie „@Singleton“ a „@Provides“. Tieto anotácie budeme v tomto module pridávať na každú metódu. Singleton anotácia znamená, že keď si vyžiadame viac krát inštanciu Retrofitu, dostaneme vždy len jednu. Metóda vráti inštanciu Retrofitu. Retrofit má závilosť na OkHttpClient a je potrebná k zostrojeniu Retrofit inštancie. Preto do parametrov metódy pridáme parameter, kde OkHttpClient získame. Pridáme druhú metódu, v ktorej zostrojíme OkHttpClient. K zostrojeniu nepotrebujeme ďalšie závislosti preto metóda nemá žiadne parametre. Ak by sme si teraz vyžiadali Retrofit inštanciu, Dagger Hilt nám ju poskytne, pretože vie ako ju zostrojiť. BASE_API_URL nahradíme adresou na náš server.

Obr. 5 Kód objektu AppModule s dvoma metódami (Dominik Ježík 2020)

3.3 Rozhrania API

Aplikácia bude obsahovať dva repozitáre a dve API. V priečinku api vytvoríme interface s názvom UsersAPI. Interface – rozhranie nie je pre Kotlin novinka, nachádza sa vo väčšine objektovo orientovaných jazykoch. Interface je abstraktný typ, obsahujúci názvy metód, bez implementácie. Z rozhrania nemôžeme vytvoriť objekt, ale môžeme vytvoriť triedu, ktorá naše rozhranie implementuje. V našom prípade, v tomto súbore UsersAPI, definujeme názvy metód. Jednotlivé metódy budú prezentovať http požiadavky na server. Prvá metóda s názvom submitLogin príjme 3 parametre – email, password a deviceName. Chceme, aby tieto parametre boli vložené do požiadavky, preto pred každý parameter pridáme anotáciu @Query(názov). Anotácia zabezpečí vloženie parametra do požiadavky. Nad názov metódy pridáme ďalšie dve anotácie. @Headers("Accept: application/json") pridá do požiadavky hlavičku, ktorá hovorí o tom, že ako odpoveď očakávame dáta vo formáte json. @POST("login") určuje cestu na odoslanie a typ požiadavky na POST. Pred metódu musíme pridať slovo „suspend“, ktoré zabezpečí, že metódu nebude možné spustiť mimo Coroutine alebo mimo inej suspend metódy. To znamená, že požiadavky na server budeme spúšťať na inom jadre ako je hlavné jadro, na ktorom beží používateľské rozhranie. Keby požiadavku spustíme na hlavnom jadre, zablokujeme ho, pretože http požiadavke trvá nejaký čas kým získa odpoveď a aplikácia by mohla spadnúť, ak by požiadavka trvala o niečo dlhšie. Ďalšie metódy v tomto rozhraní sú

odosielať registračný formulár, odhlasovací formulár, aktualizovanie profilu a získanie profilu používateľa.

```
9   interface UsersAPI {
10
11      @Headers( ...value: "Accept: application/json")
12      @POST( value: "login")
13      suspend fun submitLogin(
14          @Query( value: "email")
15          email: String,
16          @Query( value: "password")
17          password: String,
18          @Query( value: "device_name")
19          deviceName: String = "TEST_DEVICE"
20      ): Response<AuthResponse>
```

Obr. 6 Interface a suspend funkcia na prihlásenie užívateľa (Dominik Ježík 2020)

Druhý súbor bude s názvom TreesAPI, bude tak isto obsahovať interface a anotácie na priradenie parametrov do požiadavky. V tomto súbore budeme vykonávať všetky manipulácie so stromami. Metódy súboru budú: získavať všetky stromy, získavať iba stromy, ktoré pridal prihlásený používateľ, pridávať stromy, vymazávať stromy a aktualizovať informácie o stromoch. V každej metóde musíme okrem prípadných ďalších parametrov získať aj api kľúč. Kľúč autorizuje užívateľa na servery a bez kľúča nebudeme schopný vykonať žiadne akcie so stromami. Pred parameter apiKey nepridáme @Query anotáciu ako pri obyčajných parametroch, ale @Header("Authorization"). Táto anotácia zabezpečí priradenie kľúča do hlavičky požiadavky.

3.3.1 Implementácia rozhraní

Implementáciu nemusíme vytvoriť my sami, ale postará sa o ňu Retrofit. V AppModule, kde riešime závislosti, pridáme metódu provideUsersAPI. Ako parameter budeme požadovať inštanciu Retrofitu. Metóda zavolá metódu create na Retrofite a v nej prepošleme referenciu na naše rozhranie. To isté spravíme aj pri TreesAPI.

```
@Singleton
@Provides
fun provideUsersAPI(retrofit: Retrofit) = retrofit.create(UsersAPI::class.java)

@Singleton
@Provides
fun provideTreesAPI(retrofit: Retrofit) = retrofit.create(TreesAPI::class.java)
```

Obr. 7 Metódy z AppModule poskytujúce implementáciu rozhraní (Dominik Ježík 2020)

Teraz, keď si vyžiadame v nejakom súbore implementáciu jedného z API, Hilt nám ju do tohto súboru sám vloží a sme schopný ju používať, pretože Retrofit nám poskytol implementáciu.

3.5 Ukladanie API kľúča

Pri autentifikácií získame zo serveru API kľuč. Ak chceme aby bol používateľ prihlásený v aplikácií, musíme mať prístup ku kľúču aj po zavretí aplikácií. Musíme ho uložiť do zariadenia, aby sme sa nemuseli prihlasovať pri opätovnom otvorení aplikácie. Na spravovanie autentifikácie a API kľúčov v zariadení si vytvoríme obyčajnú triedu s názvom AuthManager. Uložená je v novom priečinku „util“. K ukladaniu menšieho počtu informácií súvisiacich s nastavením aplikácie využijeme SharedPreferences. Ide o ukladací priestor založený na princípe key-value. Aby sme mohli využívať toto úložisko, potrebujeme získať objekt SharedPrefences, ktorý je možný získať len z kontextu aplikácie. Preto musíme v AppModule pridať metódu na poskytnutie tohto objektu. V metóde ako parameter získame context, pred ktorý pridáme anotáciu @ApplicationContext. Metóda zavolá na contexte getSharedPreferences, s parametrami názvu úložiska a nastavením úložiska na privátne. Teraz si môžeme SharedPreferences vyžiadať v konštruktore. Budeme chcieť aby AuthManager bol cez Hilt vložený do repozitárov, preto si ešte v AppModule vytvoríme metódu na poskytnutie AuthManagera. V parametroch získame SharedPreferences a tento objekt vložíme do konštruktora.

Vrátime sa späť do AuthManager triedy a pridáme atribút na poskytnutie editoru v SharedPreferences. Funkcia setApiKey, získa kľuč a cez editor kľúč zapíše do úložiska. Metóda getApiKey získa cez editor kľúč z úložiska. Podobne pridáme funkcie na skontrolovanie či je užívateľ prihlásený – skontroluje či je v úložisku uložený kľúč, na uloženie užívateľovho mena, id a emailu.

3.6 Repozitáre a SharedPreferences

Ďalším komponentom našej aplikácie je repozitár. Repozitár bude využívať naše API, ktoré sme si vytvorili a AuthMangera. Úlohou repozitára je poskytovať dáta zo serveru ViewModelu, ktorý vytvoríme neskôr. V priečinku repositories vytvoríme triedu UsersRepository. Pred konštruktor triedy napíšeme anotáciu @Inject a pridáme dva parametre. Anotáciu sme pridali kvôli tomu, aby nám parametre/závislosti triedy poskytol Hilt. Prvý parameter, s názvom usersAPI, je typu UsersAPI a druhý je AuthManager. Hilt nám teda poskytne implementáciu z AppModule. Teraz môžeme vytvoriť metódy s rovnakými názvami ako v API, ktoré budú volať metódy z API. Metódy majú znova ako v API časti pred deklaráciou slovo „suspend“. Metódy budeme volať vo ViewModely, konkrétne pomocou Coroutines inak by sme neboli schopný ich spúšťať. Podobne vytvoríme aj TreesRepository, ktorý získa TreesAPI a AuthManagera. V metódach tohto repozitára budeme rovnako vkladať API kľuč, ale teraz do každej metódy, pretože manipulovať so stromami môže len autorizovaný užívateľ.

V našom prípade bolo hlavnou úlohou repozitárov vkladanie API kľúča pomocou Auth manažéra. Nemusíme teda vo ViewModely manuálne vkladať kľuč, keď budeme chcieť získať dáta zo servera.

```
class UsersRepository @Inject constructor(
    private val usersAPI: UsersAPI,
    private val authManager: AuthManager
) {

    suspend fun submitLogin(email: String, password: String)
        = usersAPI.submitLogin(email, password)

    suspend fun submitRegister(name: String, email: String, password: String)
        = usersAPI.submitRegister(name, email, password)

    suspend fun submitLogout() = usersAPI.submitLogout(authManager.getApiKey())

    suspend fun updateUserProfile(name: String, email: String, password: String?)
        = usersAPI.updateUserProfile(authManager.getApiKey(), name, email, password)

    suspend fun getUserProfile() = usersAPI.getUserProfile(authManager.getApiKey())

}
```

Obr. 8 Kód repozitára užívateľov (Dominik Ježík 2020)

3.7 Autentifikácia užívateľa

Zobrazovanie prihlasovacieho a registračného formulára je úlohou novej aktivity AuthActivity.kt v priečinku „ui“. Dizajn aktivity je v súbore activity_auth.xml a obsahuje jediný element – FrameLayout, ktorému nastavíme id. Auth aktivita bude slúžiť len ako držiteľ fragmentov, v tomto prípade login fragmentu a register fragmentu. Na pozícií FrameLayoutu zaberajúceho maximálnu možnú výšku a šírku budeme zobrazovať tieto dva fragmenty. V priečinku ui/fragments sa bude nachádzať LoginFragment.kt a RegisterFragment.kt. V súboroch sú triedy LoginFragment a RegisterFragment a dedia z triedy Frament. V konštruktore ako parameter prepošleme odkaz na dizajn fragmentov. Dizajny fragmentov sú taktiež v res/layout pod názvami fragment_login.xml a fragment_register.xml. Na určenie pozície elementov využijeme ConstraintLayout.

3.7.1 AuthViewModel

Po stlačení tlačidla na odoslanie formulára chceme logiku odoslania API požiadavky na server vykonať mimo našej aktivity alebo fragmentov. Spojenie medzi našou aktivitou a repozitárom bude vykonávať AuthViewModel. Tento ViewModel je závislí od UsersRepository, preto v konštruktore prijmeme tento repozitár a tiež AuthManagera. Keď chceme aby nám Hilt vložil závislosti do ViewModelu, musíme použiť miesto klasickej anotácie @Inject, anotáciu @ViewModelInject.

Vo vlastnosti response budeme držať odpoveď zo servera. Vlastnosť je typu MutableLiveData. Ide o generickú triedu a preto určíme typ s akým má táto trieda pracovať – na AuthResponse. MutableLiveData je trieda z knižnice Android Jetpack. LiveData triedy dodržiavajú observer pattern, teda umožňujú iným triedam pozorovať tento objekt a v prípade, že sa hodnota LiveData zmení, pozorovateľ bude upozornený a získa novú hodnotu. Toto chovanie využijeme v našom prípade.

Pridáme metódu na odoslanie prihlasovacieho formulára, ktorý bude príjmať email a heslo. Keďže ideme volať metódu označenú slovom „suspend“ z repozitára na odoslanie formulára, musíme ju spustiť pomocou Coroutines. Použijeme viewmodel scope a v bloku launch môžeme vykonať asynchrónne akcie. Najskôr nastavíme hodnotu vo vlastnosti LiveData response na stav loading – načítavanie. V try-catch bloku sa pokúsime získať pomocou repozitára dáta zo serveru. Ak nám server vráti odpoveď so statusom 200, môžeme dáta zo serveru uložiť do LiveData objektu. Využijeme na to pomocnú metódu, ktorá rozhodne podľa dát zo servera či

sa podarilo úspešne prihlásiť alebo používateľ zadal zlý email a heslo. Zároveň nastaví pomocou AuthManagera kľúč a dáta o užívateľovi do úložiska zariadenia, ak sa prihlásenie podarilo. V prípade serverovej chyby sa spustí catch blok a do LiveData uložíme chybnú hlášku. Podobne budeme postupovať pri registračnom formulári.

```
fun submitLogin(email: String, password: String) = viewModelScope.launch {
    response.postValue(Resource.Loading())
    try {
        val res = usersRepository.submitLogin(email, password)
        response.postValue(handleLoginResponse(res))
    } catch (exception: SocketTimeoutException) {
        response.postValue(Resource.Error( message: "Server error"))
    }
}
```

Obr. 9 Spustenie Coroutine vo viewModelScope (Dominik Ježík 2020)

3.7.2 AuthActivity

Do AuthActivity pridáme anotáciu @AndroidEntryPoint a Hilt nám do aktivity vloží inštanciu AuthViewModelu. AuthActivity spravuje zobrazenie buď login fragmentu alebo register fragmentu. Preto v dvoch vlastnostiach sú inštancie oboch tried a pomocou dvoch metód budeme vykonávať zámenu fragmentov. Pri spustení aktivity chceme vidieť ako prvý login fragment.

```
supportFragmentManager.beginTransaction().apply {
    replace(R.id.flFragment, loginFragment)
    commit()
}
```

Obr. 10 Kód na zmenu fragmentu (Dominik Ježík 2020)

3.7.3 LoginFragment a RegisterFragment

V súbore fragment_login.xml, do vnútra layoutu vložíme elementy tlačidlo na odoslanie formuláru, dve vstupné polia na email a heslo a indikátor čakania na odpoveď zo servera. Každému elementu priradíme id pre identifikáciu v triede fragmentu. Na prepínanie sa medzi fragmentami využijeme text s nápisom „Vytvoriť účet“, slúžiaci ako link. Vo fragmente

prepíšeme metódu onViewCreated, volanú automaticky v momente, keď sa zobrazí daný fragment.

V triede LoginFragment chceme pristupovať k elementom z layoutu, a preto si nastavíme view binding. View binding je možnosť akou môžeme pristupovať k elementom z dizajnu. Viewmodel získame z AuthAktivity. Na prechod z Login fragmentu na registračný nastavíme click listener na text. Vnútri tela pristúpime k aktivite a zavoláme metódu na zobrazenie registračného fragmentu. Ak teraz klikneme na text fragment sa zmení na registračný. Podobne nastavíme aj kliknutie na prihlasovacie tlačidlo. Po kliknutí pristúpime k ViewModelu a zavoláme metódu na odoslanie formulára. Z bindingu prepošleme hodnoty v textových poliach.

Ak by sme teraz vyskúšali zadať email a heslo a odoslať formulár, server by dáta prial validoval a odoslal spätnú väzbu späť do ViewModelu. Preto musíme v našom fragmente nastaviť pozorovateľa, ktorý bude pozorovať LiveData objekt z ViewModelu. Na vlastnosti LiveData zavoláme metódu observe, a vložíme blok kódu, ktorý sa bude volať ak sa dáta zmenia. Využijeme when blok, ktorý je podobný bloku switch z iných jazykov. Ak je hodnota v LiveData loading, zobrazíme indikátor čakania na odpoveď zo servera. Ak je hodnota error, skryjeme indikátor čakania a zobrazíme chybu. V poslednom prípade sa autentifikácia podarila, preto skryjeme indikátor čakania a zavoláme novú metódu v AuthAktivity.

```
viewModel.response.observe(viewLifecycleOwner, Observer {res ->
    when(res) {
        is Resource.Success -> {
            binding.progressBar.visibility = View.INVISIBLE
            (requireActivity() as AuthActivity).openMainActivity()
        }
        is Resource.Error -> {
            binding.progressBar.visibility = View.INVISIBLE
            res.message?.let {message ->
                Snackbar.make(view, message, Snackbar.LENGTH_LONG).show()
            }
        }
        is Resource.Loading -> {
            binding.progressBar.visibility = View.VISIBLE
        }
    }
})
```

Obr. 11 Pozorovateľ LiveData vlastnosti z ViewModelu (Dominik Ježík 2020)

Metóda zabezpečí otvorenie MainActivity a zavretie AuthAktivity. Využijeme na to Intent triedu. V MainActivity vytvoríme ďalšie časti aplikácie. RegisterFragment vytvoríme veľmi podobne, s rozdielom, že v prípade chyby zobrazíme validačné chyby pri poliach, obsahujúcich chybu.

```
fun openMainActivity() {
    Intent( packageContext: this, MainActivity::class.java).also {
        startActivity(it)
        this.finish()
    }
}
```

Obr. 12 Metóda na otvorenie MainActivity cez Intent (Dominik Ježík 2020)

3.8 MainActivity

Po spustení aplikácie máme už nastavenú ako prvú aktivitu na spustenie MainActivity. Táto aktivita je aktuálne prázdna. V našom prípade budeme chcieť aby sa po spustení tejto aktivity

skontrolovalo úložisko a aby sa zistilo, či je užívateľ prihlásený. Ak nie je prihlásený, zobrazíme AuthActivity aby sa mohol prihlásiť. Túto kontrolu spravíme jednoducho pomocou AuthManagera. Ak nám metóda z AuthManagera vráti false, spustíme AuthActivity cez Intent a ukončíme MainActivity.

MainActvity bude mať v sebe miesto na zobrazenie fragmentov – mapa stromčekov, zoznam stromčekov pridaných užívateľom a profil používateľa. Na navigáciu medzi týmito fragmentami využijeme Navigation Drawer – ide o vysúvací bočný panel s odkazmi na fragmenty. Dizajn v xml súbore je tvorený DrawerLayoutom – ktorý je pre tento panel určený. NavigationView je konkrétny panel, AppBarLayout a Toolbar tvoria vrchný panel s tlačidlom na otvorenie vysúvacieho panelu a nadpis fragmentu. FrameLayout slúži ako miesto na vloženie fragmentu.

3.8.1 Navigácia a menu

Navigáciu riešime pomocou Jetpack knižnice a NavigationControlleru. Takáto navigácia vyžaduje vytvorenie navigačné grafu. Graf vytvoríme v res/navigation priečinku a nazveme ho nav_graph.xml. V dizajn móde pridáme naše 3 fragmenty, ktoré vytvoríme.

Položky menu sú v samostatnom súbore v res/menu/nav_menu.xml. Id každej položky sa musí zhodovať s názvom fragmentu. Položkám nastavíme atribúty title – nadpis a icon – ikona položky. Pridáme aj štvrtú položku, ktorá nebude odkazovať na žiadny fragment ale po kliknutí sa budeme chcieť z aplikácie odhlásiť. Odkaz na menu priradíme do atribútu menu v komponente NavigationView a navigačný graf na fragment vnútri FrameLayoutu.

V MainActivity teraz musíme napísať obslužný kód, aby navigácia fungovala. Príkazom setSupportActionBar nastavíme vrchný panel na zobrazovanie názvov otvorených fragmentov. Získame NavController z fragmentu, do ktorého sme priradili navigačný graf a pomocou setupActionBarWithNavController navigáciu dokončíme.

3.8.2 Odhlásenie sa z aplikácie

Po kliknutí na položku odhlásenia sa nezobrazí žiadny fragment. Pri kliknutí chceme zobraziť dialógové okno s otázkou, či sa chce používateľ naozaj odhlásiť. Nastavíme setOnMenuItemClickListener na túto položku v menu. Po kliknutí spustíme pomocnú metódu logout. V metóde zostrojíme dialógové okno a zobrazíme ho. Po kliknutí na možnosť „Nie“ sa

dialóg skryje a užívateľ ostane prihlásený a na možnosť „Áno“ užívateľa odhlásime a zobrazíme AuthActivity.

```
val logoutDialog = AlertDialog.Builder( context: this)
    .setTitle("Odhlásiť sa")
    .setMessage("Naozaj sa chcete odhlásiť z aplikácie?")
    .setPositiveButton("Áno") { _, _ -> viewModel.submitLogout()}
    .setNegativeButton("Nie") { _, _ ->  }
    .create()

logoutDialog.show()
```

Obr. 13 Konštruovanie AlertDialog a jeho zobrazenie (Dominik Ježík 2020)

3.9 Google mapa stromov

Mapu na zobrazenie stromov a polohy používateľa nám poskytne služba Google Maps SDK for Android. V Gradle závislostiach aplikácie ju už máme pridanú teraz ju musíme sprevádzkovať. Mapu budeme zobrazovať vo fragmente MapTreesFragment. Dizajn časť bude obsahovať element fragment. Tomuto elementu nastavíme atribút name na SupportMapFragment. Na toto miesto sa nám vloží mapa. Výšku a šírku elementu nastavíme na „match_parent“, to znamená, že element sa natiahne na celú možnú výšku a šírku. V pravom dolnom rohu zobrazíme dve plávajúce tlačidlá – jedno pre pridanie stromu a druhé pre nacentrovanie mapy na používateľovu polohu.

Aby sme mapu mohli používať musíme nastaviť viacero vecí. Ako prvú vec musíme nastaviť v AndroidManifest.xml Google API kľuč. Kľuč si musíme vygenerovať pomocou Google Cloud Platformy. Prejdeme na ich stránku do konzoly a založíme nový projekt. Povolíme Maps SDK for Android a vygenerujeme klúč. Kľuč pridáme do projektu cez element meta-data.

V MapTreesFragment triede teraz inicializujeme mapu. Prepíšeme metódu onActivityCreated a spustíme v nej vlastnú metódu. V nej nájdeme pomocou id fragment, na ktorom chceme mapu zobraziť. Mapu zobrazíme a uložíme do vlastnosti triedy. Po inicializovaní sa pokúsime zobraziť používateľovu polohu.

3.9.1 Povolenia pre získanie polohy zariadenia

Aby sme mohli správne zobraziť používateľovu polohu, musí našej aplikácií povoliť prístup k polohe zariadenia. Po inicializovaní mapy spustíme pomocnú metódu checkOrRequireLocationPermissions. Návratová hodnota metódy je Boolean – vráti true, ak aplikácia už povolenie má. V tomto prípade nacentrujeme mapu na používateľovu polohu. V spravovaniu povolenia použijeme balík EasyPermissions. Tento balík sa bude starať o zobrazenie dialógového okna, v ktorom sa pokúsime získať povolenie na získanie polohy od používateľa. Ak nám používateľ dá povolenie, získame FusedLocationClienta, ktorý nám poskytne polohu, keď si ju vyžiadame. Tohto klienta uložíme do vlastnosti triedy.

```
private fun checkOrRequireLocationPermissions(): Boolean {
    if(PermissionsUtility.hasLocationPermissions(requireContext())) {
        if(!this::fusedLocationClient.isInitialized) {
            fusedLocationClient =
                LocationServices.getFusedLocationProviderClient(requireActivity())
            displayUserPosition()
        }
        return true
    }
    PermissionsUtility.requireLocationPermissions( fragment: this)
    return false
}
```

Obr. 14 Metóda na získanie povolenia pre získanie polohy zariadenia (Dominik Ježík 2020)

3.9.2 Plávajúce tlačidlá

Po kliknutí na prvé plávajúce tlačidlo chceme mapu centrovať na používateľovu polohu. Po kliknutí skontrolujeme povolenia k prístupu k polohe a potom pomocou FusedLocationClienta získame poslednú lokáciu zariadenia. Mapu nastavíme na zobrazenie tejto lokácie pomocou animácie. Maps SDK nám na animáciu poskytuje metódu animateCamera. Druhé tlačidlo bude slúžiť na pridanie nového stromu a nastavíme ho neskôr.

3.9.3 Zobrazenie pridaných stromov na mape

Na mape budeme chcieť zobraziť všetky stromy, ktoré už máme v databáze na servery pridané. Náš fragment s mapou spojíme s repozitárom stromov pomocou nového ViewModelu. V konštruktore prijmeme TreesRepository. Vlastnosť registeredTrees bude typu

MutableLiveData a uložíme tu odpoveď zo serveru. Metóda getAllTrees získa pomocou repozitára všetky stromy a uloží ich do našej vlastnosti. Vo fragmente získame objekt tohto nového ViewModelu a potom ako sa mapa inicializuje zavoláme na ViewModely metódu getAllTrees. Teraz nastavíme sledovanie vlastnosti a ak sa nám zo serveru vrátia uložené stromy zobrazíme ich na mape. Prejdeme listom stromov a pre každý strom vytvoríme nový Marker. Rozlišujeme dva druhy stromov – listnatý a ihličnatý. Oba druhy budú mať vlastnú ikonu. Na objekte mapy zavoláme addMarker a nastavíme pozíciu a ikonu.

```
private fun displayTreeMarkers(trees: List<Tree>) {
    if(!this::googleMap.isInitialized)
        return

    trees.forEach { tree ->
        val treeLatLng = LatLng(tree.lat, tree.lng)
        val icon = BitmapDescriptorFactory.fromResource(
            if(tree.type_listnaty) R.drawable.listnaty_marker
            else R.drawable.ihlicnaty_marker
        )

        googleMap.addMarker(MarkerOptions()
            .position(treeLatLng)
            .icon(icon))
    }
}
```

Obr. 14 Metóda na pridanie stromov na mapu (Dominik Ježík 2020)

3.10 Zoznam užívateľových stromov

Úlohou ďalšieho fragmentu je zobraziť zoznam stromov pridaných prihláseným užívateľom. Vo fragmente nastavíme binding ako pri predchádzajúcich fragmentoch. Do dizajnu pridáme ProgressBar, ImageView, TextView a RecyclerView. ProgressBar sa zobrazí keď aplikácia čaká na odozvu zo serveru. ImageView a TextView oznámia užívateľovi, že ešte nepridal žiadne stromy. RecyclerView sa zobrazí, ak užívateľ pridal nejaký strom. V druhom súbore item_tree.xml vytvoríme dizajn jednej položky zoznamu.

3.10.1 RecyclerView a Adaptér

RecyclerView je element určený na zobrazenie zoznamu položiek. Položky môžeme doň pridať pomocou adaptér triedy. Preto vytvoríme triedu TreesAdapter, ktorá bude dediť z generickej triedy RecyclerView.Adapter. Vnútri triedy sa musí nachádzať „inner“ trieda. Slúži na určenie šablóny položky. TreesAdapter musí implementovať 3 metódy. Prvá je určená na vrátenie počtu položiek v zozname. Zoznam bude vo vlastnosti differ. Do differu pridáme differCallback určený na rozlišovanie jednotlivých položiek. Druhú metódu prepíšeme onCreateViewHolder. V tejto metóde určíme odkaz na dizajn položky. V poslednej metóde onBindViewHolder nájdeme aktuálnu položku zoznamu a v šablóne položky zobrazíme informácie o strome.

Vo fragmente adaptér sprevádzkujeme. Vytvoríme objekt triedy TreesAdapter a priradíme ho do RecyclerVieweru. Tiež priradíme layout manažéra na LinearLayoutManager.

3.10.2 Získanie užívateľových stromov

Pre získanie stromov a naplnenie RecyclerVieweru vytvoríme ViewModel rovnako, ako pri mape stromov. Vo fragmente nastavíme pozorovateľa. Ak získame zo serveru stromy, pridáme ich pomocou adaptéru do differu a skryjeme ProgressBar. Ak nastala chyba skryjeme ProgressBar a RecyclerView a vypíšeme chybu. Na koniec v prípade, že sa zo serveru vráti prázdny list, zobrazíme text a obrázok.

3.11 Pridanie stromu

Pre pridanie stromu vytvoríme aktivitu s názvom AddTreeActivity. V dizajne sa nachádzajú vstupné polia pre zadanie informácií o strome. Tiež je tu mapa, s ktorou sa nebude dať hýbať a bude zobrazovať iba jeden bod.

V MapTreesFragment nastavíme na tlačidlo pridať spustenie tejto novej aktivity. Chceme do nej preposlať zemepisnú šírku a výšku miesta kde sa nachádzame. Dáta medzi aktivitami a fragmentami preposielame pomocou bundle. Vytvoríme nový bundle z týchto dvoch údajov a spustíme novú aktivitu, do ktorej bundle prepošleme.

Bundle získame v aktivite pomocou intent objektu. Metódou getDoubleExtra určíme dátový typ a údaje uložíme do vlastnosti triedy. Inicializujeme mapu rovnako ako

v MapTreesFragmente. V nastaveniach mapy zablokujeme možnosť s mapou hýbať. Kameru mapy nastavíme na preposlanú zemepisnú šírku a výšku a pridáme značku na toto miesto.

Pre túto aktivitu vytvoríme znova ViewModel, ktorý funguje veľmi podobne ako predchádzajúce. Pozorovateľ v aktivite, ktorý bude pozorovať vlastnosť vo ViewModely, v prípade úspešného uloženia stromu zobrazí text o úspešnom vytvorení. V prípade chyby zobrazí validačné chyby nad konkrétnym chybným vstupným polom.

3.11.1 Odosielanie fotografií stromu

K informáciám o stromu budeme chcieť preposlať aj fotografie stromu. Preto do dizajnu aktivity pridáme tlačidlo na získanie fotografií. Po stlačení skontrolujeme povolenie na čítanie úložiska zariadenia. Ak nám to používateľ povolil môžeme vytvoriť nový Intent. Typ nastavíme na „image/*“. Intent spustime príkazom startActivityForResult. Ak užívateľ vyberie fotografiu, máme možnosť ju získať v prepísanej metóde v aktivite s názvom onActivityResult. V tejto metóde získame adresu k zvolenému obrázku a file objekt. Pomocou adresy zobrazíme vybraný obrázok a file objekt prepošleme do ViewModelu. Pri odoslaní formuláru odošleme aj obrázok ak bol vybraný používateľom.

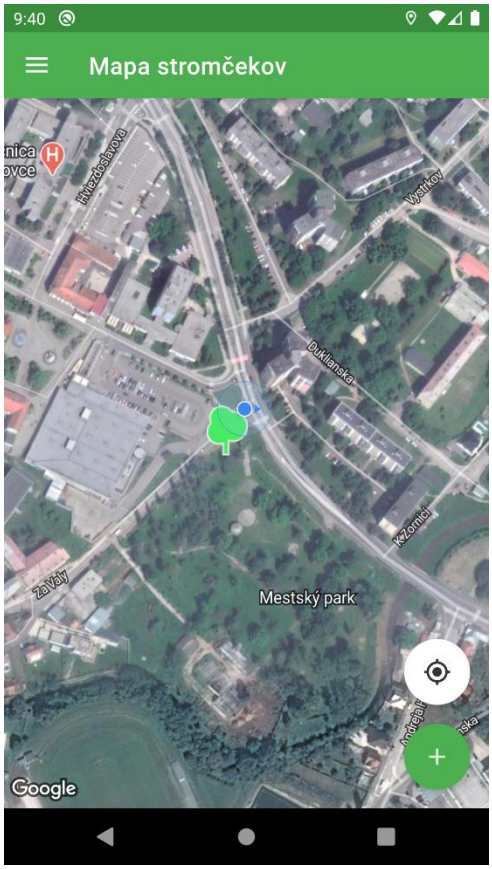
3.12 Editácia a odstránenie stromu

Editovanie stromu, a tiež jeho odstránenie bude možné v samostatnej aktivite TreeDetailsActivity. Túto aktivitu spustíme po kliknutí na položku v zozname pridaných stromov. Do aktivity prepošleme Tree objekt pomocou bundle. Keďže Tree je trieda a nie je to primitívny dátový typ, objekt získame v aktivite pomocou príkazu getSerializableExtra. Aby sme Tree objekt takto mohli získať, musí Tree trieda implementovať rozhranie Serializable.

Dizajn aktivity bude veľmi podobný dizajnu v pridaní stromu. Upravené údaje bude takisto riešiť nový ViewModel. Dizajn našej aktivity bude obsahovať navyše tlačidlo na zmazanie stromu. Po kliknutí zobrazíme dialóg, v ktorom sa užívateľa spýtame, či chce naozaj strom odstrániť. Ak odstránenie potvrdí strom odstránime pomocou požiadavky na server a aktivitu ukončíme príkazom finish. Po ukončení sa zobrazí aktivita alebo fragment, ktorý bol otvorený pred TreeDetailsActivity.

4 Výsledky a diskusia

Výsledkom tejto práce je mobilná aplikácia pre spravovanie stromov v meste Bánovce nad Bebravou. Po otestovaní je aplikácia plne funkčná a je možné vygenerovať inštalačný súbor, pomocou ktorého je možné aplikáciu nainštalovať na ľubovoľný smartphone so systémom Android s verziou 4.4 a viac. Tiež je možné aplikáciu pridať do obchodov s aplikáciami. Aplikácia je jednoduchá na ovládanie. Po prihlásení sa používateľovi zobrazí mapa pridaných stromov a vysúvacie menu. Aplikácia bola písaná tak, aby bola responzívna na viacerých mobilných zariadeniach.



Obr. 15 Ukážka MapTreesFragmentu (Dominik Ježík 2020)

5 Záver práce

Naša mobilná aplikácia je hotová a jej určenie je obyvateľov mesta Bánovce nad Bebravou, ktorí chcú pomôcť so zberom informácií o stromoch v tomto meste. Cieľom bolo vytvoriť Android aplikáciu, ktorá dokáže evidovať stromy a informácie o nich v meste. Čitateľovi práce sme zobrazili postup akým bola aplikácia tvorená od samotného návrhu aplikácie až po kompletnú tvorbu. V práci sme tiež popísali ako fungujú použité technológie a návrhové vzory. Aplikáciu je možné v budúcnosti rozširovať podľa požiadaviek, a tiež je možné túto aplikáciu rozšíriť aj medzi ďalšie mestá. V prípade väčšieho záujmu je možné po vytvorení vývojárskeho účtu aplikáciu zverejniť v obchode s aplikáciami – Google Play.

6 Zhrnutie

Vytvorením aplikácie pre spravovanie stromov sme poskytli možnosť ako riešiť evidenciu stromov v meste. Práca prevádza čitateľa informáciami o tvorbe aplikácií pre operačný systém Android, ktorý je v súčasnosti jeden z najpoužívanejších systémov. V teoretickej časti sme čitateľovi vysvetlili ako fungujú použité technológie a ako fungujú základné stavebné prvky Android aplikácií. Tiež je tu vysvetlený návrhový vzor Dependency Injection, použití pri tvorbe aplikácie. Podrobne je tu napísaný aj postup pri konkrétnej tvorbe a riešenia problémov, ktoré pri vývoji vznikali. Praktická časť práce je mobilná aplikácia.

7 Zoznam použitej literatúry

Android (operačný systém). [online],[cit. 20.11.2020]. Dostupné na internete: https://sk.wikipedia.org/wiki/Android_(opera%C4%8Dn%C3%BD_syst%C3%A9m)

Dependency Injection. [online],[cit. 22.11.2020]. Dostupné na internete: https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

Aktivity. [online],[cit. 1.12.2020]. Dostupné na internete: https://developer.android.com/guide/components/activities/intro-activities

Fragmenty. [online],[cit. 1.12.2020]. Dostupné na internete: https://developer.android.com/guide/fragments

Android software development. [online],[cit. 2.12.2020]. Dostupné na internete: https://en.wikipedia.org/wiki/Android_software_development

Android Developers. [online],[cit. 5.12.2020]. Dostupné na internete: https://developer.android.com/

Prílohy

Príloha A: DVD