Mobilna aplikacija za unaprijeđivanje organizacije i produktivnosti

Junaković, Bruno

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:376569

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-16**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

Mobilna aplikacija za unaprjeđivanje organizacije i produktivnosti

Završni rad

Bruno Junaković

Osijek, 2021.

SADRŽAJ

1. UVOD 1

1.1. Zadatak završnog rada 1

2. TEHNOLOGIJA I ALATI 2

2.1. Android studio 2

2.2. Kotlin 2

2.3. Room i Kotlin Coroutines biblioteke 3

2.3.1. Room 3

2.3.2. Kotlin Coroutines 5

2.4. RecyclerView 6

2.5. Koin dependency injection framework 8

2.6. MVVM obrazac arhitekture 10

3. MOBILNA APLIKACIJA 13

3.1. Mockup i dijagram toka 13

3.1.1. Mockup 13

3.1.2. Dijagram toka 14

3.2. Izrada mobilne aplikacije 15

3.2.1. Korisničko sučelje 15

3.2.2. Struktura mobilne aplikacije i protok podataka 21

3.2.3. Aktivnosti i fragmenti 25

4. ZAKLJUČAK 32

LITERATURA 33

SAŽETAK 34

ABSTRACT 35

ŽIVOTOPIS 36

1. UVOD

Problem organizacije prisutan je svuda oko nas, od „utrke“ s vremenom kako stići obaviti sve naše obveze do njihove raspodjele te pitanja „što napraviti prvo?“ ili češće poznato „Odakle krenuti?“. Ovaj završni rad bavi se upravo tim problemom i daje rješenje u obliku Android mobilne aplikacije.

Izrada mobilne aplikacije za unaprjeđivanje organizacije i produktivnosti zahtjeva poznavanje jednog od programskih jezika za razvoj Android mobilna aplikacija, u slučaju ovog završnog rada to je Kotlin, poznavanje i primjenu objektno-orijentiranih načela za razvoj programske podrške, sposobnost korištenja Android *persistance* biblioteka kao što su Room, Coroutines i Koin, SQLite jezika za izgradnju i upravljanje lokalnim bazama podataka te sposobnost rada s procesima na više niti. Kako bi mobilna aplikacija bila strukturirana i njena arhitektura pravilno složena, potrebno je poznavanje i primjena obrazaca arhitekture kao što su: *Model-View-ViewModel* (MVVM), *Model – View – Presenter* (MVP), *Model-View-Controller* (MVC), u ovom radu korišten je MVVM. Također je potrebno poznavanje čestih problema organizacije te adekvatnih i učinkovitih rješenja za njih i poznavanje metoda za poboljšavanje produktivnosti pojedinca.

1.1. Zadatak završnog rada

U teorijskom dijelu rada potrebno je proučiti i opisati tehnologije za izradu mobilnih aplikacija kao i profesionalne sustave koji se bave organizacijom zadataka, vođenjem timova, organizacijom događaja u svrhu povećanja produktivnosti djelatnika, timova ili pojedinaca. U praktičnom dijelu rada potrebno je izraditi mobilnu aplikaciju koja će imati funkcionalnosti organizacije aktivnosti, kategoriziranje i prioritiziranje, postavljanje podsjetnika, obavještavanje korisnika, izradu predložaka i sl. Potrebno je omogućiti spremanje informacija u bazu podataka kao i čitanje, pisanje i uređivanje istih.

2. TEHNOLOGIJA I ALATI

2.1. Android studio

Android studio službena je integrirana razvojna okolina za Google Android operacijski sustav, građena na bazi JetBrains IntelliJ IDEA programske podrške. Dizajniran i namijenjen je posebno za razvoj Androida. Android studio najavljen je na Google I/O konferenciji 16.svibnja 2013. godine, verzija 0.8 (beta) puštena je u lipnju 2014. godine, a prva stabilna verzija (1.0) izašla je u prosincu 2014. godine. Zadnja stabilna inačica Android studija je 4.2.2 pod nazivom Arctic Fox [1].

2.2. Kotlin

Kotlin je opće namjenski, besplatni programski jezik otvorenog koda, često smatran nasljednikom Jave. Dizajniran je za Java virtualni stroj i Android (koji koristi kombinaciju objektno-orijentiranog i funkcionalnog programiranja). Nastaje 2010. godine u JetBrains tvrtki koja je također zaslužna za IntelliJ IDEA razvojnu okolinu, a od 2012. godine otvorenog je koda [1]. Neke od prednosti i beneficija koje nam donosi Kotlin u razvoju Android mobilnih aplikacija su:

- Manje koda i veća čitljivost – štedi vrijeme kod pisanja koda kao i kod čitanja i razumijevanja tuđeg koda.
- Podržan je u Android Jetpack i ostalim bibliotekama – omogućava dodavanje značajki Kotlina u već postojeće Android biblioteke.
- Interoperabilnost s Javom – moguće je koristiti Kotlin uz Javu u mobilnim aplikacijama bez potrebe migriranja cijelog koda na Kotlin.
- Podržava razvoj na više platformi – Osim za razvoj Android mobilnih aplikacija, koristi se također za razvoj web i iOS mobilnih aplikacija.
- Sigurnost koda – manja količina koda te njegova čitljivost, također i neke od njegovih značajki kao sigurnost null-pokazivača rezultiraju smanjenjem grešaka u kodu. Ostatak grešaka ispravlja kompilator.
- Lako učenje – pogotovo za Java developere.
- Velika zajednica – prema Google-u više od 60% mobilnih aplikacija na listi top 1000 na Play Trgovini koriste Kotlin [3].

2.3. Room i Kotlin Coroutines biblioteke

2.3.1. Room

Mobilne aplikacije koje rukuju netrivijalnim količinama strukturiranih podataka imaju veliku korist od čuvanja tih podataka lokalno. Najčešći takav slučaj je korištenje međuspremnika za pohranu relevantnih podataka kako bi korisnik imao mogućost pregledavati podatke i kada nije spojen na mrežu.

Room biblioteka pruža sloj apstrakcije nad SQLite-om te omogućava pristup i manipulaciju bazom podataka dok spreže puni potencijal SQLitea, drugim riječima Room pruža sljedeće beneficije:

- Verifikaciju vremena prevođenja SQL upita.
- Pogodne anotacije koje smanjuju *boilerplate* kod koji je sklon ponavljanju i često sadrži greške.
- Pojednostavljene oblike putanje migracije baze podataka [4].

Zbog navedenih beneficija preporuča se korištenje Room-a umjesto direktnog korištenja SQLite APIa. Programski kod 2.1 prikazuje dodavanje potrebnih ovisnosti za korištenje Room biblioteke. Ovisnosti se dodaju u Gradle datoteku na nivou modula. Potrebno je definirati verziju biblioteke (preporuča se korištenje najnovije stabilne verzije), koja se kasnije koristi u linijama koda same implementacije. Na taj način se može lako ažurirati verzija biblioteke promjenom verzije samo u definiciji, što je lakši i sigurniji način od mijenjanja verzije u svakoj liniji koda posebno. Postoje obvezne ovisnosti, bez kojih se biblioteka ne može koristiti, ali ima i nekih opcionalnih ovisnosti koje omogućuju suradnju s ostalim bibliotekama kao što su Coroutine, RxJava2, RxJava3, Kotlin Extensions, Guava i pomagala za testiranje koda. Nakon dodavanja ovisnosti ili bilo kakve promjene potrebno je ponovno sinhronizirati Gradle datoteku s projektom, u protivnom neće doći do uvedenih izmjena.

```
dependencies {
    def room_version = "2.3.0"

    implementation("androidx.room:room-runtime:$room_version")
    annotationProcessor "androidx.room:room-compiler:$room_version"

    kapt("androidx.room:room-compiler:$room_version")

    ksp("androidx.room:room-compiler:$room_version")

    implementation("androidx.room:room-ktx:$room_version")

    implementation "androidx.room:room-rxjava2:$room_version"

    implementation "androidx.room:room-rxjava3:$room_version"

    implementation "androidx.room:room-guava:$room_version"

    testImplementation("androidx.room:room-testing:$room_version")
}
```

Programski kod 2.1 Dodavanje potrebnih ovisnosti za korištenje Room biblioteke

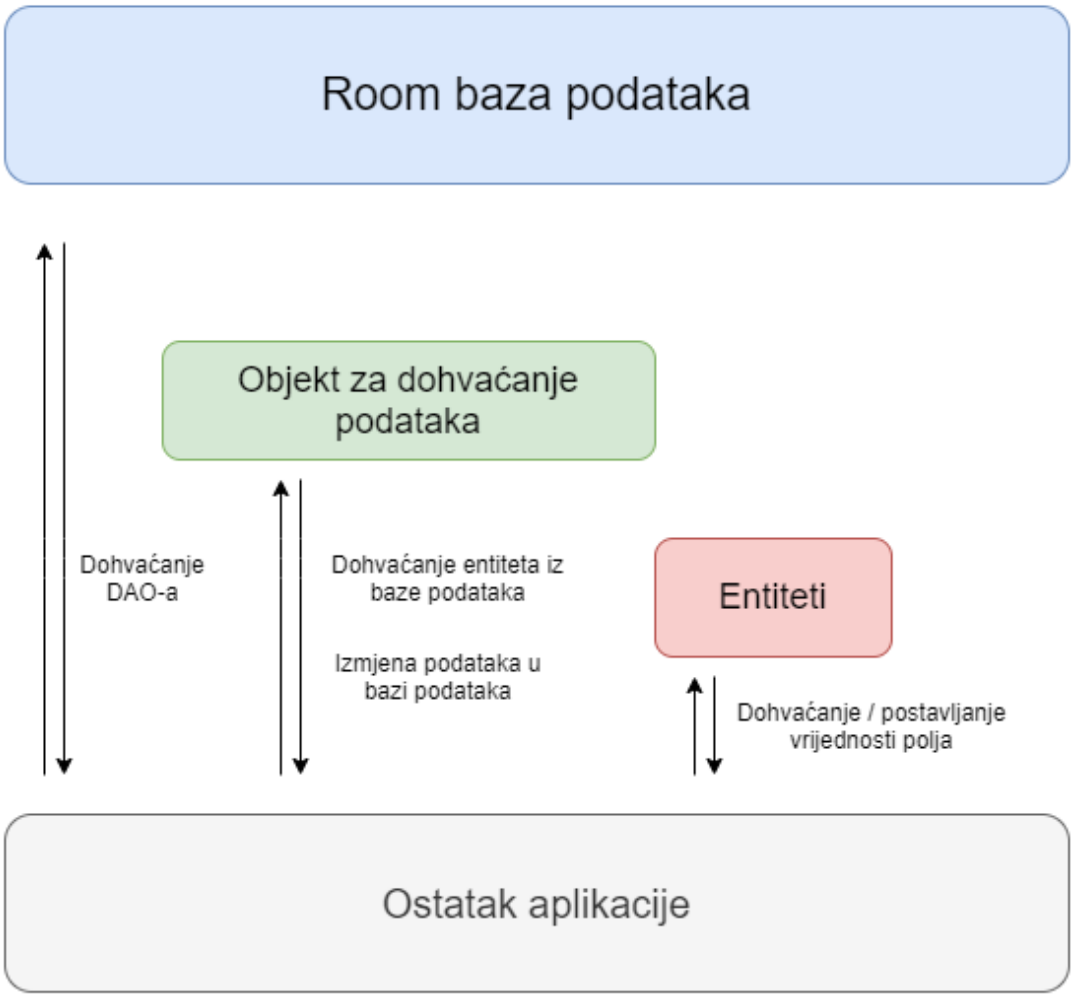
Primarne komponente Room baze podataka su:

- Klasa baze podataka (engl. *database class*) – sadrži bazu podataka i služi kao glavna točka pristupa do pohranjenih podataka.
- Podatkovni entiteti (engl. *data entities*) – predstavljaju tablice baze podataka.
- Objekti pristupa podatcima (engl. *Data access objects* (DAO)) - pruža metode koje mobilna aplikacija koristi kao upite i naredbe za dohvaćanje i manipulaciju bazom podataka[4][5].

Klasa baze podataka pruža mobilnoj aplikaciji instance objekata za pristup podatcima koje mobilna aplikacija koristi za dohvaćanje podataka iz baze podataka kao instance objekata entiteta. Mobilna aplikacija također može koristiti definirane entitete za ažuriranje redaka odgovarajuće tablice ili stvaranje novih [4].

Slika 2.1 prikazuje arhitekturu i odnose osnovnih komponenata Room biblioteke. Objekt Room baze podataka instancira se, poželjno kao *singleton*, ako se radi sa više procesnih niti potrebno je obratiti pažnju i osigurati da se ne stvaraju dodatne nepotrebne instance na ostalim nitima, to se može postići korištenjem *LOCKa* unutar *singletona.* Objekt baze podataka koristi se za dohvaćanje objekata za pristupanje podatcima. Objekt za pristupanje podatcima koristi se za dohvaćanje

entiteta baze podataka kao i za pohranjivanje promjena u bazu podataka. Entitet predstavlja tablice baze podataka i njima postavljamo vrijednosti određenih polja u bazi podataka[5].



Slika 2.1Arhitektura Room komponenti

2.3.2. Kotlin Coroutines

Asinhrono programiranje važan je dio razvoja *server-side*, desktop i mobilnih aplikacija, ono korisniku pruža fluidno i ugodno iskustvo. Kotlin rješava taj problem prilično fleksibilnim načinom – pružanjem podrške za *coroutines* biblioteku na nivou jezika i delegacijom većine funkcionalnosti na biblioteke[6].

Na Androidu *coroutine* pomažu upravljanjem dugotrajnih procesa koji bi u drugom slučaju mogli blokirati glavnu nit i uzrokovati prestanak reagiranja mobilne aplikacije. *Coroutine* su obrasci koji omogućuju pojednostavljenje koda koji se izvodi asinhrono. Konceptom su slične niti, po tome što

uzimaju blok koda koji se izvodi istovremeno s ostatkom koda. Bitno je naznačiti da nisu vezane za određenu nit, imaju mogućnost suspendirati izvođenje na jednoj niti i nastaviti ga na drugoj. Coroutine su preporučeno rješenje za asinhrono programiranje na Androidu zbog sljedećih beneficija:

- Lakoća – moguće je izvođenje više *coroutina* na jednoj niti zbog podržanosti suspendiranja, koje ne blokira nit gdje se izvodi *coroutina*. Suspendiranje također štedi memoriju.
- Smanjen broj curenja memorije .
- Ugrađena podrška otkazivanja – otkazivanje je propagirano automatski kroz hijerarhiju *coroutina* koje se izvode.
- Integracija *Jetpacka* – mnoge *Jetpack* biblioteke uključuju proširenja koja podržavaju *coroutine[7]*.

Programski kod 2.2 prikazuje dodavanje potrebnih ovisnosti za korištenje Kotlin *coroutines* biblioteke. Nakon dodavanja ovisnosti u Gradle datoteku na bazi modula potrebno je ponovno sinkronizirati Gradle datoteku s projektom.

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
android:1.3.9")
}
```

Programski kod 2.2 Dodavanje potrebnih ovisnosti za korištenje kotlin coroutines biblioteke

2.4. RecyclerView

RecyclerView biblioteka olakšava prikaz velikih setova podataka. Potrebno je predati podatke i definirati izgled svake stavke, a *RecyclerView* dinamično stvara potrebne elemente tako što „reciklira“ pojedine elemente ponovno iskorištavajući *view* za njihovo stvaranje. Takav način stvaranja novih elemenata znatno poboljšava preformansu i reaktivnost mobilne aplikacije, a u isto vrijeme i smanjuje potrošnju baterije uređaja[8].

Iako RecyclerView biblioteka može biti prilično korisna, potrebno je pripaziti na njeno ispravno korištenje kako bi kod radio, posebno treba obratiti pozornost na pravilno postavljanje klasa Adaptera i *ViewHoldera*.

Programski kod 2.3 prikazuje dodavanje potrebnih ovisnosti za korištenje RecyclerView biblioteke. Nakon dodavanja ovisnosti u Gradle datoteku na bazi modula potrebno je ponovno sinkronizirati Gradle datoteku s projektom.

```
dependencies {
    implementation("androidx.recyclerview:recyclerview:1.2.1")

    implementation("androidx.recyclerview:recyclerview-
selection:1.1.0")
}
```

Programski kod 2.3 Dodavanje potrebnih ovisnosti za korištenje RecyclerView biblioteke [9]

Programski kod 2.4 prikazuje implementaciju Adapter i ViewHolder klasa za RecyclerView biblioteku. Prvi korak je izrada *layout* datoteka koje se koriste kao predložak za prikaz podataka u RecyclerViewu. Nakon toga slijedi implementacija Adapter i Viewholder klasa, te klase međusobno surađuju te definiraju način prikaza podataka u RecyclerViewu. ViewHolder djeluje kao omotač oko *viewa* , on sadrži *layout* za individualne podatke liste koju RecyclerView prikazuje. Adapter stvara objekte ViewHolder klase po potrebi te postavlja vrijednosti za te *viewove* . Taj proces povezivanja *viewa* s podatcima naziva se vezanje (engl. *binding*).

Nakon definiranja Adapter klase potrebno je prepisati (engl.*override*) tri ključne metode:

- onCreateViewHolder() – RecyclerView poziva ovu metodu pri svakom stvaranju novog ViewHoldera, metoda se koristi za stvaranje i inicijalizaciju ViewHoldera i njemu vezanog *viewa*. Metoda onCreateViewHolder() ne postavlja sadržaj *viewa.*
- onBindViewHolder() – RecyclerView poziva ovu metodu kako bi povezao ViewHolder s podatcima, ona dohvaća odgovarajuće podatke i koristi ih za popunjavanje *layouta* ViewHoldera. Za primjer, ako RecyclerView prikazuje listu zadataka, ova metoda pronalazi odgovarajući zadatak u listi zadataka i njime popunjava TextView u ViewHolderu.
- getItemCount() – RecyclerView poziva ovu metodu za dohvaćanje veličine seta podataka (najčešće je to nekakva lista) te vraća broj podataka koji se nalaze u tom setu kao cjelobrojni tip. RecyclerView koristi ovu metodu za određivanje koliko podataka treba prikazati[8].

```
class CustomAdapter(private val dataSet: Array<String>) :
        RecyclerView.Adapter<CustomAdapter.ViewHolder>()
{

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {

        val textView: TextView

        init {
            textView = view.findViewById(R.id.textView)
        }
    }

    override fun onCreateViewHolder(viewGroup: ViewGroup, viewType:
Int): ViewHolder {

        val view = LayoutInflater.from(viewGroup.context)
                .inflate(R.layout.text_row_item, viewGroup, false)

        return ViewHolder(view)
    }

    override fun onBindViewHolder(viewHolder: ViewHolder, position:
Int) {

        viewHolder.textView.text = dataSet[position]
    }

    override fun getItemCount() = dataSet.size

}
```

Programski kod 2.4 Prikaz načina implementacije Adapter i ViewHolder klasa za RecyclerView [8]

2.5. Koin dependency injection framework

Koin je Kotlin biblioteka za ubrizgavanje ovisnosti, slična Dagger biblioteci, no jednostavnija i lakša za korištenje. Ubrizgavanje ovisnosti koncipirano je po principu inverzije kontrole koji nalaže da bi klasa trebala dobivati ovisnosti iz vana, što znači da klasa ne bi trebala instancirati drugu klasu, već primati instance od strane konfiguracijske klase. Ubrizgavanje ovisnosti preporučeno je koristiti zbog sljedećih beneficija:

- Olakšava upravljanje kompleksnim ovisnostima.
- Olakšava *unit* testiranje omogućavajući predaju svih ovisnosti iz vana.
- Lako upravljanje životnim ciklusom objekta.

Za korištenje Koina potrebno je znanje osnovne terminologije Koin biblioteke:

- *module* – stvara modul u Koinu koji se koristi za pružanje svih ovisnosti.
- *single* – stvara *singleton* koji se može koristiti u cijeloj mobilnoj aplikaciji kao jedna instanca.
- *factory* – pruža tvornicu objekata koja stvara novu instancu pri svakom ubrizgavanju.
- *get()* – metoda koja se koristi kao konstruktor klase te za pružanje potrebnih ovisnosti[10].

Programski kod 2.5 prikazuje dodavanje potrebnih ovisnosti za korištenje Koin biblioteke. Nakon dodavanja ovisnosti u Gradle datoteku na bazi modula potrebno je ponovno sinkronizirati Gradle datoteku s projektom.

```
dependencies
{

    implementation"org.koin:koin-androidx-viewmodel:$koin_version"

    testImplementation "org.koin:koin-test:$koin_version"

    implementation"org.koin:koin-android:$android_version"

    implementation"org.koin:koin-android-viewmodel:$android_version"
}
```

Programski kod 2.5 Dodavanje potrebnih ovisnosti za korištenje Koin biblioteke

Programski kod 2.6 prikazuje dodavanje potrebnog koda za korištenje Koina u klasu mobilne aplikacije, kao i dodavanje potrebnih modula. Klasa mobilne aplikacije mora nasljeđivati klasu Application(). Također, bitno je da kod za korištenje Koina bude unutar *onCreate()* metode iz razloga što se ona pokreće prva pri pokretanju mobilne aplikacije. Kao što je prikazano u programskom kodu 2.6, moduli su smješteni unutar *arrayListOf()* metode. Potrebni su moduli za *view model* (viewModelModule), repozitorij (repositoryModule), bazu podataka (databaseModule) i spremnik (storageModule).

```
class Zavrsni : Application() {

    override fun onCreate() {
        super.onCreate()


        startKoin {
            androidContext(this@Zavrsni)
            modules(
                arrayListOf(
                    viewModelModule,
                    repositoryModule,
                    databaseModule,
                    storageModule
                )
            )
        }

    }
}
```

Programski kod 2.6 Dodavanje potrebnog koda za korištenje Koin biblioteke

2.6. MVVM obrazac arhitekture

Softverski dizajn obrazac je rješenje za problem koji se često pojavljuje. Čest problem pri razvijanju mobilne aplikacije je njena arhitektura, potrebno je izbjegavanje stavljanja logike unutar aktivnosti, fragmenata i pogleda, kako nalaže *separation of concerns* načelo. Dobrom organizacijom i strukturiranjem koda olakšavamo njegovo održavanje, izmjenu, nadogradnju i slično[11]. MVVM (Model – View – ViewModel) je obrazac koji rješava taj problem, nalaže odvajanje prezentacijske logike (pogleda i korisničkog sučelja) od poslovne logike mobilne aplikacije.

Slojevi MVVM-a:

- Model – sloj odgovoran za apstrakciju podatkovnih izvora. Model i ViewModel rade zajedno pri dohvaćanju i pohrani podataka.
- View – sloj odgovoran za obavještavanje ViewModela o akcijama korisnika. Ovaj sloj promatra ViewModel i ne sadrži nikakvu logiku mobilne aplikacije.
- ViewModel – pruža podatke za specifičnu komponentu korisničkog sučelja (fragment ili aktivnost) i sadrži logiku upravljanja podatcima i komuniciranja s modelom. Služi kao poveznica između modela i pogleda[12].

Programski kod 2.7 prikazuje dodavanje potrebnih ovisnosti za korištenje MVVM obrasca. Nakon dodavanja ovisnosti u Gradle datoteku na bazi modula potrebno je ponovno sinkronizirati Gradle datoteku s projektom

```
dependencies {

    implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.3.1"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1"

}
```

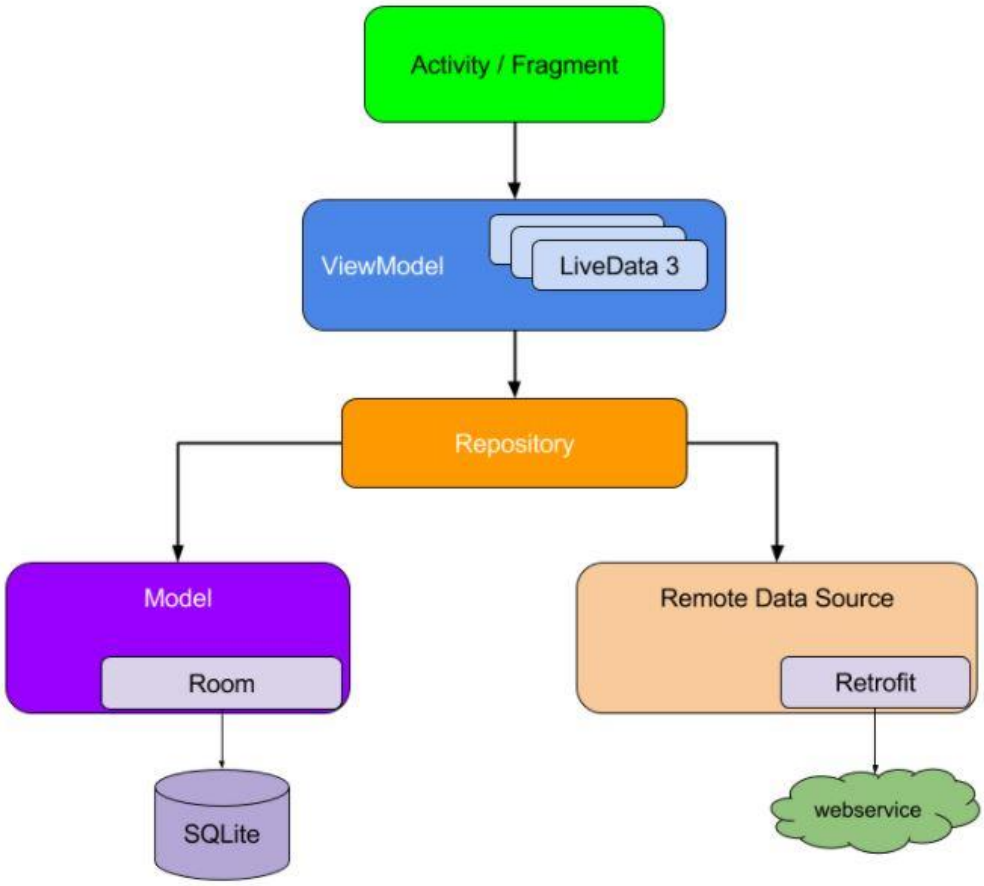
Programski kod 2.7 Dodavanje potrebnih ovisnosti za MVVM obrazac

Programski kod 2.8 prikazuje dodavanje potrebnog koda u Gradle datoteku na bazi modula kako bi se omogućio *data* i *view binding*.

```
android {

    buildFeatures {
        dataBinding true
        viewBinding true
    }
}
```

Programski kod 2.8 Dodavanje potrebnog koda za omogućavanje *data* i *view bindinga*

Slika 2.2 prikazuje dijagram interakcije među slojevima MVVMa. Na njoj vidimo još jedan dodatni sloj za pohranu i dohvaćanje podataka koji povezuje ViewModel i Model, a to je repozitorij.



Slika 2.2 Dijagram komunikacije među slojevima MVVM arhitekture[13]

Kao što je vidljivo na dijagramu sa slike 2.2 *view modelu* nije bitno dohvaćaju li se podatci iz lokalne baze podataka ili sa udaljenog izvora podataka, već taj dio komunikacije obavlja repozitorij. Između objekta za dohvaćanje podataka i repozitorija može se dodati još jedan sloj apstrakcije kao što je dodan u mobilnoj aplokaciji završnog rada, a to je spremnik. Kod *view modela* na dijagramu se vidi i *liveData*, *liveData* služi kako bi se fragment ili aktivnost mogli pretplatiti, to jest promatrati podatke i njihove promjene iz *view modela*. Uvođenjem *live date* također se omogućuje postavljanje proizvoljnih akcija koje će se pokretati pri svakoj promjeni podataka *live date*.

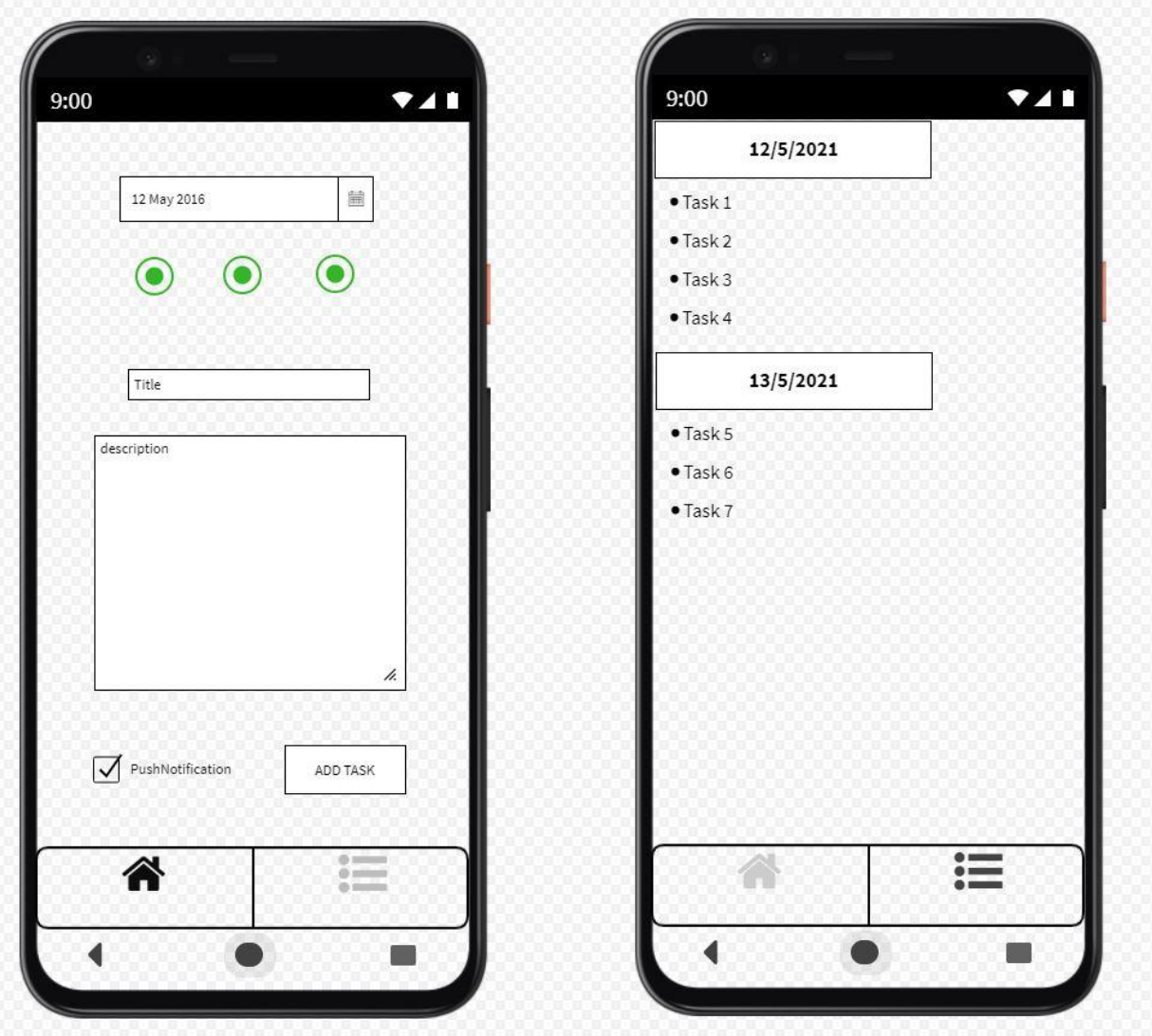
3. MOBILNA APLIKACIJA

3.1. Mockup i dijagram toka

3.1.1. Mockup

Mockup je statički dizajn mobilne aplikacije, cilj *mockupa* za dizajn je prikazati okvirni izgled aktivnosti ili fragmenata mobilne aplikacije sa komponentama koje oni uključuju. *Mockup* ne mora biti potpuno točan niti ima funkcionalnost, već služi kao smjernica pri izradi dizajna same mobilne aplikacije, obično sadrži puno *placeholdera*.

Slika 3.1 prikazuje *mockup* mobilne aplikacije, na njemu se vidi okvirni dizajn, elementi koje mobilna aplikacija sadrži, traka navigacije i prikaz dva fragmenta.

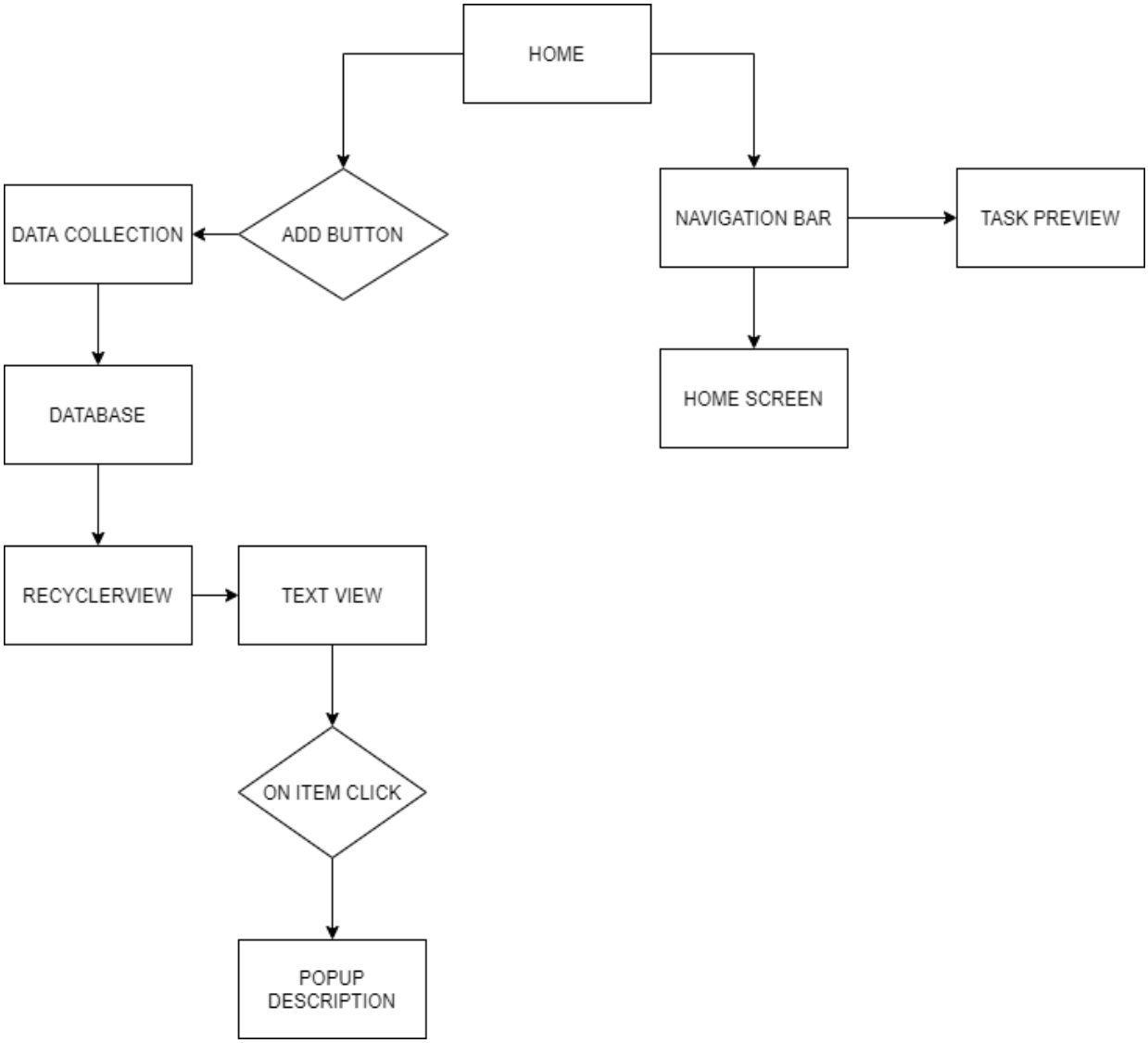


Slika 3.1 prikaz mockupa za dizajn mobilne aplikacije

3.1.2. Dijagram toka

Dijagrami toka su važan dio izrade mobilne aplikacije, iako nije nužan korak, izrada dijagrama toka znatno olakšava planiranje izrade mobilne aplikacije. Kod kompliciranijih mobilnih aplikacija s puno funkcionalnosti gotovo su nužni, vizualno prikazuju procese i odnose komponenti mobilne aplikacije te doprinose izradi same logike mobilne aplikacije.

Slika 3.2 Prikazuje dijagram toka mobilne aplikacije, iz njega se vidi kako radi navigacija, dodavanje novih podataka u bazu, dohvaćanje podataka iz baze u RecyclerView i funkcija klika na bilo koji TextView liste.



Slika 3.2 Dijagram toka mobilne aplikacije

3.2. Izrada mobilne aplikacije

3.2.1. Korisničko sučelje

Korisničko sučelje mobilne aplikacije napravljeno je tako da bude jednostavno za korištenje i razumljivo korisniku, a rađeno je u XMLu. XML (*extensible markup language*) je *markup* jezik kojim se definira izgled komponenata korisničkog sučelja. Mobilna aplikacija ima jednu aktivnost (MainActivity) i dva fragmenta (HomeFragment i TasksFragment). Navigacija među fragmentima napravljena je preko donje navigacijske trake. Izgled MainActivitya opisan je u activity_main.xml *layout* resursu, vrlo je jednostavan, sadrži komponentu FrameLayout koja se koristi za prikaz fragmenta i BottomNavigationView komponentu odgovornu za prikaz donje navigacijske trake.

Programski kod 3.1 prikazuje XML kod activity_main.xml datoteke.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout 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"
    tools:context=".ui.activities.MainActivity">

    <FrameLayout
        android:id="@+id/fl_wrapper"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_above="@+id/bottom_navigation" />

    <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/bottom_navigation"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_alignParentBottom="true"
        app:labelVisibilityMode="unlabeled"
        app:menu="@menu/bottom_nav_menu" />

</RelativeLayout>
```

Programski kod 3.1 activity_main.xml datoteka

Donja navigacijska traka napravljena je u odvojenoj XML datoteci (bottom_nav_menu.xml), sadrži dvije *item* komponente za navigaciju. Programski kod 3.1 prikazuje XML kod bottom_nav_menu.xml datoteke

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/nav_home"
        android:icon="@drawable/ic_home"
        android:title="@string/home" />

    <item
        android:id="@+id/nav_tasks"
        android:icon="@drawable/ic_list"
        android:title="@string/tasks" />

</menu>
```

Programski kod 3.2 bottom_nav_menu.xml datoteka

Na slici 3.3 prikazan je izgled donje navigacijske trake.

Slika 3.3 Donja navigacijska traka s označenim HomeFragmentom

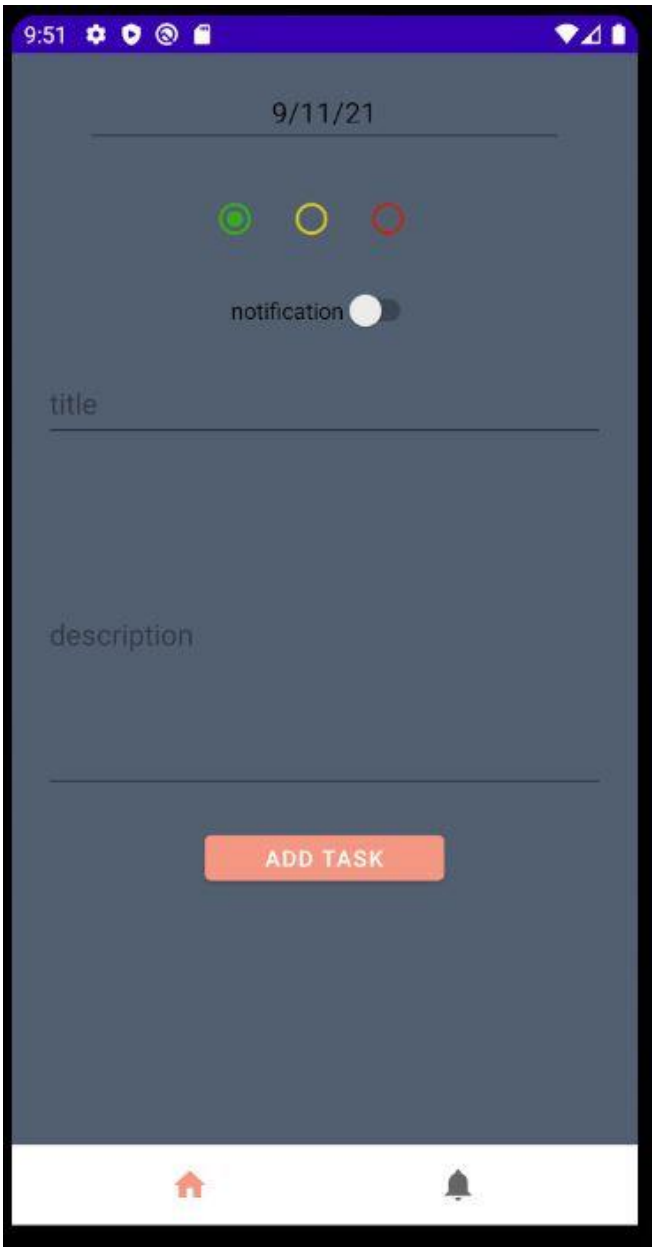
Izgled HomeFragment fragmenta definiran je u fragment_home.xml datoteci i sadrži tri EditText komponente za naslov opis i datum zadatka, Button komponentu za dodavanje zadatka u bazu podataka, RadioGroup komponentu s 3 gumba za odabir prioriteta zadatka i jednu Switch komponentu za odabir notifikacije. Kompletan izgled fragmenta omotan je u ScrollView komponentu kojom se postiže mogućnost lakog i kvalitetnog prikaza fragmenta u slučaju okretanja uređaja horizontalno.

Programski kod 3.3 prikazuje XML kod fragment_home.xml datoteke (posebne komponente su sklopljene zbog preglednosti koda).

```xml
<?xml version="1.0" encoding="utf-8"?>


<ScrollView 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:fillViewport="true">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/grey_backgorund">

        <EditText...>

        <EditText...>

        <EditText...>

        <Button...>

        <RadioGroup...>

        <Switch...>

    </androidx.constraintlayout.widget.ConstraintLayout>

</ScrollView>
```

Programski kod 3.3 fragment_home.xml datoteka

Slika 3.3 prikazuje izgled HomeFragment fragmenta u mobilnoj aplikaciji. Pri dnu se nalazi donja navigacijska traka s označenim trenutnim fragmentom (HomeFragment), vide se TextBox komponente s naputcima gdje se pišu naslov i opis zadatka te TextBox komponenta koja prikazuje datum odabran *DatePickerom*. Vidimo da je u grupi radijalnih gumbova označen zeleni gumb što znači zadatak najnižeg prioriteta (crveni gumb označava zadatak najvišeg prioriteta, žuti označava zadatak srednje razine prioriteta).



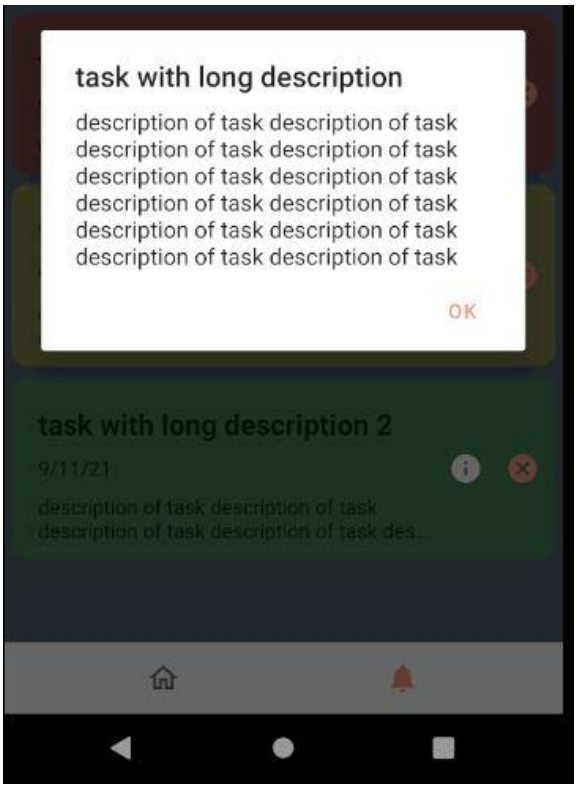
Slika 3.4 HomeFragment fragment u mobilnoj aplikaciji

Izgled FragmentTasks fragmenta opisan je u tasks_fragment.xml datoteci. Vrlo je jednostavan te sadrži samo jednu komponentu, a to je RecyclerView komponenta. Programski kod 3.4 prikazuje XML kod tasks_fragment.xml datoteke. Kako se radi o fragmentu odabran je FrameLayout i dodana je pozadina koja se nalazi u posebnoj XML datoteci. Kod odabira RecyclerView komponente potrebno je obratiti pozornost da se odabere androidx komponenta. Također, svakoj komponenti treba dodijeliti id.

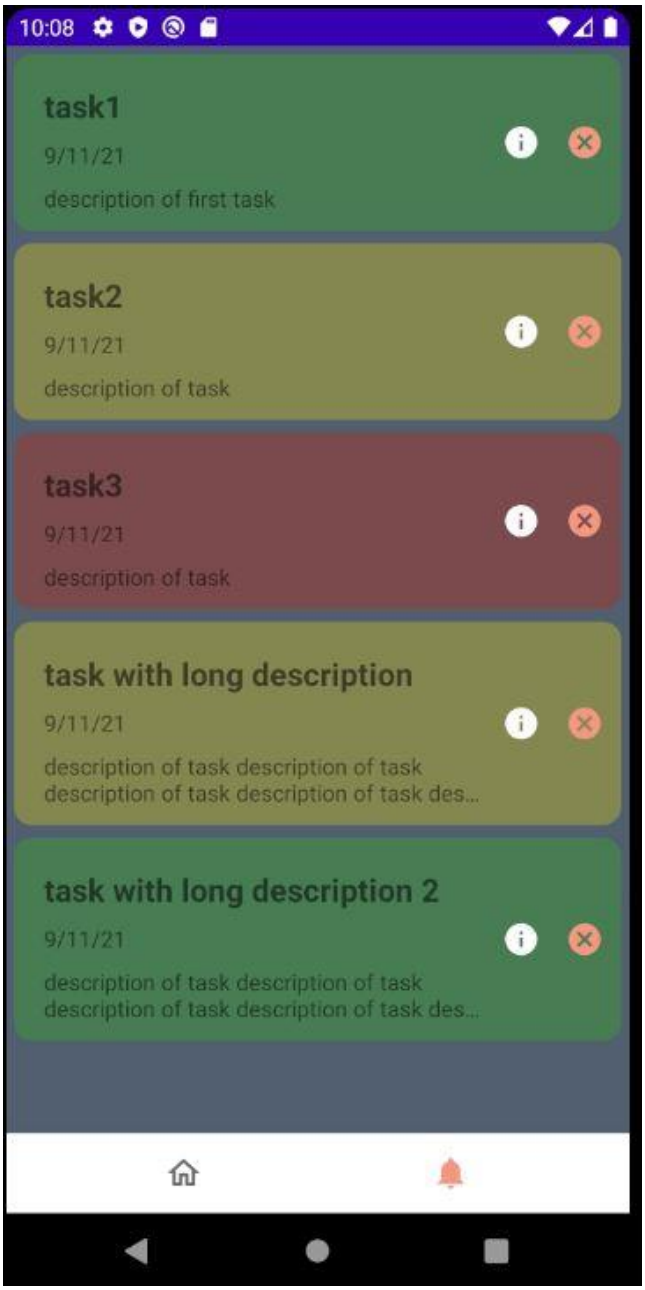
```xml
<?xml version="1.0" encoding="utf-8"?>

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/grey_backgorund">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rvTasks"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_margin="5dp"
        app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
/>

</FrameLayout>
```

Programski kod 3.4 fragment_tasks.xml datoteka

Slika 3.6 prikazuje TasksFragment fragment u mobilnoj aplikaciji. Vidljivi su zadatci različitih prioriteta prikazani *recyclerViewom*, poredani po datumu (od najbližeg prema najdaljem), gumb s opcijom za brisanje pojedinog zadatka te gumb koji prikazuje potpuni opis zadatka u slučaju da je njegov opis predug kao što je prikazano na slici 3.5.



Slika 3.5 Prikaz opisa pojedinog zadatka



Slika 3.6 Prikaz TasksFragment fragmenta u mobilnoj aplikaciji

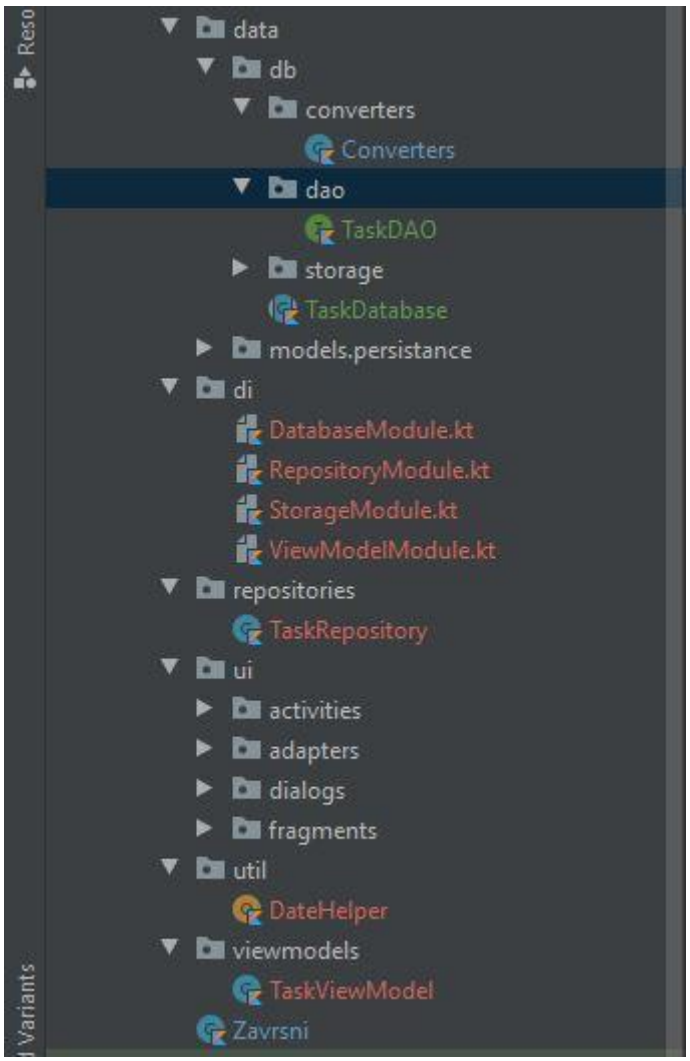
Predložak za prikaz pojedinog zadatka u *recyclerViewu* opisan je u posebnoj XML datoteci, item_task.xml. Sadrži tri TextView komponente za prikaz naslova, opisa i datuma i dvije ImageView komponente koje podržavaju funkciju klika, služe za prikaz cijelog opisa i brisanje stavke.

3.2.2. Struktura mobilne aplikacije i protok podataka

Kako bi ostatak mobilne aplikacije bio razumljiv, prvo je potrebno objasniti strukturu mobilne aplikacije i protok podataka. Počevši od strukture, kod je raspoređen u šest glavnih paketa, a to su:

- data – sadrži pakete baze podataka (db) i modela (models.persistance)
- di (*dependency injection*) – sadrži module ubrizgavanja ovisnosti
- repositories – sadrži repozitorij
- ui (*user interface*) – sadrži pakete korisničkog sučelja (activities, adapters, dialogs i fragments)
- util (*utilities*) – sadrži pomoćne alate
- viewmodels – sadrži *view modele*

Slika 3.7 prikazuje strukturu mobilne aplikacije.



Slika 3.7 Glavna struktura mobilne aplikacije

Sljedeća bitna stvar je protok podataka. Kao što je u prethodnim poglavljima napisano, komunikacija pogleda i modela odvija se preko *view modela*, što znači da su metode upravljanja podatcima definirane u *view modelu*, a to je TaskViewModel.kt datoteka. Definirane su metode umetanja novog zadatka, brisanja zadatka, brisanja starih zadataka s datumima koji su prošli, i dohvaćanje broja zadataka s opcijom obavijesti za trenutni dan. Modul TaskViewModela definiran je unutar di paketa za ubrizgavanje podataka.

Programski kod 3.5 prikazuje TaskViewModel.kt datoteku.

```
class TaskViewModel(
    private val taskRepository: TaskRepository,
) : ViewModel() {

    val tasks: LiveData<List<Task>> = taskRepository.getTasksLiveData()

    fun insertTask(task: Task) {
        viewModelScope.launch {
            taskRepository.insertTask(task)
        }
    }

    fun deleteTask(task: Task) {
        viewModelScope.launch {
            taskRepository.deleteTask(task)
        }
    }

    fun deleteOldTasks(currentTime: Long) {
        viewModelScope.launch {
            taskRepository.clearOldTasks(currentTime)
        }
    }

    suspend fun getTasksSumForCurrentDay(): Int {
        val currentTime = GregorianCalendar(Locale.getDefault()).timeInMillis
        val startOfDay = DateHelper.getStartTimeOfDate(currentTime)
        val endOfDay = DateHelper.getEndTimeOfDate(currentTime)

        return taskRepository.getTasksAsynchronous()
            .filter { task ->
                task.date in startOfDay..endOfDay
                        && task.notification
            }.size

    }
```

Programski kod 3.5 TaskViewModel.kt datoteka

Kao što je vidljivo u programskom kodu 3.5, TaskViewModel komunicira s repozitorijem (TaskRepository.kt, programski kod 3.6) koji se dalje poziva na metode u TaskStorage.kt (programski kod 3.7) datoteci, tek ona u sebi sadrži objekt za dohvaćanje podataka iz baze podataka koji, ovisno o pozvanoj metodi, ima definiran upit za rad s bazom podataka.

```
class TaskRepository(
    private val taskStorage: TaskStorage
) {

    fun getTasksLiveData() = taskStorage.getTasksLiveData()

    suspend fun insertTask(task: Task) {
        taskStorage.insertTask(task)
    }
    suspend fun getTasksAsynchronous(): MutableList<Task> {
        return taskStorage.getAllTasksAsynchronous()
    }
    suspend fun deleteTask(task: Task) {
        taskStorage.deleteTask(task)
    }
    suspend fun clearOldTasks(currentTime: Long) {
        taskStorage.deleteOldTasks(currentTime)
    }

}
```

Programski kod 3.7 TasksRepository.kt datoteka

```
class TaskStorage(
    private val taskDAO: TaskDAO
) {
    fun getTasksLiveData() = taskDAO.getAllTasksLiveData()

    suspend fun getAllTasksAsynchronous(): MutableList<Task> {
        return taskDAO.getAllTasksAsynchronous()
    }
    suspend fun insertTask(task: Task) {
        taskDAO.insertTask(task)
    }
    suspend fun deleteTask(task: Task) {
        taskDAO.deleteTask(task)
    }
    suspend fun deleteOldTasks(currentTime: Long) {
        taskDAO.clearOldTasks(currentTime)
    }

}
```

Programski kod 3.6 TaskStorage.kt datoteka

TaskDAO.kt datoteka je sučelje objekta za dohvaćanje podataka, kako je baza podataka rađena pomoću *Room* biblioteke to je ključan objekt za rad s bazom podataka. U TaskDAO.kt datoteci definirani su upiti za umetanje zadatka, dohvaćanje liste zadataka iz baze podataka, dohvaćanje *LiveData* liste zadataka, uklanjanje starih zadataka i uklanjanje pojedinog zadatka. Programski kod 3.8 prikazuje Task.DAO.kt datoteku, funkcije koje su definirane kao *suspend fun* se pozivaju asinhrono u opsegu *coroutine*.

```
@Dao
interface TaskDAO {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertTask(task: Task)

    @Query("SELECT * FROM tasks ORDER BY date ASC")
    suspend fun getAllTasksAsynchronous(): MutableList<Task>

    @Query("SELECT * FROM tasks ORDER BY date ASC")
    fun getAllTasksLiveData(): LiveData<List<Task>>

    @Query("DELETE FROM tasks WHERE date <:currentTime")
    suspend fun clearOldTasks(currentTime: Long)

    @Delete
    suspend fun deleteTask(task: Task)
}
```

Programski kod 3.8 TaskDAO.kt datoteka

Zbog korištenja *Room* biblioteke baza podataka je definirana vrlo jednostavno, u TaskDatabase.kt datoteci definirana je anotacijom @Database te sadrži popis entiteta, verziju i apstraktnu metodu za dohvaćanje TaskDAO objekta. *Singletoni* baze podataka, repozitorija i spremnika definirani su u di paketu unutar modula za ubrizgavanje ovisnosti. Programski kod 3.9 prikazuje modul baze podataka s definiranim *singletonom*.

```
val databaseModule = module {
    single {
        Room.databaseBuilder(androidApplication(), TaskDatabase::class.java,
DATABASE_NAME)
            .allowMainThreadQueries()
            .addMigrations()
            .build()
    }
    single { get<TaskDatabase>().getTaskDAO() }
}
```

Programski kod 3.9 ViewModelModule.kt datoteka

Za definiran entitet zadatka iz baze podataka napravljen je i model s potrebnim atributima (Task.kt), prikazan u programskom kodu 3.10.

```
@Entity(tableName = TASK_TABLE_NAME)

data class Task(

        @PrimaryKey(autoGenerate = true) val id: Long = 0,
        val title: String,
        val description: String,
        val date: Long,
        val priority: TaskPriority,
        val notification: Boolean
)
```

Programski kod 3.10 Task.kt datoteka

3.2.3. Aktivnosti i fragmenti

U mobilnoj aplikaciji se nalazi jedna aktivnost, MainActivity, koja je odgovorna za kontroliranje navigacije između fragmenata. Pri stvaranju aktivnosti (onCreate() metoda) instanciraju se fragmenti te ako je prvi puta kreirana aktivnost (ako je tek otvorena mobilna aplikacija), postavlja se HomeFragment kao početni fragment, prikazuje se obavijest za zadatke s označenom opcijom obavijesti i brišu se stari zadatci (zadatci proteklih datuma). Brisanje starih zadataka i dohvaćanje broja zadataka s obavijesti obavlja se asinhrono, preko taskViewModela koji je ubrizgan inject() metodom. Također je regulirana i navigacija preko donje navigacijske trake. Programski kod 3.11 prikazuje MainActivity.kt datoteku.

```
class MainActivity : AppCompatActivity() {
    val TAG = "MainActivity"

    private val taskViewModel: TaskViewModel by inject()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val homeFragment = HomeFragment()
        val tasksFragment = TasksFragment()

        if (savedInstanceState == null) {
            setCurrentFragment(homeFragment)
            showDailyTasksNotification()
            clearOldTasks()
        }
        bottom_navigation.setOnItemSelectedListener {
            when (it.itemId) {
                R.id.nav_home -> {
                    setCurrentFragment(homeFragment)
                    true
                }
                R.id.nav_tasks -> {
                    setCurrentFragment(tasksFragment)
                    true
                }
                else -> false
            }
        }
        bottom_navigation.setOnItemReselectedListener { }
    }
    private fun showDailyTasksNotification() {
        CoroutineScope(Main).launch {
            val sum = taskViewModel.getTasksSumForCurrentDay()
            Toast.makeText(
                this@MainActivity,
                getString(
                    R.string.important_tasks_message,
                    sum.toString()
                ), Toast.LENGTH_LONG
            ).show()
        }
    }
    private fun clearOldTasks() {
        val calendar = GregorianCalendar()
taskViewModel.deleteOldTasks(DateHelper.getStartTimeOfDate(calendar.timeInMillis))
    }
    private fun setCurrentFragment(fragment: Fragment) =
        supportFragmentManager.beginTransaction().apply {
            replace(R.id.fl_wrapper, fragment)
            commit()
        }
}
```

Programski kod 3.11 MainActivity.kt datoteka

HomeFragment fragment početni je zaslon mobilne aplikacije i služi za dodavanje novih zadataka. Kao i kod MainActivitya, ubrizgan je *view model* kako bi spremanje novog zadatka u bazu podataka bilo moguće. Prije dodavanja novog zadatka potrebno je provjeriti jesu li uneseni svi podatci potrebni za stvaranje podatka (unos datuma, naslova i opisa) te provjera prioriteta, to je osigurano metodama prikazanim u programskom kodu 3.12.

```
private fun validateTaskInput(): Boolean {
    var isValid = true
    etTitle.apply {
        if (text.isEmpty()) {
            error = getString(R.string.title_required)
            requestFocus()
            isValid = false
        }
    }
    etDate.apply {
        if (text.isEmpty()) {
            error = getString(R.string.date_required)
            requestFocus()
            isValid = false
        }
    }
    etDescription.apply {
        if (text.isEmpty()) {
            error = getString(R.string.description_required)
            requestFocus()
            isValid = false
        }
    }
    return isValid
}
private fun checkTaskPriority(): TaskPriority {
    return when {
        rbRed.isChecked -> {
            TaskPriority.HIGH
        }
        rbYellow.isChecked -> {
            TaskPriority.MEDIUM
        }
        else -> {
            TaskPriority.LOW
        }
    }
}
```

Programski kod 3.12 HomeFragment.kt metode za provjeru ispravnosti unosa naslova, opisa, datuma i prioriteta zadatka

Datum se postavlja *date pickerom*, klikom na TextBox komponentu za datum, otvara se dijalog s DatePickerFragment fragmentom reguliran showDatePicker() metodom (programski kod 3.13) te se nakon odabira, taj datum prikazuje u TextBox komponenti onDateSet() metodom(programski kod 3.13). U koliko se ne odabere datum, bit će postavljen na trenutni.

```
private fun showDatePicker(title: String, dialogId: Int) {
    val datePickerFragment = DatePickerFragment()
    val date = Date()

    currentDate.let {
        date.time = it
    }

    val arguments = bundleOf(
        DATE_PICKER_ID to dialogId,
        DATE_PICKER_TITLE to title,
        DATE_PICKER_DATE to date
    )

    datePickerFragment.arguments = arguments
    datePickerFragment.show(childFragmentManager, "datePicker")
}
override fun onDateSet(view: DatePicker?, year: Int, month: Int, dayOfMonth: Int)
{
    when (view?.tag as Int) {
        DIALOG_TASK_DATE -> {
            val calendar = GregorianCalendar()
            calendar.set(year, month, dayOfMonth, 0, 0, 0)
            currentDate = calendar.timeInMillis
            currentDate.let { etDate.setText(getLocalityFormattedDate(it,
requireContext())) }
        }
        else -> throw IllegalArgumentException("Invalid mode when receiving
DatePickerDialog result")
    }
}
```

Programski kod 3.14 HomeFragment.kt metode za odabir i prikaz datuma

Nakon što su unesene potrebne informacije, klikom na gumb za dodavanje pokreće se kod unutar *btnAddTask.setOnClickListenera* koji stvara novu instancu objekta Task() te ju preko taskViewModela sprema u bazu podataka kao što je prikazano u programskom kodu 3.14.

```
btnAddTask.setOnClickListener {
    if (validateTaskInput()) {
        val priority = checkTaskPriority()
        val notification = sbNotification.isChecked
        val task = Task(
            title = etTitle.text.toString().trim(),
            description = etDescription.text.toString().trim(),
            date = currentDate,
            priority = priority,
            notification = notification
        )
        taskViewModel.insertTask(task)
    }
}
```

Programski kod 3.13 HomeFragment.kt dodavanje novog zadatka

TasksFragment je fragment koji je odgovoran za prikaz i uklanjanje zadataka iz baze podataka. Kako je potrebno dohvatiti sve zadatke pohranjene u bazu podataka i neke moći i ukloniti, ubrizgan je TaskViewModel preko *inject()* metode. U *onCreate()* metodi (programski kod 3.15) fragmenta TasksFragment pretplaćen je kao promatrač na *live data* listu zadataka taskViewModela kako bi pri svakoj promjeni mogao osvježiti listu zadataka.

```
private val tasksAdapter = TaskAdapter(emptyList(), this)
private val taskViewModel: TaskViewModel by inject()


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

    taskViewModel.tasks.observe(this, { tasks ->
        tasksAdapter.loadNewTasks(tasks)
    })
}
```

Programski kod 3.15 TasksFragment.kt onCreate() metoda

Klikom na pojedini zadatak prikazuje se dijalog s njegovim punim opisom pomoću *showDescriptionDialog()* metode prikazane u programskom kodu 3.16.

```
private fun showDescriptionDialog(
    context: Context,
    title: String,
    message: String,
    buttonText: Int = R.string.ok
) {

    AlertDialog.Builder(context)
        .setTitle(title)
        .setMessage(message)
        .setPositiveButton(buttonText) { _, _ -> }
        .setCancelable(true)
        .show()

}
```

Programski kod 3.16 TasksFragment.kt showDescriptionDialog() metoda

Zadatci se prikazuju u *RecyclerViewu* pomoću adaptera postavljenog u *onViewCreated()* metodi prikazanoj u programskom kodu 3.17.

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

    rvTasks.layoutManager = LinearLayoutManager(context,
LinearLayoutManager.VERTICAL, false)

    rvTasks.adapter = tasksAdapter
    super.onViewCreated(view, savedInstanceState)
}
```

Programski kod 3.17 TasksFragment.kt onViewCreated() metoda

Kako je TasksFragment pretplaćen na sučelje adaptera s dvije metode, za uklanjanje zadatka i prikaz dijaloga, prepisane su i uvedena je funkcionalnost kao što je prikazano u programskom kodu 3.18.

```
override fun onDeleteClick(task: Task) {
    taskViewModel.deleteTask(task)
}

override fun onInfoClick(task: Task) {
    showDescriptionDialog(requireContext(), task.title, task.description)
}
```

Programski kod 3.18 TasksFragment.kt metode za uklanjanje zadatka i prikaz punog opisa u dijalogu

Prikaz liste zadataka u *RecyclerViewu* reguliran je adapterom (TasksAdapter.kt). Adapter kao atribute ima listu zadataka i *listenera* koji je tipa sučelja *OnTaskClickListener* na kojeg je TasksFragment pretplaćen. Dodana je nova metoda za osvježavanje liste zadataka (programski kod 3.19).

```
fun loadNewTasks(newTasks: List<Task>) {

    tasks = newTasks
    notifyDataSetChanged()
}
```

Programski kod 3.19 TasksAdapter.kt metoda za osvježavanje liste zadataka

U klasi adaptera nalazi se klasa TaskViewHolder s *onBind()* metodom (programski kod 3.20) koja se poziva za svaki zadatak liste prilikom stvaranja pogleda. U toj metodi je određeno, ovisno o prioritetu pojedinog zadatka, koji pozadinski resurs će se uzimati za prikaz zadatka (crvena žuta ili zelena pozadina), predaju se vrijednosti naslova opisa i datuma te su postavljeni *.setOnClickListeneri* za uklanjanje i prikaz opisa.

```
fun onBind(task: Task, listener: OnTaskClickListener) {

    when (task.priority) {
        TaskPriority.LOW -> {
            itemView.background =
itemView.context.getMyDrawable(R.drawable.bg_green)
        }
        TaskPriority.MEDIUM -> {

            itemView.background =
itemView.context.getMyDrawable(R.drawable.bg_yellow)
        }

        TaskPriority.HIGH -> {
            itemView.background =
itemView.context.getMyDrawable(R.drawable.bg_red)
        }
    }

    itemView.tvTitle.text = task.title
    itemView.tvDescription.text = task.description
    itemView.tvDate.text = DateHelper.getLocalityFormattedDate(task.date,
itemView.context)

    itemView.ivDelete.setOnClickListener {
        listener.onDeleteClick(task)
    }

    itemView.ivInfo.setOnClickListener {
        listener.onInfoClick(task)
    }

}
```

Programski kod 3.20 TasksAdapter.kt onBind() metoda TaskViewHolder klase

4. ZAKLJUČAK

U završnom radu razvijena je mobilna aplikacija za unaprjeđivanje organizacije i produktivnosti, riješeni su problemi organizacije vremena, prioritiziranja zadataka i zaboravljanja na zadatke. Napravljena je mogućnost dodavanja i brisanja zadataka, a prikaz zadataka sortiran je po datumu, od najbližeg prema najdaljem. Navedeni su potrebni alati i vještine za izradu Android mobilne aplikacije s opisom izrade i programskim kodom pisanim u Kotlinu. Prikazan je proces izrade mobilne aplikacije počevši od *mockupa* i dijagrama toka zatim izrade korisničkog sučelja u XML-u, postavljanje strukture mobilne aplikacije, objašnjen je protok podataka i komunikacija između pogleda, *view modela* , modela i same baze podataka te prikazano i objašnjeno uvođenje logike, funkcionalnosti i povezivanje komponenti Android mobilne aplikacije.

LITERATURA

[1] Android developer, „Android studio“, [online]. Dostupno na: https://developer.android.com/. [26.lipnja 2021]

[2] Infoworld, „what is kotlin“, [online]. Dostupno na: https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html. [26.lipnja 2021]

[3] Kotlinlang, „android overview“, [online]. Dostupno na: https://kotlinlang.org/docs/android-overview.html. [26.lipnja 2021]

[4] Android developer, „data storage“, [online]. Dostupno na: https://developer.android.com/training/data-storage/room. [26.lipnja 2021]

[5] Ray Wanderlich, Aldo O. Dominguez, Jennifer Bailey, Dean Djermanović, SavingData on Android (First Edition): Learning Room, Firebase and SQLite with Kotlin, RazerwareLLC, 2019.

[6] Kotlinlang, „coroutines overview“, [online]. Dostupno na: https://kotlinlang.org/docs/coroutines-overview.html. [26.lipnja 2021]

[7] Android developer, „coroutines“, [online]. Dostupno na: https://developer.android.com/kotlin/coroutines. [26.lipnja 2021]

[8] Android developer, „recyclerview“, [online]. Dostupno na: https://developer.android.com/guide/topics/ui/layout/recyclerview. [24.lipnja 2021]

[9] Android developer, „recyclerview“, [online]. Dostupno na: https://developer.android.com/jetpack/androidx/releases/recyclerview. [24.lipnja 2021]

[10] Mindorks, „Kotlin koin“, [online]. Dostupno na: https://blog.mindorks.com/kotlin-koin-tutorial. [3.rujna 2021]

[11] Robert C. Martin, Clean Architecture: A Craftsman's guide to Software Structure and Design, Pearson, United States, 2017.

[12] Geeksforgeeks, „mvvm pattern in android“, [online]. Dostupno na: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/. [3.rujna 2021]

[13] Android developer, „Guide to app arhitecture“, [online]. Dostupno na: https://developer.android.com/jetpack/guide. [4.rujna 2021]

SAŽETAK

Završni rad objašnjava potrebna znanja, tehnologije, alate i vještine za izradu Android mobilne aplikacije. Objašnjen je rad s lokalnim bazama podataka na Android uređajima pomoću Room biblioteke, asinhroni procesi pomoću Kotlin Coroutines biblioteke te njihove prednosti i zašto bi se trebali koristiti. Važnost i primjena ubrizgavanja ovisnosti pomoću Koin dependency injection biblioteke i postavljanje pravilne strukture mobilne aplikacije po MVVM obrascu. Navedeni alati i vještine upotrijebljeni su u izradi Android mobilne aplikacije završnog rada koja rješava problem organizacije i produktivnosti.

Ključne riječi: Android, Coroutines, Koin, Kotlin, Mobilna aplikacija, MVVM, Organizacija, Produktivnost, Room.

ABSTRACT

Mobile application for improvement of organisation and productivity

The thesis covers and explains necessary knowledge, technology, tools and skills for development of an Android mobile application. Thesis explains local database handling with Room library on Android devices, asynchronous process handling with Kotlin Coroutines library with their advantages and explanation why should they be used. Dependency injection with Koin dependency injection library and what is the right structure of mobile application based on MVVM structural pattern is also covered. Stated tools and skills are used to develop the Android mobile application of the thesis, which resolves problem of organisation and productivity.

Keywords: Android, Coroutines, Koin, Kotlin, Mobile application, MVVM, Organization, Productivity, Room.

ŽIVOTOPIS

Bruno Junaković rođen je 19.05.1997 u Osijeku, nakon završetka II. Gimnazije upisuje preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek na Sveučilištu Josipa Jurja Strossmayera u Osijeku.

_____

Potpis autora