Izrada Android aplikacije korištenjem Dagger2 okoline

Curilović, Nikola

Sveučilište u Rijeci – Odjel za informatiku

Poslovna informatika

Nikola Curilović

Izrada aplikacije za Android uređaje korištenjem Dagger2 okoline

Diplomski rad

Mentor: dr.sc. Marina Ivašić Kos

Rijeka, rujan 2018.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Zadatak za diplomski rad

Akademska 2016./2017. godina

Student: Nikola Curilović
Mentor/komentar: dr.sc. Marina Ivašić-Kos

Naziv diplomskog rada: Izrada aplikacije za Android uređaje korištenjem Dagger2 *frameworka*
Naziv diplomskog rada na engleskom jeziku: Developing Android application with Dagger2 framework

Sažetak teme/sadržaj zadatka diplomskog rada: U ovom diplomskom radu prikazat ću današnje standarde izrade Android aplikacije. Sama je izrada Android aplikacija podosta napredovala te mi je cilj napraviti i prikazati uporabu biblioteka i okoline poput MVP-a, Daggera i RxJave i mnogih drugih.

Mentor:

Voditelj za diplomske radove:

Komentar:

Zadatak preuzet:

_____

Sadržaj

Sažetak

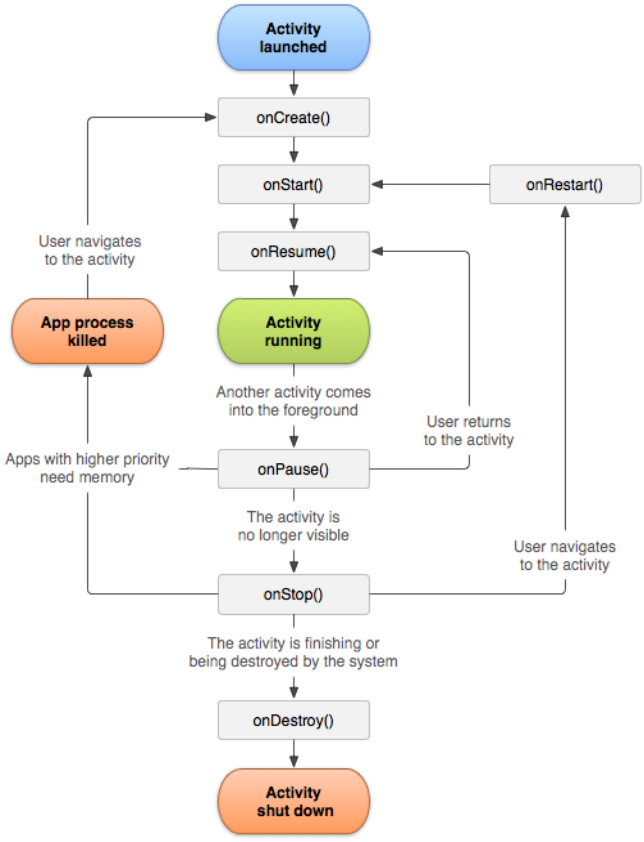
U diplomskom je radu opisana izrada Android aplikacije pomoću najnovijih tehnologija, odnosno opisana je implementacija MVP arhitekture i najpoznatijih biblioteka (Dagger2, RxJava) koje omogućuju da aplikacija bude skalabilna, sklona promjenama i laka za testiranje. Aplikacija se sastoji od geografske karte na kojoj su prikazane lokacije restorana (preuzete sa lokalnog poslužitelja) koje možemo dodatno uređivati (mijenjati adresu, ime ili sliku restorana) te spremati u lokalnu bazu podataka.

Ključne riječi: Mobilna aplikacija, Android, MVP arhitektura, Dagger2, RxJava2, Retrofit2, Kotlin, Dependency injection, Realm baza podataka, testiranje, Google Maps, Gradle.

1. Uvod

Mobilna aplikacija je računalni program namijenjen prijenosnim uređajima kao što su pametni telefoni, tablet računala i pametni satovi. Prvobitno su služile za obavljanje osnovnih funkcija, no zbog velike potražnje i brzog razvitka tehnologije poprimile su sasvim novi oblik.

Danas, deset godina kasnije od kada su postale popularne, mobilne aplikacije dio su naše svakodnevice te se koriste u različite svrhe. Mobilna aplikacija u današnje vrijeme može biti videoigra, kalkulator, kompas pa čak i sredstvo za naručivanje brze hrane. Samim time, njihova izrada je postala kompleksnija pa se dosta pažnje počelo posvećivati arhitekturi (predlošcima) izrade. Kako bi se arhitektura uklopila u samu izradu, potrebno je poštivati komponente operacijskog sustava na kojemu se arhitektura primjenjuje, te životni vijek aplikacije koja se izvodi na mobilnom uređaju. Android operacijski sustav ima jedinstveni životni vijek koji je prikazan dijagramom 1.



Dijagram 1. životni ciklus aplikacije

Svaka aktivnost ima svoj životni ciklus, od pokretanja aplikacije pa sve do zaustavljanja. Za svaku fazu životnog ciklusa definirana je odgovarajuća metoda. Kada je aplikacija pokrenuta (*onCreate()* metoda) i prikazana na korisnikovom ekranu, aktivnost je u fokusu (aktivna) – *onStart()* metoda. Ako je aktivnost izgubila fokus, aktivnost je pauzirana (*onPaused()* metoda), ako je aktivnost ponovo vraćena u fokus - *onResume()* metoda. Aktivnost može u cijelosti izgubiti fokus (ako je pokrenuta neka druga aktivnost unutar same aplikacije ili druge aplikacije), aktivnost je zaustavljena – *onStop()* metoda. Aktivnost može biti u cijelosti zaustavljena sa strane korisnika ili samog sustava (ako je potrebno osloboditi memoriju operacijskog sustava) – *onDestroy()* metoda.

U ovom diplomskog radu cilj je prikazati korištenje i implementaciju najpoznatijih biblioteka i arhitektura za izradu Android aplikacije. Kroz cijeli se projekt koristi Kotlin programski jezik [1] te je korišten Android studio [2] kao editor (IDE) za pisanje. Koristi se Dagger2 biblioteka [3] za Dependency Injection, RxJava2 [4] za upravljanje pozadinskim procesima, Retrofit2 [5] za komuniciranje s udaljenim poslužiteljima, Ted biblioteka [6] za lakše upravljanje korisničkim dozvolama, SquareCamera [7] za snimanje fotografija te Google Maps [8] za prikazivanje geografskih karti. U aplikaciji se koristi MVP (eng. *Model-View-Presenter*) arhitektura [9] koja omogućuje podjelu aplikacije u tzv. slojeve koji su zaslužni za odvajanje logike od elemenata za prikazivanje korisničkog sučelja. Spomenuta arhitektura sa gore navedenim bibliotekama osigurava skalabilnost, sigurnost i modularnost koja vodi do lakšeg testiranja samog koda [10].

2. Programski jezik Kotlin

Kotlin [1] je objektno orijentirani programski jezik koji se pokreće preko Java virtualnog stroja (eng. *Java virtual machine*). Također, Kotlin kod može biti kompajliran u JavaScript kod korištenjem LVVM [11] (eng. L*ow Level Virtual Machine*) kompajlera. Kotlin programski jezik je razvijen od strane JetBrains kompanije koja stoji iza mnogih alata za pisanje programskog koda (IntellJ alati) te se temelji na otvorenom kodu. Sintaksa koda nije kompatibilna sa Javom, ali je Kotlin napravljen da "surađuje" s Javom, odnosno mogu se koristiti u istom Java paketu [12] te se klase i metode mogu međusobno pozivati bez obzira jesu li one pisane u Java ili u Kotlin programskom jeziku.

Android studio od verzije 3.0 službeno podržava Kotlin kao jezik za izradu Android aplikacija što je uvelike doprinijelo popularnosti jezika. Kotlin stavlja naglasak na interoperabilnost, sigurnost, jasnoću i podršku te kao takav predstavlja "nadogradnju" Jave [1]. U nastavku je promjer Kotlin koda koji ispisuje "Hello, world" u metodi *main* koja označava početak svakog progama (kao i u Javi).

```
fun main(args: Array<String>) {
  val world = "world"
  println("Hello, $world!")
}
```

Metoda main()

U nastavku su nabrojane neke od karakteristika Kotlina koje čine razliku u odnosu na Java programski jezik. Važno je napomenuti kako je Kotlin relativno mlad programski jezik (prva verzija izašla je 2011. godine) koji se neprestano razvija.

Funkcija proširenja

Kao što samo ime sugerira, funkcija proširenja nam omogućuje da dodatno proširimo već postojeću metodu (ili klasu) bez da ju naslijeđujemo. Metoda *greet* ispisuje " je najbolji" svaki put kada se pozove na nekom stringu (u ovom slučaju "Marko", dobivamo "Marko je najbolji!").

```
fun String.greet() : String {
   return this.plus(" je najbolji!")
}
"Marko".greet()
```

Metoda greet()

Funkcija višeg stupnja

Funkcija višeg stupnja je ona funkcija koja prihvaća funkciju kao parametar ili vraća funkciju. Isto tako, postoje i funkcije prvog reda koje prihaćaju i vraćaju sve vrste parametara osim funkcije. Metoda *addValue* vrši definiranu radnju (prihvaća funkciju kao parametar) te vraća rezultat te radnje. U prvom slučaju vraća se rezultat množenja dok u drugom slučaju oduzimanja.

```
fun addValue(operation:(Int, Int) -> Int) : Int {
   return operation(10, 20)
}
addValue{num1, num2 -> num1 * num2} // Rezultat je 200
addValue{num1, num2 -> num1 - num2} // Rezultat je -10
```

Metoda addValue()

Null pointer

U Kotlinu se null pointer može lako izbjeći uporabom ?. operatora koji prvo provjerava postoji li ikakva vrijednost. Ako vrijednost postoji, program se nastavlja dalje izvršavati.

```
val str:String? = "ime"
str = null
str.length // Prikazati će se greška u obliku ?. te se greška ne prikazuje
kao NPE (null pointer exception)

str?.length // Izraz je potpuno valjan
str!!.length // Prikazati će se NPE ako varijabla str ne poprimi vrijednost
```

Primjer null varijable

Zadani i imenovani argumenti

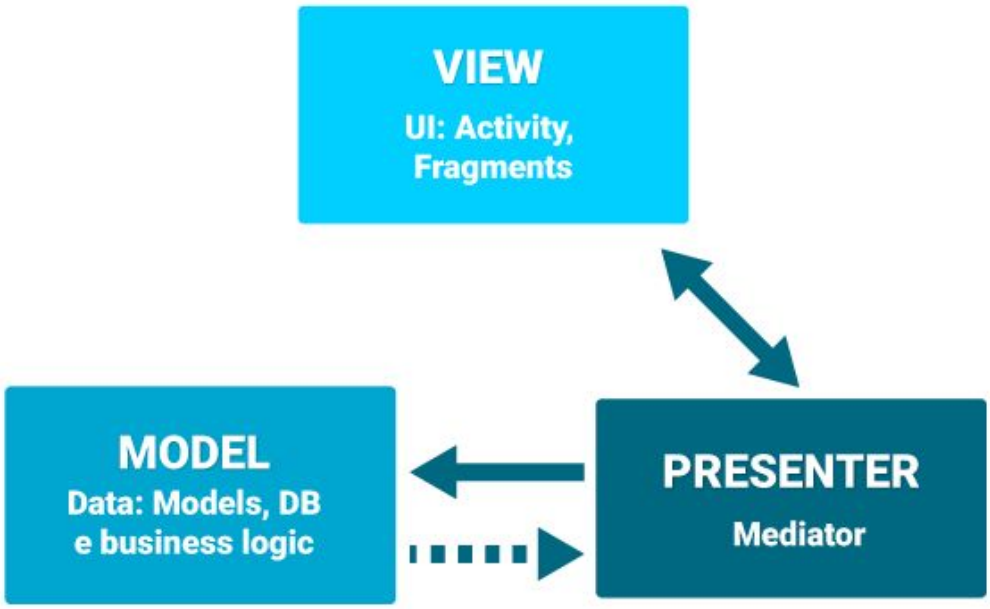
Dopušteno je imenovanje argumenata koje nam omogućuje mijenjanje redoslijeda argumenata u funkciji. U metodi *argumentValid* zadana je početna vrijednost drugog parametra dok prvi parametar nije definiran

```
fun argumentValid(num: Int, str: String = "12") {
}
argumentValid(15, "Bok bok")
argumentValid(str = "Ime", num = 45) //Imenovani argument
argumentValid(45) //Uporaba zadanog argumenta
```

Metoda argumentValid()

3. MVP arhitektura (predložak)

MVP (eng. *Model-View-Presenter*) arhitektura (eng. *Pattern no* zbog boljeg razumijevanja, u nastavku seminara koristi se riječ *arhitektura*) je nastala od dobro poznate MVC (Model-View-Contoller) arhitekture. MVP se sve više upotrebljava pri izradi Android aplikacija te primarno služi kako bi se odvojio prezentacijski sloj od same logike. Bez dodatne arhitekture, sva se logika nalazi u pogledu (eng. *view*) odnosno Activitiu (ili Fragmentu), koji samim time postaje neprikladan za pisanje testova te nije sklon naknadnim promjenama. Idealna je MVP arhitektura postignuta onda kada se može mijenjati sučelje a logika ostaje ista. Dakle, sučelje je odvojeno i ne ovisi o tome kako je prikazano.



Dijagram 2. MVP arhitektura

Iz dijagrama je vidljivo da *view* komunicira sa *presenterom*. U Javi odnosno Kotlinu to se postiže sučeljem (eng. *interface*) kojeg implementiramo u *view*. *Presenter* komunicira sa *modelom* te kada je akcija završena *presenter* obavještava *view*. Dakle, *view* nema nikakve veze sa *modelom*. Ovom arhitekturom dobivamo tri različita sloja koja možemo zasebno testirati pa se tako implementiraju jedinični testovi [13] (eng. *unit tests)* odnosno intrumentacijski testovi [14] u *viewu*.

Presenter

Presenter je posrednik između *viewa* i *modela* koji prihvaća podatke iz *modela* i formatirane ih vraća u View. MVP arhitektura razlikuje se od MVC [15] i po tome što *presenter* odlučuje što će se dogoditi kada imamo interakciju s *viewom*.

View

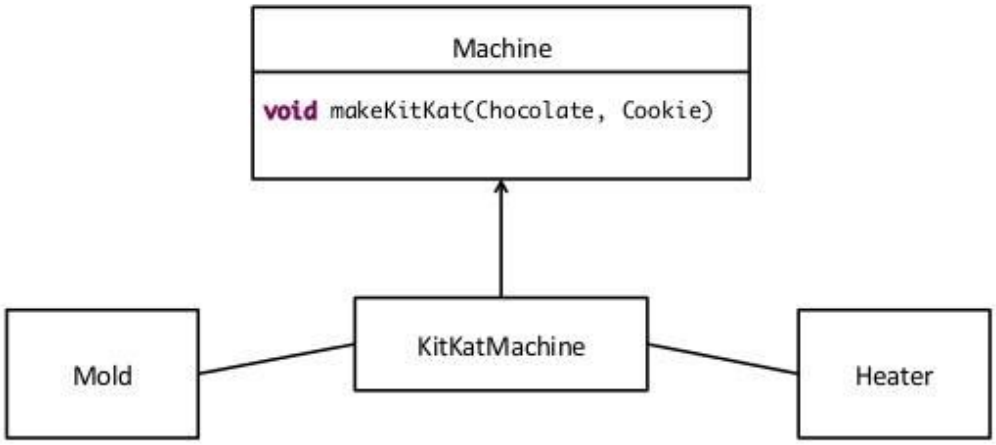
View je u većini slučajeva implementiran u Activityju (ili Fragmentu) koji sadrži referencu za *presenter*. U idealnom slučaju, kada se upotrebljava Dagger biblioteka, *presenter* će biti kreiran pomoću njega. Ako se ne koristi Dagger, *view* je odgovoran i za kreiranje objekta *presentera*. View se koristi za prihvaćanje korisničke interakcije te on zatim zove metodu iz *presentera*.

Model

U dobro strukturiranoj aplikaciji, *model* predstavlja posrednika do sloja domene ili poslovne logike. Neki ga nazivaju i *interactor* ili *controller*, ovisno o slučaju i parametrima.

4. Dependency injection

Kada se raspravlja o pojmu *dependency injection* [16] potrebno je spomenuti pravilnu terminologiju. Dakle, kada neki kod u klasi A referencira klasu B, tada je klasa A ovisna o klasi B. U tom kontekstu klasa A postaje “klijent”, a klasa B “usluga”. Primjer ovisnosti klasa prikazan je dijagramom 3. Klasa *KitkatMachine* (klijent) implementira sučelje *Machine* (usluga) te se sastoji od klasa *Mold* (usluga) i *Heater* (usluga).



Dijagram 3. Dijagram ovisnosti

Isto tako, prije opisivanja pojma dependency injection treba napomenuti kako postoji bitna razlika izmedu pojmova dependency injection i Dependency Injection (velika slova). Prvi pojam odnosi si se na akciju pružanja (eng. injecting) usluga klijentu dok drugi pojam predstavlja arhitekturu koja definira tehnike i principe pružanja usluga na sistemskoj razini. Samim time, ovi se pojmovi odnose i na drugačije apstrakcije - dependency injection djeluje na razini klase, dok Dependency Injection djeluje na sistemskoj.

Tehnike za dependency injection

Postoje tri tehnike koje se koriste za dependency injection ovisno o slučaju: *Constructor Injection, Method Injection* i *Field Injection*. Zanimljiva je činjenica da se u objektnom programiraju ove tehinike koriste na svakidašnjoj razini, ali se skoro nikad ne izdvajaju zasebno.

Dependency Injection kao arhitektura

Glavna karakteristika dobre implementacije arhitekture je razdvajanje logike u dva (matematički rečeno) skupa klasa:

- *Funkcionalni set* - klase koje enkapsuliraju glavnu jezgru (eng. core) aplikacije
- *Konstrukcijski set* - klase opredjeljuju ovisnosti i konstruiraju objekte za funkcionalni skup

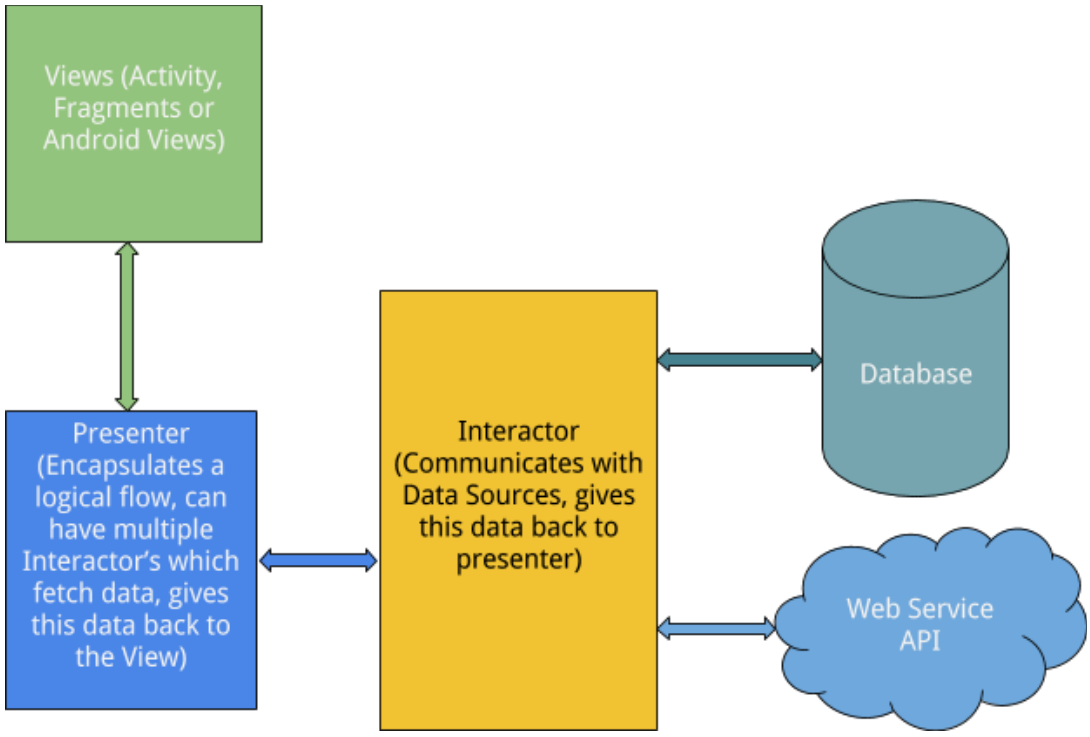
Kako bi se zadovoljili navedeni skupovi klasa, potrebno je zadovoljiti slijedeća pravila:

- Klase koje enkapsuliraju glavnu funkcionalnost aplikacije ne smiju dijeliti zavisnosti ili inicijalizirati klase iz funkcionalnog skupa
- Klase koje dijele zavisnosti ili inicijaliziraju klase iz funkcionalnog seta ne smiju enkapsulirati niti jednu glavnu funkcionalnost

Da bi se postigla potpuna neovisnost od glavnih funkcionalnosti i ostalih dijelova aplikacije, u ovom seminarskom radu koristi se biblioteka Dagger2 koja olakšava povezivanje funkcionalnog i konstrukcijskog skupa u jednu cijelnu.

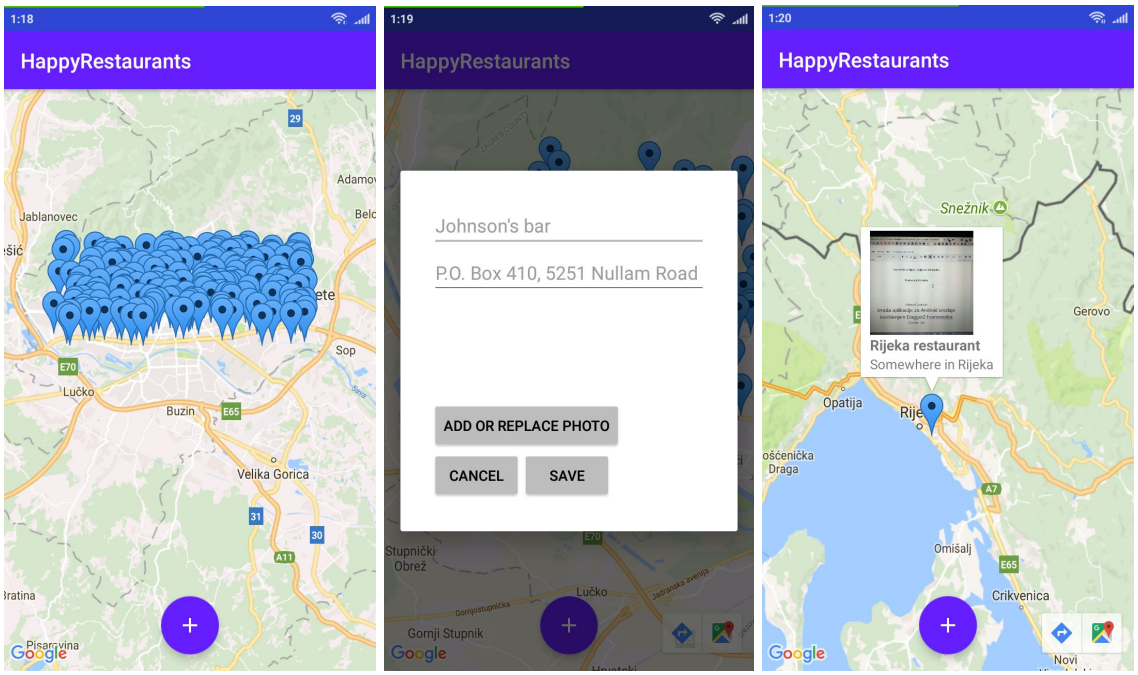
5. Opis aplikacije

Aplikacija prikazuje mapu na kojoj se nalazi lista restorana koji su na mapi označeni pinovima (Slika 1). Svaki restoran ima ime, adresu i/ili sliku. Prilikom prvog pokretanja, aplikacija automatski preuzima sadržaj sa udaljenog poslužitelja te ga sprema lokalno u bazu podataka (dijagram 4.). Za preuzimanje podataka sa udaljenog poslužitelja i spremanje u lokalnu bazu podataka zaslužan je *GetRestaurantsInteractor* (detaljniji opis nalazi se u nastavku rada - poglavlje 7.3)



Dijagram 4. Prikaz arhitekture sa lokalnim bazom podataka i udaljenim serverom

Prilikom odabira nekog restorana otvaraju se detalji koje je moguće mijenjati. Ako korisnik želi mijenjati sliku, otvara mu se kamera kojom može zabilježiti sliku. Ako korisnik nije uključio GPS, nema internet konekciju ili nije dopustio dozvole za korišćenje kamere ili pozicije, bit će obavješten. Aplikacija koristi geografsku lokaciju kako bi se odredila trenutna pozicija korisnika te je time omogućeno dodavanje vlastitog restorana. U desnom uglu nalaze se upute (koordinate lokacije) koje se otvaraju u Google Maps aplikaciji što omogućuje lakši dolazak do destinacije.



Slika 1. Prikaz aplikacije

5.1. Gradle sustav automatizacije

Android Studio koristi Gradle sustav [17] za izgradnju, testiranje, objavljivanje i testiranje projekta (aplikacije) koji ima mogućnost upravljanja ovisnostima (bibliotekama). Gradle je jedan od najpoznatijih sustava za automatizaciju te ga koriste mnogi softverski projekti. Android studio koristi dodatak (eng. *plugin*) koji se koristi za komunikaciju sa Gradle sustavom no isti nije potpuno ovisan sa Android Studiom, što znači da se može izgraditi Android aplikacija pomoću komandne linije i bez Android studia. Isto tako, Gradle je baziran na JVM sustavu čime je omogućeno pisanje vlastitih skripti u Java programskom jeziku.
Android aplikacija sastoji se od dvije Gradle datoteke (build.gradle) od kojih se jedna odnosi na aplikacijski modul, a druga na cjelokupni projekt koja ima zajednička svojstva za sve module.
U Gradle datoteci koja se odnosi na modul navedene su neke početne postavke za projekt (aplikacijska oznaka, minimalna verzija SDK Android paketa, verzija koda itd.), postavke za potpisivanje Android aplikacije kojom možemo ubrzati cijeli proces (umjesto manualnog upisivanja možemo unaprijed definirati željene postavke) te postavke za produkcijsku verziju aplikacije i verziju aplikacije dok je aplikacija u izradi (*API_KEY* sadrži adresu udaljenog poslužitelja).

```
// Glavne postavke verzije koda i aplikacije
defaultConfig {
   applicationId "ncurilovic.masterthesis.happyrestaurants"
   minSdkVersion rootProject.minSdkVersion
   targetSdkVersion rootProject.targetSdkVersion
   versionCode rootProject.versionCode
   versionName rootProject.versionName
   testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
}
// Ključ za potpisivanje aplikacije te mjesto spremanja ključa
// Ključ je potrebno generirati pri završetku
signingConfigs {
   release {
       keyAlias 'HappyRestaurants'
       storeFile
file('/home/nikola/Android/apk/HappyRestaurants/keystore.jks')
       keyPassword 'HappyRestaurants'
```

```
        storePassword 'HappyRestaurants'
    }
}
// Postavke udaljenog poslužitelja koje mogu biti drugačije ovisno da li se
radi o verziji u izradi ili finalnoj (produkcijskoj)
buildTypes {
    def API_KEY = '"http://www.mocky.io/"'
    debug {
        debuggable true
        applicationIdSuffix '.development'
        buildConfigField "String", "API_KEY", API_KEY
    }
    release {
            buildConfigField "String", "API_KEY", API_KEY
    }
}
```

Gradle datoteka - aplikacijski modul

5.2. Korištene biblioteke

Kao i u većini današnjih Android aplikacija, i u ovome se projektu koriste biblioteke koje olakšavaju razvoj:

- *JUnit* [18] - okruženje koje služi za pisanje jediničnih testova (eng. unit tests)
- *Dagger2* [3] - biblioteka za dependency injection
- *RxJava2* [4] - reaktivne ekstenzije koje koristimo za upravljanje procesima koji rade u pozadini
- *Realm* [19] - lokalna baza podataka koja omogućuje objektno spremanje podataka
- *SquareCamera* [7] - biblioteka koja nam služi za upravljanje kamerom, namijenjena je svim uređajima koji imaju SKD veći od 14 što rješava problem fragmentiranosti Android uređaja
- *Powermock* [20] - biblioteka koja nam omogućuje testiranje statičkih metoda
- *Remember* [21] - “omotač” za pohranjivanje stringova u lokalnu memoriju
- *Ted* - biblioteka za upravljanje korisničim dozvolama
- *Mockito* [22]- koristi se za oponašanje realnih objekata kako nebi morali imati stvarne podatke za testiranje

- *Retrofit* [5] - biblioteka koja nam omogućuje lako i jednostavno povezivanje na udaljeni poslužitelj preko sučelja

Osim gore navedenih biblioteka, u projektu se koriste i biblioteke koje nam služe kao podrška ostalima:

- *Support Library* [23] - biblioteka koja sadrži sve osnovne metode za nesmetani rad aplikacije
- *RxKotlin, RxAndroid, RxAdapter* [4] - reaktivne ekstenzije koje se koriste za rad s bibliotekom RxJava na Android uređajima odnosno omogućuju da RxJava nesmetano radi s Kotlin programskim jezikom i Retrofit bibliotekom
- *OkHttp* [5] - koristi se uz Retrofit biblioteku te služi za upravljanje http zahtjevima
- *Play Services* [24] - biblioteka zaslužna za prikazivanje geografske karte

5.3. Inicijalizacija biblioteka

Neke biblioteke koje se koriste u ovom radu zahtjevaju da se inicijaliziraju na samom startu (pokretanju) aplikacije. Najefikasniji način za rješavanje tog zahtjeva je izrada vlastite klase (*TheApplication.kt*) koja nasljeđuje klasu *Application()*. Ona se inicijalizira u glavnim postavkama projekta (*AndroidManifest*):

```
android:name="ncurilovic.masterthesis.happyrestaurants.TheApplication"
```

Od sada se pri pokretanju aplikacije najprije pokreće klasa *TheApplication* u kojoj se nalazi inicijalizacija Dagger2, Realm i Remember biblioteka.

```
class TheApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        initialize()
    }

    private fun initialize() {
        appComponent = DaggerAppComponent.create()
        Realm.init(this)
        Remember.init(applicationContext, "happy.restaurants")
```

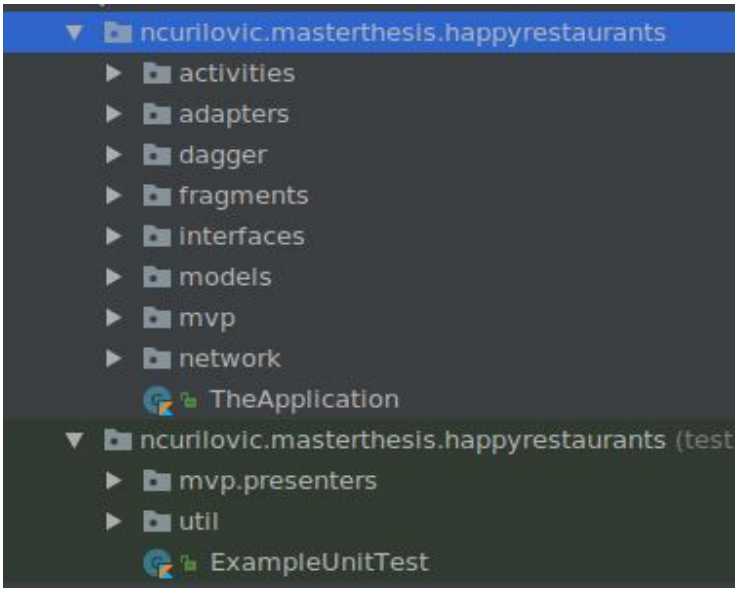
```
    }

    object Holder {
        val INSTANCE_HOLDER = TheApplication()
    }

    companion object {
        val instance: TheApplication by lazy { Holder.INSTANCE_HOLDER }
        lateinit var appComponent: AppComponent
    }
}
```

Klasa *TheApplication*

Kao i svaka klasa u Android okruženju sadrži *onCreate()* metodu koja se prva pokreće. Nakon toga se inicijaliziraju biblioteke, stvori Dagger komponenta i objekt koji sadrži instancu aplikacije koju kasnije pozivamo u ostalim klasama projekta. *Companion* objekt je ekvivalent statičnoj klasi u Javi.

6. Pregled korištenih paketa

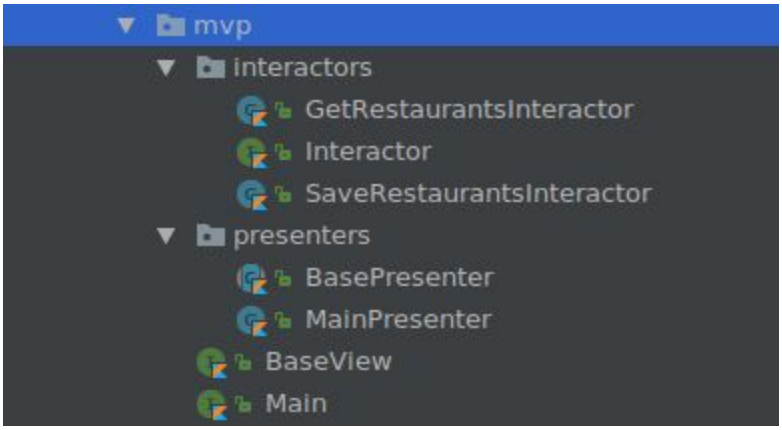
Kao i Java programskom jeziku, Kotlin podržava pakete s kojima možemo "grupirati" klase. Klase time dobivaju više smisla te je preglednost veća. Paketi su razvrstani po komponentama ovisno o funkciji. *Activities* i *Fragments,* kao što i ime govori, sadrže aktivnosti i fragmente koji su zaslužni za prikazivanje korisničkog sučelja. U paketu *adapters* nalazi se adapter koji prikazuje pinove na mapi. *Dagger* paket sadrži sve klase vezane uz dependency injection. *Interfaces, network i models* sadrže klase koji se tiču preuzimanja podataka sa udaljenog poslužitelja dok *mvp* paket sadrži sučelja, modele (interaktore u nastavku) i presentere arhitekture.



Slika 2. Pregled paketa

MVP paket

MVP paket sastoji se od dodatna dva paketa: *interactors* i *presenters.* U njima se nalaze modeli (interaktori) i presenteri. Također, u MVP paketu se nalaze i sučelja koja sadrže dodatna sučelja s metodama za svaku od klasa te dvije bazične klase u kojima se nalaze najkorištenije metode. U interaktorima se nalazi logika dok presenteri sadrže reference za poglede i interaktore.



Slika 3. Pregled MVP paketa

6.1. Podešavanje Dagger2 biblioteke

Moduli

Osnovni dijelovi Dagger biblioteke su komponente i moduli. Moduli su zapravo dijelovi koda koje želimo "dostaviti" odnosno *injectati*. Klase se označavaju sa anotacijama *@Module* dok se ispred metoda (ime metoda nije bitno) nalazi anotacija *@Provides.* Važno je naglasiti da kada se implementiraju metode u modulu koje su ovisne, ne treba ih specifično naglasiti jer Dagger automatski prepoznaje takav slučaj te sam izgenerira potreban dio koda. U projektu imamo tri modula:

1. *ApiModule* - modul zaslužan za kreiranje objekta Retrofita
2. *AppContextModule* - modul u kojem se nalazi aplikacijski kontekst
3. *MainModule* - modul za MVP u kojem se nalaze sučelja koja grade arhitekturu

```
@Module
class AppContextModule {

    @Provides
    fun provideAppContext(): TheApplication? {
        return TheApplication.instance
    }
}
```

Klasa *AppContextModule*

MainModule nam služi kako bi injectali komponente MVP arhitekture. Sastoji se od sučelja koja implemetiraju klase čime postižemo da svi dijelovi budu odvojeni, polimorfizam te lakše testiranje. *MainModule* sadrži reference za *view* i *presenter* sučelja te sučelja *interactora* (*GetRestaurantInteractor* i *SaveRestaurantInteractor*).

```
@Provides
fun provideMainView(): Main.View {
    return view
}

@Provides
fun provideMainPresenter(mainPresenter: MainPresenter): Main.Presenter {
    return mainPresenter
}

@Provides
fun provideGetRestaurantsInteractor(getRestaurantsInteractor:
GetRestaurantsInteractor): Interactor.GetRestaurants {
    return getRestaurantsInteractor
}

@Provides
fun provideSaveRestaurantsInteractor(saveRestaurantsInteractor:
SaveRestaurantsInteractor): Interactor.SaveRestaurants {
    return saveRestaurantsInteractor
}
```

Klasa *MainModule*

Nakon *modula*, moramo specificirati kako injectati ovisnosti. Dagger nudi tri mogućnosti: *Constructor injection, field injection* i *method injection. MainModule* sadrži metode koje kao rezultat vraćaju sučelja koja implementiraju klase pa je zato u ovom slučaju prikladno koristiti *Constructor injection* u presenteru odnosno interactoru. Da bi se to postiglo, koristimo anotaciju @*Inject* koju stavljamo ispred konstruktora.

```
MainPresenter @Inject constructor()
```

Primjer korištenja @*Inject* anotacije

Komponente

Dagger komponente služe kako bi izgradili "mrežu" zavisnosti te odredili mjesto injectiona. Komponente mogu biti dostupne na aplikacijskoj razini (koristi se anotacija @*Component*) ili na razini jedne klase (anotacija @*Subcomponent*). Pored notacija, u nastavku se moraju navesti i željeni moduli. Radi boljih performansa, često se koristi notacija @*Singleton* kako bi se označilo instancu koju je dovoljno jednom napraviti u aplikaciji. U projektu imamo dvije komponente:

1. *AppComponent* - komponenta koja se koristi na razini aplikacije, sadrži *AppContextModule* i *ApiModule* te *MainModule* koji je zapravo podkomponenta pa nije potrebno specifično navesti
2. *MainComponent* - podkomponenta koja omogućava injectanje *MainModula*

```
@Component(modules = arrayOf(AppContextModule::class, ApiModule::class))
@Singleton
interface AppComponent {

    fun plus(module: MainModule): MainComponent

    fun apiService(): ApiService
}
```

Komponenta *AppComponent*

```
@Subcomponent(modules = arrayOf(MainModule::class))
interface MainComponent {

   fun inject(activity: MainActivity)
}
```

Komponenta *MainComponent*

Korištenje objekata

Nakon što su kreirani moduli i komponente, objekti se mogu *injectati.* U ovom slučaju objekt presentera se *injecta* u klasu *MainActivity.* U metodi *injectDependencies()* (koja je apstraktna metoda te se nalazi u klasi *BaseActivity*) kreiramo objekt modula.

```
override fun injectDependencies(appComponent: AppComponent) {
   appComponent.plus(MainModule(this))
           .inject(this)
}
```

Metoda *injectDependencies()*

Operacija *inject* vrši se anotacijom *@Inject.* U Kotlin programskom jeziku dodajemo ispred varijable izraz *lateinit* ako se varijabla ne inicijalizira odmah, već kasnije u izvođenju programa. U našem slučaju, varijabla je inicijalizirana, ali se *lateinit* koristi zbog sintakse programskog jezika.

```
@Inject
lateinit var presenter: Main.Presenter
```

Varijabla *presenter*

6.2. Podešavanje Retrofit biblioteke

Podešavanje Retrofita sastoji se od tri dijela: modula, sučelja i modela. Modul je specificiran u *AppComponent* komponenti što znači da je dostupan na razini cijele aplikacije. *ApiModul* kreira Retrofit objekt koji se sastoji od adaptera koji dodaje ekstenzije za integraciju sa RxJava2 bibliotekom, GSON konvertera koji omogućuje komunikaciju preko REST sučelja te bazičnu adresu udaljenog poslužitelja koji prikazuje podatke u JSON [25] obliku. Objekt kreira *ApiService* sučelje u kojemu se nalazi proširena adresa.

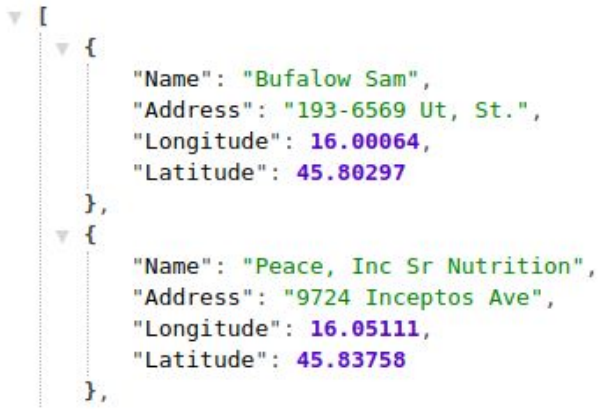
```
@Module
class ApiModule {

    @Provides
    @Singleton
    fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
              .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
              .addConverterFactory(GsonConverterFactory.create())
              .baseUrl(BuildConfig.API_KEY)
              .build()
    }

    @Provides
    @Singleton
    fun provideApiService(retrofit: Retrofit): ApiService {
        return retrofit.create<ApiService>(ApiService::class.java)
    }
}
```

Klasa *ApiModule*

Sučelje se sastoji od anotacije *@Get* koja označava da se radi o pozivu za dohvaćanje podataka, proširene adrese poslužitelja i apstraktne funkcije koja dohvaća informacije za restorane. *Single* je oznaka rxJave koja označava da se radi o samo jednom emitiranju podataka. Emitirani se podaci prihvaćaju u obliku liste te su prikazani pomoću modela *Restaurant*.

```
interface ApiService {

    @GET("/v2/54ef80f5a11ac4d607752717")
    abstract fun getRestaurants(): Single<List<Restaurant>>
}
```

Sučelje *ApiService*



Model odgovara JSON shemi koju dobivamo od poslužitelja. Podaci nisu realni te nam služe za testiranje. JSON shema je jednostavna te se sastoji od jednog reda (eng. array) i objekata. Ako se pokrene adresa u internet pregledniku, dobivamo rezultat kao sa slike 4.

Slika 4. Prikaz podataka u JSON formatu

Da bi se dobila odgovarajuća shema kojom se može dohvatiti podatke tj. prikazati u obliku Kotlin objekta, u modelu se treba specificirati imena varijabli kao u originalnoj shemi. Koristi se anotacija *@SerializedName* kako bi ime varijable bilo isto kao u JSON shemi. Varijable se nalaze u konstruktoru klase, klasa *Restaurant* je tipa *RealmObject* koji automatski generira metode za dohvaćanje podataka (eng. getters and setters).
Kod varijabla se nalazi *"?"* koji označava da varijable ne moraju poprimiti nikakvu vrijednost (mogu biti null vrijednosti - `name = null`).

```
open class Restaurant(

        @SerializedName("Name")
        var name: String?,
        @SerializedName("Address")
        @Expose
        var address: String?,
        @SerializedName("Longitude")
        @Expose
        var longitude: Double?,
        @SerializedName("Latitude")
        @Expose
        var latitude: Double?,
        var imagePath: String?

 ) : RealmObject() {
}
```

Model *Restaurant*

7. Pokretanje aplikacije

Nakon pokretanja aplikacije pokreće se *MainActivity* klasa odnosno metoda *onCreate()*. U njoj se nalaze četiri dodatne metode:

1. *addData()* - metoda koja provjerava je li aplikacija prvi put pokrenuta
2. *addMapFragment()* - metoda zaslužna za dodavanje geografske mape
3. *checkPersmissions()* - metoda u kojoj se se prikazuju dozvole koje su potrebne za normalno korišćenje aplikacije
4. *setupFAB()* - metoda u kojoj se inicijalizira plutajući gumb

Metoda *addData()* koristi biblioteku Remember koja pomoću varijable provjerava da li je aplikacija prvi put pokrenuta. Ako je pokrenuta, aplikacija preuzima podatke sa udaljenog poslužitelja i postavlja zastavicu koja to označava, dok u suprotnom preuzima podatke iz lokalne baze podataka. U oba slučaja poziva se odgovarajuća metoda iz presentera.

```
private fun addData() {
    if (Remember.getBoolean(IS_FIRST_RUN, true)) {
        presenter.getRestaurantsFromAPI()
        Remember.putBoolean(IS_FIRST_RUN, false)
        Log.d(TAG, "This is first run")
    } else {
        presenter.getRestaurantsFromLocalDb()
        Log.d(TAG, "This is not a first run")
    }
}
```

Metoda *addData()*

7.1. Postavljanje korisničkog sučelja

Sučelje se sastoji od trake s imenom aplikacije, geografske mape i plutajućeg gumba koji se nalazi pri dnu ekrana. Postavljanje mape vrši se pomoću metode *addMapFragment()* koja referencira geografsku mapu iz *activity_main.xml* datoteke (sadrži kod sučelja) te se vrši asinkrono učitavanje mape.

```
private fun addMapFragment() {
    val mapFragment = supportFragmentManager
            .findFragmentById(R.id.map) as SupportMapFragment
    mapFragment.getMapAsync(this)
}
```

Metoda *addMapFragment()*

Kada se učita geograska mapa, poziva se metoda onMapReady() u kojoj se postavlja varijabla mape kao globalna što nam omogućava kasnije dodavanje lokacija restorana.

```
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap
}
```

Metoda *onMapReady()*

Slično kao i geografsku mapu, postavlja se i plutajući gumb. U ovom slučaju dodatno se postavlja metoda koja se pokreće kada korisnik klikne na gumb.

```
private fun setupFAB() {
    val fab = findViewById(R.id.fab) as FloatingActionButton
    fab.setOnClickListener(this)
}
```

Metoda setupFAB()

Datoteka *activity_main.xml* sadrži kod korisničkog sučelja. U njemu su specificirana jedinstvena imena koja referenciramo u klasama, određena veličina geografske mape, boje itd. Na vrhu ekrana nalazi se traka u kojoj je prikazano ime aplikacije (eng. *toolbar*).

```
<android.support.design.widget.CoordinatorLayout
        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:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

    <android.support.design.widget.AppBarLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
        <include layout="@layout/toolbar"/>
    </android.support.design.widget.AppBarLayout>

    <fragment
            android:id="@+id/map"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:name="com.google.android.gms.maps.SupportMapFragment"
            tools:context="com.example.nikola.maps.MapsActivity"/>

    <android.support.design.widget.FloatingActionButton
            android:id="@+id/fab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/fab_margin"
            android:src="@drawable/ic_add"
            app:backgroundTint="@color/primary"
            app:layout_anchor="@id/map"
            app:layout_anchorGravity="bottom|center_horizontal"/>
</android.support.design.widget.CoordinatorLayout>
```

7.2. Upravljanje korisničkim dozvolama

Korisničke dozvole bitan su dio Android ekosustava i imaju veliku ulogu u sigurnosti. U aplikaciji se koristi biblioteka *Ted* koja nam olakšava cijeli proces davanja dozvola korisniku. U datoteci *AndroidManifest.xml* deklarirane su dozvole koje se koriste u aplikaciji:

- INTERNET - dozvola za korištenje interneta, koristi se za dohvaćanje podataka sa udaljenog poslužitelja

- ACCESS_FINE_LOCATION - dozvola koja omogućuje dohvaćanje trenutne pozicije korisnika, koristi se prilikom dodavanja novog restorana
- CAMERA - dozvola za korištenje kamere, koristi se prilikom dodavanja slike restorana
- WRITE_EXTERNAL_STORAGE - dozvola za pisanje podataka u lokalnu memoriju
- READ_EXTERNAL_STORAGE - dozvola za čitanje podataka iz lokalne baze podataka

Metoda *checkPersmissions()* zaslužna je za prikazivanje upita dozvola te prikazuje poruku kada je dozvola odbijena od strane korisnika. Upit sadrži samo tri dozvole (za kameru, čitanje i pisanje u lokalnu memoriju) jer one predstavljaju dozvole višeg stupnja te trebaju biti eksplicitno potvrđene sa strane korisnika aplikacije.

```
private fun checkPermissions() {
   TedPermission.with(this)
           .setPermissionListener(this)
           .setDeniedMessage(getString(R.string.permission_denied))
           .setPermissions(Manifest.permission.READ_EXTERNAL_STORAGE,
Manifest.permission.CAMERA, Manifest.permission.ACCESS_FINE_LOCATION)
           .check()
}
```

Metoda *checkPermissions()*

Metode *onPermissonDenied()* i *onPermissionGranted()* govore jesu li dozvole odbijene ili prihvaćene. U ovom slučaju te metode nemaju funkciju osim da ispišu poruku u konzolu.

```
override fun onPermissionDenied(deniedPermissions: ArrayList<String>?) {
   Log.d(TAG, "Permission denied")
}

override fun onPermissionGranted() {
   Log.d(TAG, "Permission granted")
}
```

Metode *onPermissionDenied()* i *onPermissionGranted()*

7.3. Preuzimanje podataka iz udaljenog poslužitelja

Ako je aplikacija prvi put pokrenuta, preuzimaju se podaci za udaljenog poslužitelja. Metoda *addData()* je zaslužna za provjeravanje tog slučaja (poglavlje 8) te se u njoj nalazi referenca presentera - *presenter.getRestaurantsFromAPI()*. Ova metoda nalazi se u klasi *MainPresenter,* u njoj se nalazi odbacujući objekt (eng. *disposable*) koji poziva metodu interaktora te emitira rezultate. Poziv se odvija u novom pozadinskom procesu (Schedulers.io), dok su rezultati prikazani u u glavnoj niti (MainThread. Ako je operacija uspješna, rezultat će biti prikazan u *onSuccess()* metodi u kojoj se poziva metoda za prikazivanje restorana u glavnoj aktivnosti aplikacije (*MainActivity).* Ukoliko je došlo do greške, prikazujemo korisniku obavijest.

```
override fun getRestaurantsFromAPI() {
   compositeDisposable.add(
           getRestaurants.fetchRestaurantsFromAPI()
                   .observeOn(AndroidSchedulers.mainThread())
                   .subscribeOn(Schedulers.io())
                   .subscribeWith(object :
DisposableSingleObserver<List<Restaurant>>() {
                       override fun onSuccess(restaurants:
List<Restaurant>?) {
                            if (restaurants != null) {
                               view.showRestaurants(restaurants)
                           }
                       }

                       override fun onError(e: Throwable?) {
                           Log.e(TAG, e.toString())
                           view.showNoDataError()
                       }
                   })
   )
}
```

Metoda *getRestaurantsFromAPI()*

Metoda *fetchRestaurantsFromAPI()* nalazi se u klasi *GetRestaurantsInteractor* i kao rezultat vraća sučelje Retrofita koje poziva *map()* metodu. *Map*() je dio biblioteke RxJava, sastoji se od dvije dodatne metode (u ovom slučaju koristi se samo *apply()* metoda) te služi kako bi se izvršila neka operacija nad rezultatima. U ovom slučaju rezultat se sastoji od liste restorana koju spremamo u lokalnu bazu podataka putem *Realm* biblioteke. Prije spremanja briše se postojeća baza podataka (ako ona postoji) kako bi se izbjeglo dupliciranje rezultata. Zatim se otvara se transakcija, kopiraju se podaci, zatvara se transkacija i vraća se rezultat kao objekt u *presenter*).

```
override fun fetchRestaurantsFromAPI(): Single<List<Restaurant>> {
   return apiService.getRestaurants()
           .map(object : Func1<List<Restaurant>, List<Restaurant>>,
Function<List<Restaurant>, List<Restaurant>> {
               override fun apply(restaurants: List<Restaurant>):
List<Restaurant> {
                   val realm = Realm.getDefaultInstance()
                   realm.executeTransaction { realm -> realm.deleteAll() }
                   realm.beginTransaction()
                   realm.copyToRealm(restaurants)
                   realm.commitTransaction()
                   realm.close()
                   return restaurants;
               }
           });
}
```

Metoda *fetchRestaurantsFromAPI()*

Ako je preuzimanje sa udaljenog poslužitelja uspješno, u klasi *MainPresenter* odnosno metodi *onSuccess()* dobiva se lista restorana (poglavlje 6.2, slika 4.). Ako lista nije prazna, preko sučelja se šalje u klasu *MainActivity* preko metode *showRestaurants()*.

```
if (restaurants != null) {
   view.showRestaurants(restaurants)
}
```

Metoda *showRestaurants()* je zaslužna za prikazivanje lokacija na geografskoj karti. U njoj se postavljaju markeri (oznake) sa koordinatama restorana, ime i isječak koji sadrži adresu i adresu slike (ako ona postoji). Zatim, postavlja se adapter koji raspoređuje informacije, postavlja ime, adresu i sliku na markere te se postavlja *listener* koji omogućava prepoznavanje korisničkog dodira na marker. Nakon što su markeri postavljeni, animiramo kameru na zadnji marker.

```
override fun showRestaurants(restaurants: List<Restaurant>) {
   restaurants.forEach { it ->
       map.addMarker(MarkerOptions().position(LatLng(it.latitude!!,
it.longitude!!)).title(it.name).snippet(it.address + "-" +
it.imagePath).icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFa
ctory.HUE_AZURE)))
   }

   map.setInfoWindowAdapter(CustomInfoWindowAdapter(this))
   map.setOnInfoWindowClickListener(this)
   animateCameraToLastMarker(restaurants.last())
}
```

Metoda *showRestaurants()*

Ako je došlo do greške (npr. uređaj nije spojen na internetsku vezu) prilikom preuzimanja podataka, u klasi *MainPresenter* pokreće se metoda *onError()* koja poziva metodu *showNoDataError()*.

```
view.showNoDataError()
```

Metoda *showNoDataError()* prikazuje korisniku obavijest da je došlo do greške.

```
override fun showNoDataError() {
   showError(getString(R.string.check_connection))
}
```

Metoda *showNoDataError()*

7.4. Prikazivanje podataka iz lokalne baze podataka

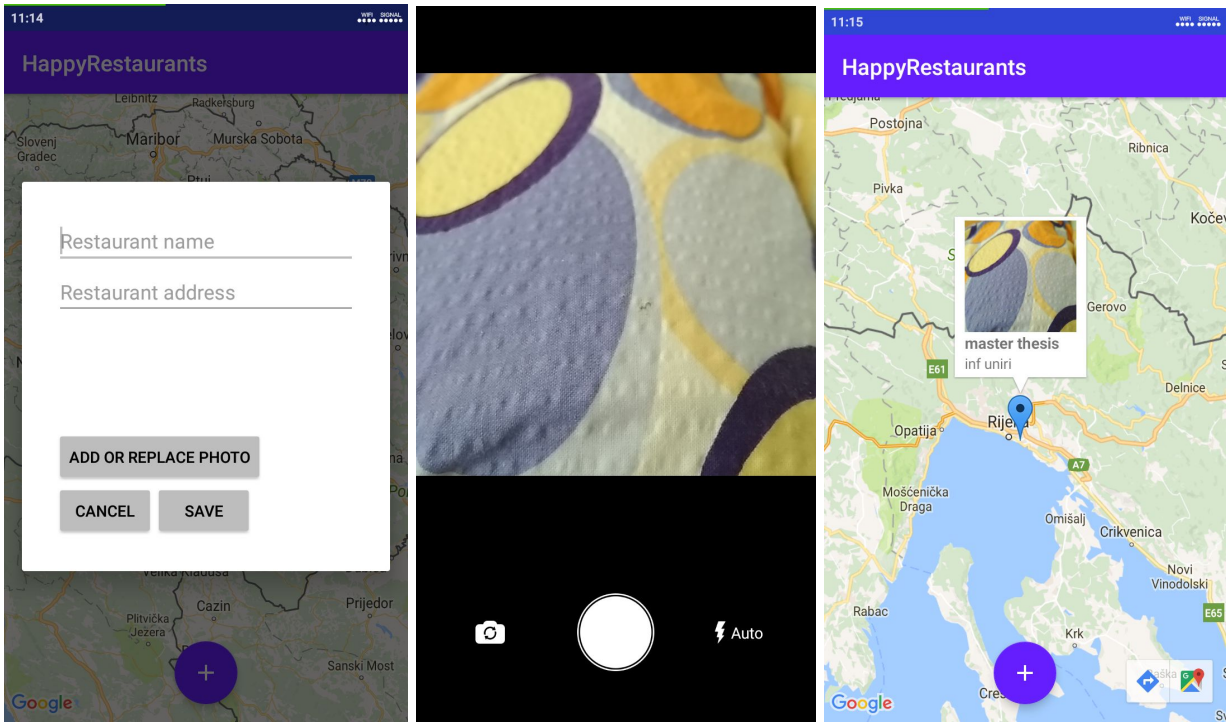
Prikazivanje podataka iz lokalne baze podataka teče istim tokom (metodama) kao i dohvaćanje podataka sa udaljenog poslužitelja - klasa *MainActivity* poziva metodu u *MainPresenter presenteru* koja dalje poziva metodu iz *GetRestaurantsInteractora.* Logika u interaktoru je drugačija pa je ona i opisana. Podaci iz lokalne baze podataka vrše se u metodi *getRestaurantsFromLocalStorage().* Metoda kao rezultat vraća listu restorana preko *Realm* biblioteke. Kreira se nova instanca *Realm* varijable. Upit dohvaća sve podatke iz *Restaurants* modela, kopiraju se dohvaćeni objekti u novu varijablu (pomoću metode *copyFromRealm())* jer se proces odvija u novom pozadinskom procesu te se zatvara instanca. Ako nema rezultata u bazi ili je došlo do greške, prikazuje se prikladna poruka. Dohvaćena lista dolazi u *MainPresenter* koji dalje šalje podatke preko sučelja u *MainActivity.*

```
override fun getRestaurantsFromLocalStorage(): Single<List<Restaurant>> {
    return Single.create<List<Restaurant>> { subscriber ->
        try {
            val realm = Realm.getDefaultInstance()
            val query = realm.where(Restaurant::class.java).findAll()
            val results = realm.copyFromRealm(query)
            realm.close()
            if (results.isEmpty()) {
                subscriber.onError(Throwable("List is empty"))
            } else subscriber.onSuccess(results)
        } catch (e: Exception) {
            subscriber.onError(e)
        }
    }
}
```

Metoda *getRestaurantsFromLocalStorage()*

7.5. Dodavanje novih podataka u bazu podataka

Dodavanje novog restorana vrši se pomoću plutajućeg gumba pri dnu ekrana. Pritiskom na gumb otvara se dialog (slika 5). Dialog se sastoji od imena, adrese i i gumba za dodavanje slike.



Slika 5. Dijalog (lijevo), kamera (sredina), mapa (desno)

Kada korisnik aplikacije klikne na plutajući gumb, pokreće se metoda *onClick()* koja je implementirana sučeljem u klasi *MainActivity.* U metodi se dohvaća korisnikova lokacija (ako je korisnik prije dopustio dozvolu) pomoću lokacijskog upravitelja (eng. *Location Manager*). Lokacijski upravitelj dohvaća lokaciju preko GPS-a ili internetske veze, pa je potrebno pronaći najbolju lokaciju uspoređujući lokacije preko varijable *bestLocation*. Ako je lokacija pronađena, pokreće se metoda *showRestaurantDialog()* koja prikazuje korisniku dijalog za dodavanje restorane. Ako lokacija nije dostupna, ispisuje se prikladna poruka.

```
override fun onClick(p0: View?) {
   locationManager = getSystemService(Context.LOCATION_SERVICE) as
LocationManager
```

```
   if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) ==
PackageManager.PERMISSION_GRANTED) {
       val providers = locationManager.getProviders(true)
       var bestLocation: Location? = null
       for (provider in providers) {
           val l = locationManager.getLastKnownLocation(provider) ?:
continue
           if (bestLocation == null || l.accuracy <
bestLocation.getAccuracy()) {
               bestLocation = l
           }
       }
       if (bestLocation != null) {
           currentLocation = bestLocation
           showNewRestaurantDialog(currentLocation)
       } else showToast(getString(R.string.current_location_error))
   } else showToast(getString(R.string.grant_location_permission))
}
```

Metoda *onClick()*

Metoda *showNewRestaurtantDialog()* prikazuje korisniku dialog za dodavanje restorana (slika 5). Na početku metode inicijaliziraju se pogledi i dodajemo akciju koja se pokreće ako korisnik odabere gumb za dodavanje slike, koji onda zatvara dialog.

```
bAddImage.setOnClickListener {
   newRestaurantDialog.dismiss()
   val startCustomCameraIntent = Intent(this, CameraActivity::class.java)
   startActivityForResult(startCustomCameraIntent,
ADD_RESTAURANT_REQUEST_CODE)
}
```

Metoda za pokretanje kamere

Nakon što je fotografija uslikana, dobiva se lokacija fotografije u metodi *onActivityResult().* Ako je kod zahtjeva jednak traženome i ako nije došlo do nikakve pogreške, u globalnoj varijabli *currentImageUri* dohvaćamo lokaciju fotografije u obliku teksta te se ponovo prikazuje dialog koji postavlja fotografiju restorana.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent) {
    if (resultCode != Activity.RESULT_OK) return
    if (requestCode == MODIFY_RESTAURANT_REQUEST_CODE) {
        val photoUri = data.data
        currentImageUri = photoUri.toString()
        showPlaceInfoDialog(currentRestaurantName, currentRestaurantAddress,
currentRestaurantImagePath)
    } else if (requestCode == ADD_RESTAURANT_REQUEST_CODE) {
        val photoUri = data.data
        currentImageUri = photoUri.toString()
        showNewRestaurantDialog(currentLocation)
    }
    super.onActivityResult(requestCode, resultCode, data)
}
```

Metoda *onActivityResult()*

Nakon što korisnik unese andresu, ime i/ili sliku restoran te odabere gumb za spremanje, dohvaćaju se unesene informacije u nove varijable. Ako su ime i adresa restorana prazni, obavještavamo korisnika da te informacije trebaju biti popunjene. Ako su svi podaci popunjeni, dohvaća se lokacija korisnika, kreira se novi objekt modela *Restaurant* sa popunjenim podacima (parametrima) i pokreće se metoda *addNewRestaurant()* koja se nalazi u klasi *MainPresenter*.

```
val bSave = newRestaurantDialog.findViewById(R.id.bSave) as Button
bSave.setOnClickListener {
    val newName = tvName.text.toString()
    val newAddress = tvAddress.text.toString()
    if (newName.isBlank() && newAddress.isBlank()) {
        showToast(getString(R.string.name_address_blank))
    } else {
        newRestaurantDialog.dismiss()
        val newLongitude = this.currentLocation.longitude
        val newLatitude = this.currentLocation.latitude
        val newImagePath = currentImageUri
        val newRestaurant = Restaurant(newName, newAddress, newLongitude,
newLatitude, true, newImagePath)
        presenter.addNewRestaurant(newRestaurant)
    }
}
```

Metoda za spremanje u lokalnu bazu podataka

Klasa *MainPresenter* poziva interaktor (klasa *SaveRestaurantInteractor)* odosno metodu *saveNewRestaurant()* koja je zaslužna za spremanje podataka u lokalnu bazu. Objekt (model) *Restaurant* je parametar u metodi i sadrži informacije za novi restoran. U ovom slučaju funkcija ne vraća ništa, već samo izvršava radnju pa je stoga tip funkcije *Unit.* Prije spremanja podataka provjerava se postoji li u trenutnoj bazi restoran s istim koordinatama. Ako postoji, trenutni se podaci zamjenjuju s novim podacima te se operacija sprema metodom *insertOrUpdate().* Ako restoran ne postoji, dodaje se u bazu metodom *copyToRealm()*. U slučaju pogreške, korisniku se ispisuje greška kako operacija nije uspješna, dok se u suprotnom ispisuje poruka da je restoran dodan na listu.

```
override fun saveNewRestaurant(restaurant: Restaurant): Single<Unit> {
   return Single.create<Unit> { subscriber ->
       try {
           val realm = Realm.getDefaultInstance()
           realm.beginTransaction()
           val query =
realm.where(Restaurant::class.java).equalTo(RESTAURANT_LONGITUDE_FIELD,
restaurant.longitude).equalTo(RESTAURANT_LATITUDE_FIELD,
restaurant.latitude)
                   .findFirst()
           if (query != null) {
               query.name = restaurant.name
               query.address = restaurant.address
               query.latitude = restaurant.latitude
               query.longitude = restaurant.longitude
               query.imagePath = restaurant.imagePath
               query.isCustom = restaurant.isCustom
               realm.insertOrUpdate(query)
           } else realm.copyToRealm(restaurant)
           realm.commitTransaction()
           subscriber.onSuccess(Unit)
       } catch (e: Exception) {
           subscriber.onError(e)
       }
   }
}
```

Metoda *saveNewRestaurant()*

8. Testiranje

Rad sadrži jedinične testove (eng. *Unit tests)* koji testiraju metode odnosno logiku koda. Korištena arhitektura, Dagger2 i RxJava2 biblioteke omogućuju jednostavno pisanje testova. Testovi su pisani za klasu *MainPresenter* jer je on "most" između pogleda (klasa *MainActivity*) i interkatora (*getRestaurantsInteractor* i *saveRestaurantsInteractor*), pa samim time pokriva cijelu logiku aplikacije. U klasi *BasePresenterTest* nalaze se početne postavke koje nasljeđuju ostale klase.
Anotacija *@RunWith* označava kojom se klasom pokreću testovi. U ovom slučaju, svi se testovi pokreću pomoću biblioteke PowerMock koja osim normalnih jediničnih testova, podržava i testiranje statičnih metoda (npr. *Log.d()*). Pomoću anotacije *@Rule* postavljamo pravila testova, u ovom slučaju koristimo pravila za testiranje biblioteke RxJava. Anotacija *@Before* koristi se kako bi postavili varijable prije pokretanja testova.

```
@RunWith(PowerMockRunner::class)
abstract class BasePresenterTest() {

   @Rule
   @JvmField
   var testSchedulerRule = RxSchedulersOverrideRule()

   @Before
   abstract fun setUp()
}
```

Klasa *BasePresenterTest*

Klasa *RxSchedulersOverrideRule* naslijeđuje klasu *TestRule*. U klasi *MainPresenter* koristi se RxJava biblioteka koja upravlja procesima koji se izvršavaju paralelno. Jedinični testovi se izvršavaju serijski, odnosno jedan za drugim, što znači da se treba nadjačati paralelizam s pravilima koja postavljaju serijska izvršavanja za sve niti (procese).

```
RxJavaPlugins.setIoSchedulerHandler { Schedulers.trampoline() }
RxJavaPlugins.setComputationSchedulerHandler { Schedulers.trampoline() }
RxJavaPlugins.setNewThreadSchedulerHandler { Schedulers.trampoline() }
RxAndroidPlugins.setInitMainThreadSchedulerHandler {
Schedulers.trampoline() }
```

Postavljanje pravila za niti

U klasi *MainPresenterTest* nalaze se jedinični testovi za klasu *MainPresenter.* Prvi vrhu klase nalaze se varijable koje se koriste kasnije u testovima. Pomoću anotacije *@Mock* kreiramo objekte koji zapravo simuliraju prave objekte.

```
private lateinit var presenter: MainPresenter
@Mock
private lateinit var view: Main.View
@Mock
private lateinit var getRestaurants: Interactor.GetRestaurants
@Mock
private lateinit var saveRestaurants: Interactor.SaveRestaurants
@Mock
private lateinit var restaurant: Restaurant
@Mock
private lateinit var restaurantsList: List<Restaurant>
```

Varijable u klasi *MainPresenterTest*

U metodi *setUp()* inicijaliziraju se objekti i varijable. Metoda se pokreće prije izvršavanja testova. Objekt *MainPresenter* nema anotaciju *@Mock* pa je potrebna inicijalizacija. Pomoću biblioteke PowerMockito omogućavamo izvršavanje statične klase *Log te* inicijaliziramo varijable koje se koriste u jediničnim testovima.

```
override fun setUp() {
    presenter = MainPresenter(view, getRestaurants, saveRestaurants)
    PowerMockito.mockStatic(Log::class.java)

    name = "oldName"
    newName = "newName"
    address = "newAddress"
    imagePath = "newImagePath"
}
```

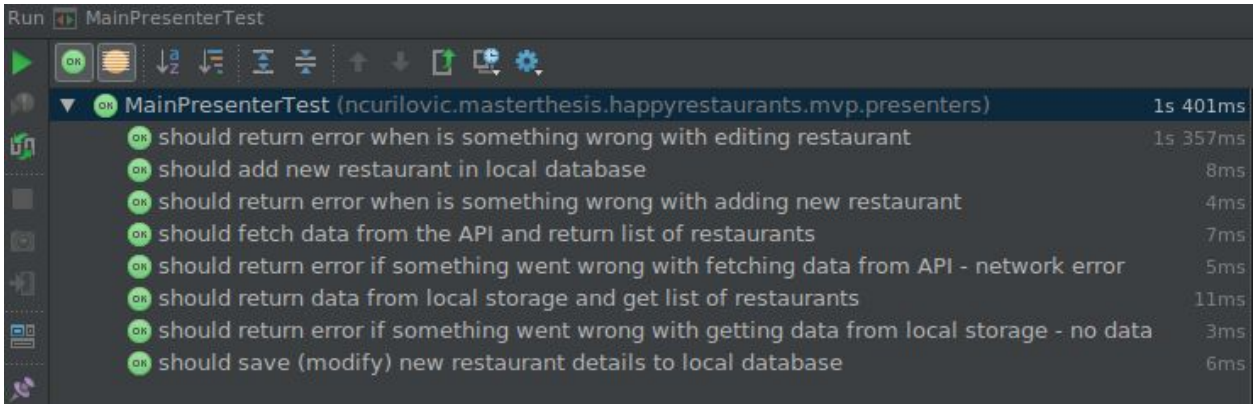
Metoda *setUp()*

Anotacija *@Test* služi za testiranje metoda. Kotlin omogućuje nazivanje metoda s razmakom te je naziv metoda koje testiraju logiku zapravo opis radnje. Pomoću statične metode *whenever* opisuje se plan izvršavanja (tok funkcija), dok metoda *verify* služi za provjeravanje funkcije. Prije metode *verify* pozove se funkcija presentera. U nastavku slijedi prikaz jediničnih testova koji preuzimaju podatke s udaljenog poslužitelja te prikazuju korisniku listu restorana kada je operacija uspješna odnosno neuspješna.

```
@Test
fun `should fetch data from the API and return list of restaurants`() {

whenever(getRestaurants.fetchRestaurantsFromAPI()).thenReturn(Single.just(r
estaurantsList))

   presenter.getRestaurantsFromAPI()

   verify(view).showRestaurants(restaurantsList)
}

@Test
fun `should return error if something went wrong with fetching data from
API - network error`() {

whenever(getRestaurants.fetchRestaurantsFromAPI()).thenReturn(Single.error(
Throwable()))

   presenter.getRestaurantsFromAPI()

   verify(view).showNoDataError()
}
```

Prikaz jediničnih testova

U projektu je sveukupno osam jediničnih testova koji testiraju četiri metode. Svaka metoda ima dva testa jer se testiraju dva stanja - uspješno i neuspješno izvršavanje metode.



Slika 6. Prikaz svih jediničnih testova (izvršenih)

8.1 Testiranje bez Dagger2 biblioteke

U nastavku usporediti ćemo testiranje bez Dagger2 biblioteke. Važno je napomenuti kako su veće razlike u kodu nastale u drugim klasama a ne u samoj klasi koju testiramo (*MainPresenterTest*) jer se te razlike odnose na samu arhitekturu aplikacije i to na sve komponente u arhitekturi.

U klasi *MainActivity* ručno je kreiran objekt presentera. Metoda za definiciju modula više nije potrebna.

```
    lateinit var presenter: Main.Presenter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        presenter = MainPresenter(this)
          ...
          }
    ...
    override fun injectDependencies(appComponent: AppComponent) {
//        appComponent.plus(MainModule(this))
//                .inject(this)
    }
```

Klasa *MainActivity*

Klasa *MainPresenter* više ne sadrži Dagger2 anotaciju za ubrizgavanje (*@Inject*). Samim time, ručno su definirane varijable za interaktore te je dodan sekundarni konstruktor koji nam je potreban za stvaranja objekta interaktora kod testiranja.

```
class MainPresenter constructor(var view: Main.View)
    : BasePresenter(), Main.Presenter {

    private var getRestaurants: Interactor.GetRestaurants =
```

```
GetRestaurantsInteractor()

   private var saveRestaurants: Interactor.SaveRestaurants =
SaveRestaurantsInteractor()

   constructor(view: Main.View, getRestaurants: Interactor.GetRestaurants,
saveRestaurants: Interactor.SaveRestaurants) : this(view) {
       this.getRestaurants = getRestaurants
       this.saveRestaurants = saveRestaurants
   }
...
```

Klasa *MainPresenter*

Klase *GetRestaurantsInteractor* i *SaveRestaurantsInteractor* također više nemaju anotaciju za ubrizgavanje te dodatno kreirana varijabla za API klijent (*Retrofit*).

```
class GetRestaurantsInteractor : Interactor.GetRestaurants {

   private var apiService: ApiService = Retrofit.Builder()
           .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
           .addConverterFactory(GsonConverterFactory.create())
           .baseUrl(BuildConfig.API_KEY)
           .build()
           .create<ApiService>(ApiService::class.java)
...
```

Klasa *GetRestaurantsInteractor*

Klasa *MainPresenterTest* od sada kreira objekt presentera pomoću interaktora te je to ujedno i jedina promjena u toj klasi.

```
override fun setUp() {
   presenter = MainPresenter(view, getRestaurants, saveRestaurants)
   ...
   }
```

Metoda *setUp()*

9. Zaključak

U radu je opisana MVP arhitektura i korištenje najpopularnijih biblioteka koje olakšavaju izradu Android aplikacija. Pomoću MVP arhitekture “razdvojili” smo logiku i prikazivanje sučelja na tri sloja, prikazali preuzimanje podataka preko udaljenog poslužitelja, korištenje objektne baze podataka te dohvaćanje pozicije korisnika. Iako neke biblioteke zahtjevaju dosta podešavanja i nisu jednostavne za implementaciju (poput Dagger2 biblioteke) te čine projekt kompleksnijim, smatram kako su korištene biblioteke i arhitektura postali standard za izradu kompleksnijih, skalabilnih i modularnih aplikacija koje se jednostavno testiraju.

Prednosti korištenja Dagger2 biblioteke:

- Modularan kod
- Lako testiranje (simuliranje pravih objekata)
- Generira kod koji se lako može pratiti
- Eliminira dodatan kod u samoj logiki koji služi samo za testiranje (npr. definiranje dodatnih kontruktora klasa)
- Korištenje generiranih objekata kroz cijeli životni vijek aplikacije

Nedostaci korištenja Dagger2 biblioteke:

- Povećava kompleksnost aplikacije
- Teška implementacija
- Teško praćenje grešaka pri generiranju koda
- Sporije pokretanje aplikacije (kompleksnijih projekata)

10. Literatura

[1.] Kotlin (programming language), Wikipedia, the free encyclopedia,
https://en.wikipedia.org/wiki/Kotlin_(programming_language)

[2.] Android Studio (IDE), Android developer, official Android documentation,
https://developer.android.com/studio/index.html

[3.] Dagger2 library, Github repository,
https://github.com/google/dagger

[4.] RxJava2 library, Github repository,
https://github.com/ReactiveX/RxAndroid

[5.] Retrofit library, Github repository,
https://github.com/square/retrofit

[6.] TedPermission library, Github repository,
https://github.com/ParkSangGwon/TedPermission

[7.] SquareCamera library, Github repository,
https://github.com/boxme/SquareCamera

[8.] Google Maps, Android developer, official Android documentation,
https://developers.google.com/maps/documentation/android-api/

[9.] MVP, Wikipedia, the free encyclopedia,
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter

[10.] Android testing, Github blog,
https://fedepaol.github.io/blog/2016/08/27/android-mvp-testing/

[11.] LVVM (compiler), Wikipedia, the free encyclopedia,
https://en.wikipedia.org/wiki/LLVM

[12.] Java package, Wikipedia, the free encyclopedia,
https://en.wikipedia.org/wiki/Java_package

[13.] Unit testing, Wikipedia, the free encyclopedia,
https://en.wikipedia.org/wiki/Unit_testing

[14.] Instrumentation testing, Android developer, official Android documentation,
https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html

[15.] MVC, Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[16.] Dependency injection, TechYourChange blog,
https://www.techyourchance.com/dependency-injection-android/

[17.] Gradle build system, official website,
https://gradle.org/

[18.] jUnit testing tools, official website,
http://junit.org/junit4/

[19.] Realm database, official website,
https://realm.io/docs/

[20.] Powermock library, Github repository,
https://github.com/powermock/powermock

[21.] Remember library, Github repository,
https://github.com/tumblr/Remember

[22.] Mockito, official website,
http://site.mockito.org/

[23.] Support library, Android developer, official Android documentation,
https://developer.android.com/topic/libraries/support-library/index.html

[24.] Play Services, Android developer, official Android documentation,
https://developer.android.com/google/index.html

[25.] JSON, Wikipedia, the free encyclopedia,
https://en.wikipedia.org/wiki/JSON