UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Jelena Ivković

PRIMENA PROGRAMSKOG JEZIKA KOTLIN U RAZVOJU VIŠEPLATFORMSKIH MOBILNIH APLIKACIJA

master rad

Beograd, 2022.

Mentor:

dr Saša MALKOV, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Vladimir FILIPOVIĆ, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Stefan MIŠKOVIĆ, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: 29.9.2022.

Porodici.

Naslov master rada: Primena programskog jezika Kotlin u razvoju višeplatformskih mobilnih aplikacija

Rezime: *Kotlin* je višeplatformski programski jezik koji se prvi put pojavio 2011. godine kao novi jezik za *JVM*. Od 2017. Kotlin je od strane kompanije *Google* podržan kao zvanični programski jezik za razvoj *Android* aplikacija. Pored razvoja mobilnih aplikacija, kako se Kotlin prevodi i izvršava na svim uređajima koji podržavaju *JVM*, može se koristiti i za razvoj veb i desktop aplikacija. Cilj ovog rada je predstavljanje tehnika razvoja višeplatformskih mobilnih aplikacija pomoću programskog jezika *Kotlin*, a na primeru razvoja aplikacije za ostvarivanje programa lojalnosti u apotekama. Aplikacija je napravljena kao *KMM* (*Kotlin Multiplatform Mobile*) – aplikacija, koristeći arhitekturalni softverski šablon *MVVM* (*Model-View-ViewModel*).

Ključne reči: programski jezik Kotlin, višeplatformske mobilne aplikacije, KMM, MVVM

Sadržaj

Glava 1

1. Uvod

Sa konstantnim napretkom tehnologija, sve veća je potreba za jasnim, brzim i efikasnim rešenjima u oblasti razvoja softvera. Tako se i u razvoju novih programskih jezika dosta značaja pridaje tome da oni omogućavaju lakše i brže pisanje kôda. Upravo sa tim ciljem, a da bi nadomestio neke od nedostataka programskog jezika *Java*, nastao je *Kotlin*.

Kotlin je programski jezik koji se prvi put pojavio 2011. godine kao novi jezik za *Java* virtualnu mašinu (*JVM*), a verzija iz februara 2016. smatra se prvom stabilnom [1]. Kompanija *Google* je 2017. godine podržala *Kotlin* kao zvanični jezik za razvoj *Android* aplikacija [2]. Pored toga što se može koristiti za razvoj mobilnih aplikacija, kako se u osnovi radi o jeziku opšte namene koji se prevodi i izvršava na svim uređajima koji podržavaju *JVM*, *Kotlin* se može korisititi i za razvoj veb i desktop aplikacija [3]. Zastupljenost jezika *Kotlin* je sve veća, posebno u razvoju mobilnih aplikacija za *Android* i višeplatformskih aplikacija [4]. Neke od kompanija koje za svoje mobilne aplikacije koriste *Kotlin* su *Pinterest*, *Coursera*, *Netflix*, *Uber*, *Duolingo*, *Airbnb*, *Slack* i mnoge druge [5].

Cilj ovog rada je predstavljanje tehnika razvoja višeplatformskih mobilnih aplikacija pomoću programskog jezika *Kotlin*, a na primeru razvoja aplikacije za ostvarivanje programa lojalnosti u apotekama.

U poglavlju 2 ovog rada se govori o mobilnim aplikacijama i višeplaformskom programiranju. Nakon toga, u poglavlju 3, govori se o programskom jeziku *Kotlin*. Opisani su istorija i filozofija njegovog nastanka i razvoja, osnovni koncepti koje *Kotlin*

nudi, poređenje sa programskim jezikom *Java*, kao i značaj *Kotlina* u višeplatformskom programiranju. U poglavlju 3 je opisan i arhitekturalni softverski šablon *MVVM* (*Model–View-ViewModel*). Opis aplikacije razvijene u okviru ovog rada nalazi se u poglavlju 4. Na kraju, u poglavlju 5 je prikazan zaključak i mogućnosti daljeg razvoja aplikacije.

Glava 2

2. Mobilne i višeplatformske aplikacije

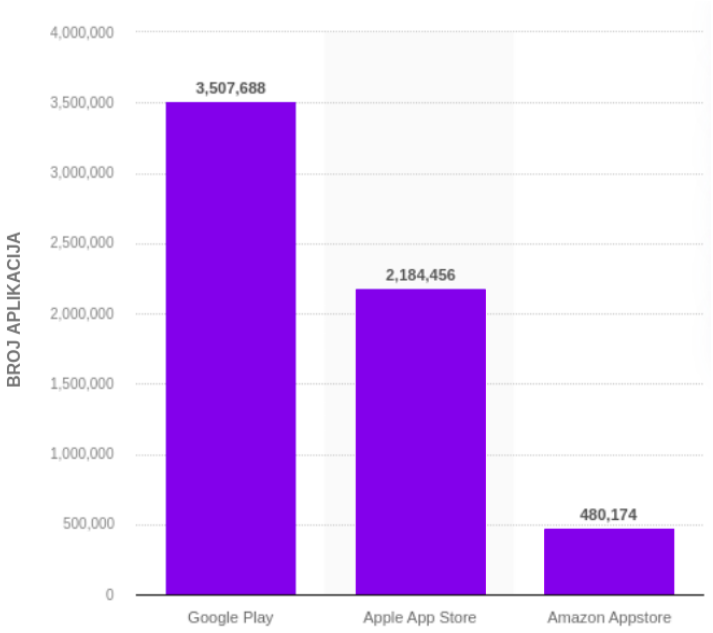
2.1. Značaj mobilnih aplikacija

Pametni telefoni i mobilne aplikacije promenili su način funkcionisanja mnogih oblasti života savremenog čoveka. Ljudi žele da brzo i jednostavno, korišćenjem mobilnog telefona, završavaju svoje obaveze (kupovina, plaćanje računa, komunikacija, učenje i drugo). S obzirom na navike koje su nove tehnologije donele, kompanije teže da se tome prilagode i da pruže svoje usluge na način koji korisnicima najviše odgovara. Štaviše, kada je aplikacija dobro dizajnirana, umnogome može doprineti popularnosti kompanije koja ju je napravila.

Google Play je platforma nastala 2012. godine objedinjavanjem tri različite aplikacije: *Android Market*, *Google Music* i *Google eBookstore*. Aplikacija *Google Market* je postojala od 2008. godine, Google *eBookstore* od 2010, a *Google Music* od 2011. godine. *Google Play* je na samom početku imala 450 hiljada aplikacija [6].

Stiv Džobs je 2007. godine predstavio prvi *iPhone* sa nekoliko aplikacija koje su se i danas zadržale na ovim uređajima. Međutim, *Apple App Store* je nastala tek naredne godine, sa oko 500 aplikacija [7].

Tržište mobilnih aplikacija je danas veoma razvijeno, a broj aplikacija stalno raste. *Google Play Store* danas ima nešto preko 3,5 miliona aplikacija, a *Apple App Store* preko 2,18 miliona. Naredna najveća platforma za preuzimanje aplikacija je *Amazon Appstore* sa malo više od 480 hiljada aplikacija. Na slici 2.1 se može videti grafički prikaz broja aplikacija po prodavnici u drugom kvartalu 2022. godine [8].



Slika 2.1: Broj aplikacija dostupnih na vodećim prodavnicama aplikacija u drugom kvartalu 2022. godine

2.2. Višeplatformsko programiranje

Kako su oglasi za posao programera verovatno najmnogobrojniji, jasno je da postoji manjak ovog kadra. Kako bi se to nadomestilo, mnogi počinju sa korišćenjem alata koji omogućavaju pravljenje aplikacija za više platformi odjednom, na primer za *Android* i *iOS*. Osim toga, često i nema razlike u kôdu koji, na primer, služi za pozivanje nekog *REST* servisa, a mora se pisati po jednom za svaku platformu. Ono što višeplatformsko programiranje nudi jeste upravo mogućnost pisanja jednog kôda koji se izvršava na više različitih platformi. Na taj način, veliki deo kôda je zajednički, a samo neke stvari koje su specifične za platformu se moraju pisati odvojeno. Kada se prave izmene, to se uglavnom radi samo jednom, upravo zbog toga što je veliki deo kôda zajednički. Nekada je bolje imati nativne aplikacije za svaku platformu ponaosob, ali za to je potreban i veći broj programera.

Neke od prednosti ovakvog načina razvoja aplikacija u odnosu na nativne mobilne aplikacije su:

1. **Veći doseg** – Što se veći broj platformi pokrije, više ljudi će koristiti aplikaciju. *Google* sa *Androidom* i *Apple* sa *iOS*-om su dominantni na tržištu mobilnih telefona. Međutim, nije zanemarljiv ni broj onih koji koriste *Blackberry*, *Windows Phone* ili neke druge manje poznate platforme, za koje je takođe moguće praviti aplikacije.
2. **Lakši marketing** – Kada se aplikacija odjednom pravi za više platformi, to znači i da će se marketing raditi istovremeno, što štedi vreme i novac.
3. **Jednostavnije održavanje** – Kako je veliki deo kôda zajednički za sve platforme, izmene se vrše jednom i odmah su primenjene na sve platforme.
4. **Nativni izgled** – Deo kôda koji nije zajednički za sve platforme, uglavnom bude kôd vezan za korisnički interfejs. Razlog tome je upravo to da aplikacije mogu izgledati kao da su nativne, za svaku platformu posebno.
5. **Korišćenje poznate tehnologije** – Za sve platforme se, u velikoj meri, može korisiti isti programski jezik. Dakle, nema potrebe učiti novi programski jezik za svaku platformu za koju se razvija aplikacija.
6. **Manji troškovi razvoja** – Jeftinije je imati jedan tim programera koji razvija višeplatformsku aplikaciju, nego imati jedan tim po platformi. Pored toga, timovi ne moraju praviti kompromise oko rešenja jer se odluke donose u okviru jednog tima.

Što se tiče mana razvoja višeplatformskih aplikacija, uglavnom su tehničke prirode. Nekada se aplikacija ne može napraviti tako da radi potpuno isto i na *Androidu* i na *iOS*-u, na primer zbog različitih načina rada ekrana, korišćenja sistema datoteka, pristupa kameri ili geolokaciji uređaja.

Glava 3

3. Kotlin

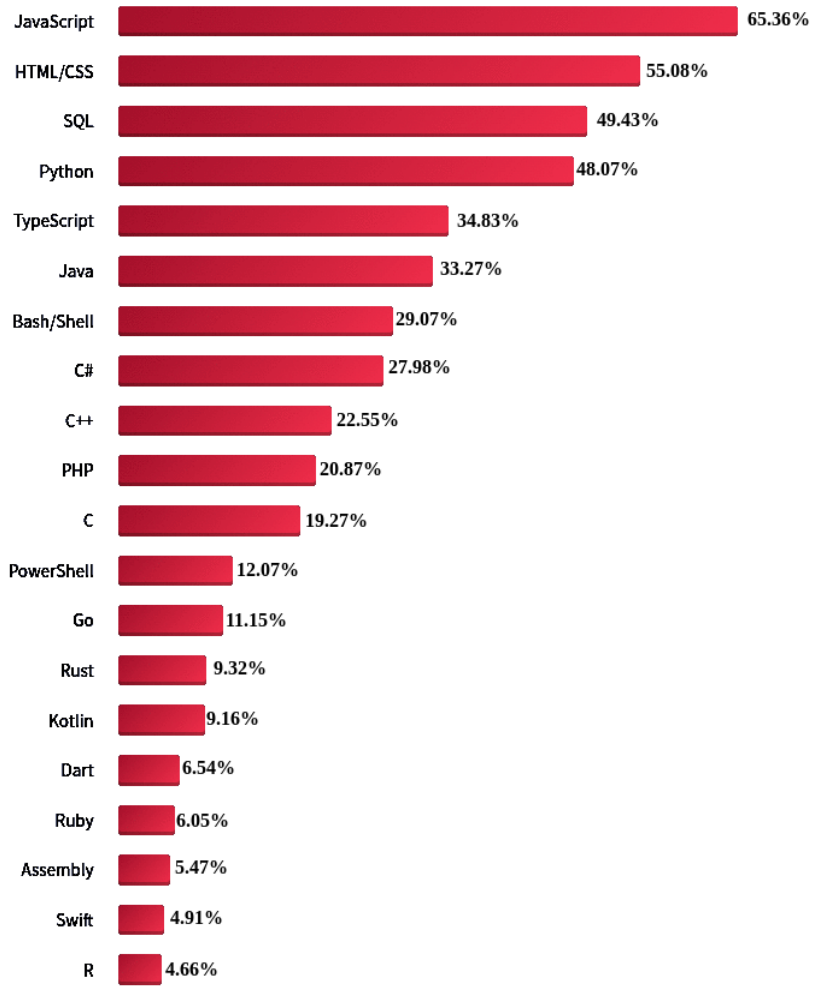
3.1. Nastanak Kotlina

Na samitu jezika za *Java* virtualnu mašinu (*JVM*) 2011. godine u Santa Klari, SAD, kompanija *JetBrains* je predstavila svoj novi projekat na kome je rađeno već skoro godinu dana. Bio je to novi programski jezik, *Kotlin*. Prvobitno je bio nazvan *Jet*, ali je ubrzo odabrano novo ime koje se zadržalo do danas. Po uzoru na neke druge programske jezike (*Java* i *Ceylon*), i Kotlin je ime dobio po ostrvu Kotlin koje se nalazi nadomak Sankt Peterburga. Cilj je bio obezbediti programski jezik koji je dovoljno produktivan na savremenim integrisanim razvojnim okruženjima (*IDE*), a istovremeno i dovoljno jednostavan za učenje. U tom trenutku, *Kotlin* nije bio u toj meri razvijen da bi se mogao koristiti u produkcione svrhe. Kako je *JetBrains* kompanija koja se primarno bavi razvojem razvojnih okruženja, radilo se i na razvoju podrške novom jeziku od strane razvojnih okruženja paralelno sa razvojem samog jezika [9, 10].

Već od 2010. godine, inženjeri kompanije *JetBrains*, tvorci čuvenog razvojnog okruženja *IntelliJ* na čijim osnovama je kasnije nastao *AndroidStudio*, bavili su se razvojem *Kotlina*. Uviđajući načine na koje je programski jezik *Java* ograničavao njihove projekte, pokušali su da pronađu dugoročnu alternativu. Uslov je bio da taj drugi programski jezik bude 100% kompatibilan sa postojećim kôdom napisanim u *Javi*. Bilo bi gotovo nemoguće ispočetka pisati tako velike projekte kao što su projekti kompanije *JetBrains*. Nakon pokušaja prelaska na *Skalu*, kako je vreme kompilacije bilo predugo, doneta je odluka da se napravi novi programski jezik [11].

Od 2017. godine kompanija *Google* je i zvanično proglasila *Kotlin* jezikom za razvoj *Android* aplikacija, a od 2019. [11, 12] *Google* programerima mobilnih aplikacija preporučuje korišćenje *Kotlina*. Danas je *Kotlin* moderan programski jezik koji je evoluirao je od alternative za *Javu* do čitavog ekosistema koji omogućava pisanje kôda različitih namena:

1. *Back-end* aplikacija,
2. *Front-end* aplikacija,
3. Višeplatformskih mobilnih aplikacija,
4. Višeplatformskih biblioteka,
5. *Android* aplikacija.



Slika 3.1: Najpopularniji programski jezici 2022. godine – *StackOveflow*

U prilog popularnosti *Kotlina* govore i istraživanja, poput onih koje izvodi i objavljuje *StackOverflow*. Naime, na listi najpopularnijih jezika među programerima 2022. godine, *Kotlin* zauzima 15. mesto sa 9.16% glasova, a jedan je od mlađih jezika. *Java* se na istoj listi nalazi na 6. mestu sa 33.27% (slika 3.1) [13]. Pored toga, sredinom 2022. godine, broj pitanja vezanih za *Kotlin* na istoj platformi zauzima oko 1%, a za *Javu*, kao jedan od najpopularnijih jezika, oko 7% od ukupnog broja pitanja [14].

3.2. Filozofija Kotlina

Kao i svaki drugi projekat, i razvoj *Kotlina* je imao svoju filozofiju, odnosno ciljeve koje je trebalo ostvariti, a to su: pragmatičnost, konciznost, sigurnost, i potpuna interoperabinost sa *Javom* [11].

3.2.1. Pragmatičnost

Kotlin je dizajniran tako da bude pragmatičan alat za programere, a to se ogleda u tri principa [15]:

1. **Održavanje jezika modernim tokom godina** – ono što je nekada bila moderna tehnologija, danas može biti zastarelo. Zato se jezik mora održavati modernim, da bi mogao da ispuni očekivanja savremenih korisnika. Pored dodavanja novih svojstava, ovo podrazumeva i postepeno isključivanje onih koja se više ne preporučuju za korišćenje u produkcione svrhe.
2. **Konstantna razmena iskustava sa korisnicima** – isprobavanje stvari u realnom životu, na realnim projektima je najbolje merilo dizajna programskog jezika. Upravo zbog toga, *JetBrains* često objavljuje eksperimentalne verzije koje korisnici mogu da koriste i da jave svoje utiske.
3. **Jednostavan prelazak na novije verzije** – nekompatibilne promene, kao što je uklanjanje svojstava iz jezika, može da dovede do teže migracije sa jedne verzije

na sledeću. Ukoliko se takve promene spremaju, one moraju da uvek budu unapred najavljene, kako bi svi bili spremni kada se dese.

3.2.2. Konciznost

Opšte je poznato da programeri više vremena provode čitajući nego pišući novi kôd [15]. Na primer, kada se radi na velikim projektima na kojima je potrebno napraviti neku promenu, potrebno je pročitati i razumeti dosta kôda da bi se došlo do onoga što treba promeniti. Što je kôd jednostavniji i koncizniji, razumevanje je brže i lakše. Pored dobrog dizajna i imenovanja klasa, funkcija i promenljivih, značajnu ulogu u ovome igra i izbor odgovarajućeg programskog jezika. Jezik je koncizan ako njegova sintaksa jasno izražava namenu napisanog kôda.

Šablonski kôd koji je u *Javi* uobičajen, kao što su geteri, seteri i logika dodeljivanja parametara konstruktora poljima, u *Kotlinu* je implicitan. Kôd nekada bude dugačak i zato što je potrebno implementirati neke standardne zadatke, kao što je pronalaženje elementa u listi, međutim, moderni programski jezici imaju standardne biblioteke koje rešavaju ovaj problem. Ono što povećava konciznost Kotlina u ovom slučaju je podrška lambda funkcijama.

Istovremeno, Kotlin ne pokušava da svede kôd na najmanji mogući broj karaktera. Na primer, u Kotlinu nije moguće definisati sopstvene operatore, samim tim, programeri biblioteka ne mogu imena metoda zameniti operatorima [15].

Za prikaz konciznosti *Kotlina* u odnosu na *Javu* može se iskoristiti kôd jednostavne klase. Na slici 3.2 prikazan je kôd napisan u *Kotlinu*, a na slici 3.3 isti kôd napisan u *Javi*. Na ovom primeru se vidi značaj implicitnog konstruktora, getera i setera.

```
class User(firstName: String, lastName: String, phoneNumber: String, email:
String)
```

Slika 3.2: Klasa *User* napisana u programskom jeziku *Kotlin*

```
public class User {
   private String firstName;
   private String lastName;
   private String phoneNumber;
   private String email;

   public User(String firstName, String lastName, String phoneNumber,
String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.phoneNumber = phoneNumber;
        this.email = email;
   }

   public String getFirstName() { return firstName; }
   public String getLastName() { return lastName; }
   public String getPhoneNumber() { return phoneNumber; }
   public String getEmail() { return email; }

   public void setFirstName(String firstName) { this.firstName = firstName;
}
   public void setLastName(String lastName) { this.lastName = lastName; }
   public void setPhoneNumber(String phoneNumber) { this.phoneNumber =
phoneNumber; }
   public void setEmail(String email) { this.email = email; }
}
```

Slika 3.3: Klasa *User* napisana u programskom jeziku *Java*

3.2.3. Sigurnost

Kada se kaže da je programski jezik siguran, misli se na dizajn koji sprečava nastanak određenih vrsta grešaka u programima. Naravno, ne postoji jezik koji može da spreči sve moguće greške. Pored toga, sprečavanje grešaka obično ima svoju cenu - kompilatoru se mora dati više podataka o nameni programa kako bi mogao da proveri da li se to poklapa sa onim što program radi.

Samim tim što se izvršava na *JVM*, *Kotlin* već ima neki vid sigurnosti, na primer sprečeni su problemi izazvani nepravilnom upotrebom dinamički alocirane memorije. Takođe, kao statički tipiziran jezik, *Kotlin* osigurava ispravnost i saglasnost tipova.

Kotlin ide i korak dalje, što znači da proverama u toku kompilacije može biti sprečeno više grešaka nego u *Javi*. Pre svega, nastoji se uklanjanju izuzetka *NullPointerException* tako što se prate promenljive koje mogu i ne mogu imati vrednost *null*. Dodatna cena ovoga je minimalna pošto samo jedan karakter (‘?’) na kraju imena tipa označava da li promenljiva može biti *null* (slika 3.4). Još jedan tip izuzetaka koje *Kotlin* izbegava je *ClassCastException*, do koga dolazi kada se pokuša kastovanje objekta bez prethodne provere njegovog tipa. Ovaj izuzetak se sprečava tako što je dovoljno izvršiti samo proveru tipa, bez eksplicitnog kastovanja nakon toga. Ukoliko provera prođe uspešno, promenljiva se može koristiti kao objekat odgovarajućeg tipa (slika 3.5) [16].

```
val s1: String? = null
val s2: String = ""
```

Slika 3.4: Sprečavanje izuzetka *NullPointerException*

```
if (value is String) {
   print(value.toUpperCase())
}
```

Slika 3.5: Sprečavanje izuzetka *ClassCastException*

3.2.4. Interoperabilnost

Interoperabilnost *Kotlina* sa *Javom* ogleda se u mogućnosti korišćenja svih biblioteka pisanih u *Javi*. Mogu se pozivati metode, nasleđivati klase, implementirati interfejsi, korisititi anotacije i drugo iz *Jave*. Za razliku od nekih drugih jezika za *JVM*, interoperabilnost *Kotlina* ide korak dalje i omogućava i pozivanje kôda pisanog u *Kotlinu* iz *Jave*. Ovim se dozvoljava fleksibilno kombinovanje *Jave* i *Kotlina* gde god je to potrebno, a samim tim i postepeno prebacivanje postojećih projekata sa *Jave* na *Kotlin* [16].

3.3. Osnove programskog jezika Kotlin

3.3.1. Tipovi i deklaracije promenljivih

Kotlin je statički tipiziran jezik, ali ne zahteva eksplicitno navođenje tipa promenljive u svim situacjiama. Nekada se na osnovu vrednosti dodeljene promenljivoj može zaključiti o kom tipu je reč. Ukoliko je, ipak, potrebno navesti tip promenljive, to se radi tako što se nakon imena promenljive unesu ':' nakon čega sledi tip. U situaciji kada se promenljiva ne inicijalizuje odmah pri deklaraciji, neophodno je navesti njen tip jer nije moguće zaključiti ga.

U *Kotlinu* svaka promenljiva je objekat, odnosno ne postoje primitivni tipovi (*byte*, *short*, *int*, *long*, *float*, *double*, *char*, *boolean*). Ovo znači da se nad svakom promenljivom mogu pozivati metode (*member functions*) i svojstva. *Kotlin* obezbeđuje skup ugrađenih tipova koji predstavljaju primitivne tipove. Za cele brojeve, to su *Byte*, *Short*, *Int* i *Long*. *Byte* obuhvata brojeve iz opsega $[-2^7, 2^7 - 1]$, *Short* iz opsega $[-2^{15}, 2^{15} - 1]$, *Int* $[-2^{31}, 2^{31} - 1]$ i *Long* $[-2^{63}, 2^{63} - 1]$. Kada se promenljiva inicijalizuje bez eksplicitno zadatog tipa, kompilator će automatski zaključiti da se radi o promenljivoj tipa *Int*, osim ako njena vrednost izlazi iz opsega ovog tipa. U tom slučaju je *Long*. Takođe, ako se tip navede pri inicijalizaciji, kompilator će proveriti da li je vrednost promenljive u opsegu navedenog tipa [18]. Pored navedenih tipova, postoje i njihove neoznačene verzije UByte $[0, 2^8 - 1]$, UShort $[0, 2^{16} - 1]$, UInt $[0, 2^{32} - 1]$ i ULong $[0, 2^{64} - 1]$ [17]. Za brojeve sa decimalnim zarezom, odgovarajući tipovi su *Float* i *Double*. Kompilator, u slučaju kada tip nije naveden, zaključuje da se radi o promenljivoj tipa *Double*, osim ako vrednost promenljive nema sufiks 'f' ili 'F'. *Kotlin* ne podržava implicitne konverzije tipova, na primer promenljiva tipa *Int* ne može biti prosleđena funkciji koja kao argument prima promenljivu tipa *Double*. Za to se koriste odgovarajuće funkcije (*toLong()*, *toDouble()*, *toChar()* i druge) [18]. *Boolean* je tip koji može imati dve vrednosti *true* ili *false* [19]. Karakteri se predstavljaju tipom *Char* [20].

Promenljive se u *Kotlinu* mogu deklarisati pomoću ključnih reči *var* i *val*. Kada se promenljiva deklariše sa *val*, ona je zapravo nepromenljiva, tj. konstantna. Vrednost joj se dodeljuje inicijalizacijom i nakon toga se ta vrednost može samo čitati, nikada se

ne menja. Ukoliko je potrebno menjati vrednost promenljive, ona se deklariše korišćenjem ključne reči *var*.

3.3.2. Kontrola toka

Naredba *if* omogućava izvršavanje jedne ili bloka naredbi samo ako je neki uslov ispunjen. Vitičaste zagrade su obavezne samo ako se radi o dve ili više naredbi. U *Kotlinu*, naredba *if* je izraz, vraća vrednost. Samim tim, ne postoji ternarni operator (*condition* ? *then* : *else*) jer je izraz *if* dovoljan za obavljanje njegove uloge. Ako se *if* koristi kao izraz, mora se korisiti zajedno sa granom *else*. Takođe, ako se radi o bloku naredbi, poslednji izraz u bloku predstavlja vrednost bloka [21]. Primeri korišćenja su na slici 3.6.

```
var max = a
if (a < b) max = b

if (a > b) {
   max = a;
} else {
   max = b
}

max = if (a > b) a else b

max = if (a > b) {
   print("a is greater than b")
   a
} else {
   print("b is greater than a")
   b
}
```

Slika 3.6: Naredba *if* – primeri korišćenja

Naredba *when* definiše uslovni izraz sa više grana, a sličan je naredbi *switch* iz drugih programskih jezika (*C*, *C++*, *Java*). Kao i *if*, i naredba *when* se može koristiti i kao naredba kontrole toka i kao izraz. Kada se koristi kao izraz, važe ista pravila vezana

za vrednosti blokova i upotrebu grane *else*, kao kod naredbe *if* [21]. Na slici 3.7 prikazano je nekoliko načina korišćenja ovog izraza.

```
when (x) {
   1 -> print("x == 1")
   2 -> print("x == 2")
   else -> print("x != 1 && x != 2")
}

when (x) {
   in 1..10 -> print("x is in range")
   in validNumbers -> print("x is valid")
   !in 10..20 -> print("x is outside the range")
   else -> print("none of the above")
}

fun hasPrefix(x: Any) = when(x) {
   is String -> x.startsWith("prefix")
   else -> false
}
```

Slika 3.7: Naredba *when* – primeri korišćenja

Petlja *for* se koristi za iteraciju kroz opsege, nizove, kolekcije ili bilo šta što ima iterator [21]. Primeri korišćenja petlje *for* su prikazani na slici 3.8.

```
for (item in collection) print(item)

for (item: Int in ints) {
   print(item)
}

for (i in 1..3) {
   print(i)
}

for (i in array.indices) {
   print(array[i])
}
```

Slika 3.8: Petlja *for* – primeri korišćenja

Petlje *while* i *do-while* ponavljaju izvršavanje naredbe ili bloka naredbi sve dok je neki uslov ispunjen. Razlikuju se samo po vremenu provere ispunjenosti uslova. Kod petlje *while* se prvo proveri uslov pa izvršavaju naredbe, a kod petlje *do-while* se ispunjenost uslova proverava tek na kraju izvršavanja jedne iteracije [21].

3.3.3. Klase

Deklaracija klase u *Kotlinu* se sastoji od ključne reči *class*, imena klase, zaglavlja klase (parametri, njihovi tipovi, geteri, seteri i primarni konstruktor) i tela klase ograničenog vitičastim zagradama, s tim što su zaglavlje i telo klase opcioni. Takođe, ako klasa nema telo, ni vitičaste zagrade se ne moraju navoditi. Za primarni konstruktor se koristi ključna reč *constructor*, ali se može izostaviti u slučaju kada nema nikakvih dodatnih anotacija ili modifikatora pristupa. Primarni konstruktor ne može da sadrži nikakav kôd. Ukoliko je to potrebno, kôd za inicijalizaciju se može smestiti u telo klase unutar bloka *init*. Blokova *Init* može biti više u okviru jedne klase, a izvršavaju se prilikom inicijalizacije instance redosledom kojim su navedeni. Sekundarni konstruktori se definišu u okviru tela klase. Sekundarni konstruktori uvek moraju pozivati primarni konstruktor direktno ili preko drugog sekundarnog konstruktora, pomoću ključne reči *this* [22].

Sve klase u *Kotlinu* imaju zajedničku natklasu *Any*, koja ima tri metoda (*equals*, *hashCode* i *toString*). Klase u *Kotlinu* su podrazumevano finalne, ne mogu se nasleđivati. Da bi se klasa mogla naslediti, potrebno je obeležiti je ključnom rečju *open* [23]. Na slici 3.9 je prikazano nasleđivanje klasa u *Kotlinu*.

Modifikatori pristupa u *Kotlinu* su *private* – vidljivo unutar konkretne klase, *protected* – vidljivo unutar konkretne klase i potklasa, *internal* – vidljivo unutar modula i *public* – vidljivo svuda. Podrazumevana vidljivost je *public* [24].

```
open class Base(p: Int)
class Derived(p: Int) : Base(p)
```

Slika 3.9: Nasleđivanje

3.3.4. Funkcije

Funkcije se u *Kotlinu* deklarišu korišćenjem ključne reči *fun*. Pozivaju se na uobičajen način. Parametri se navode u formatu *name: type*, odvojeni su zarezima i tipovi im uvek moraju biti eksplicitno navedeni. Takođe, mogu imati podrazumevane vrednosti, a pri pozivanju funkcija, mogu se i imenovati, što je korisno u slučaju kada funkcija ima više parametara. Na ovaj način se pri pozivanju ne mora poštovati redosled iz deklaracije funkcije [25]. Na slici 3.10 je prikazan primer funkcije sa ukupno četiri parametra, od kojih tri imaju podrazumevane vrednosti, kao i tri različita poziva ove funkcije.

Kako su promenljive svih tipova objekti, operacije nad njima su zapravo pozivi funkcija (slika 3.11) [26].

```
fun foo(
    s: String,
    b1: Boolean = true,
    b2: Boolean = true,
    c: Char = ' ',
) { /*...*/ }

foo(
    "foo foo foo",
    false,
    c = '*',
    b2 = true
)

foo("foo foo foo")

foo(
    s = "foo foo foo",
    b2 = false,
    c = '_'
)
```

Slika 3.10: Primer funkcije u *Kotlinu*

Izraz	Metod
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a++	a.inc()
a--	a.dec()
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a += b	a.plusAssign(b)

Slika 3.11: Operatori u *Kotlinu* i funkcije kojima su oni interno predstavljeni

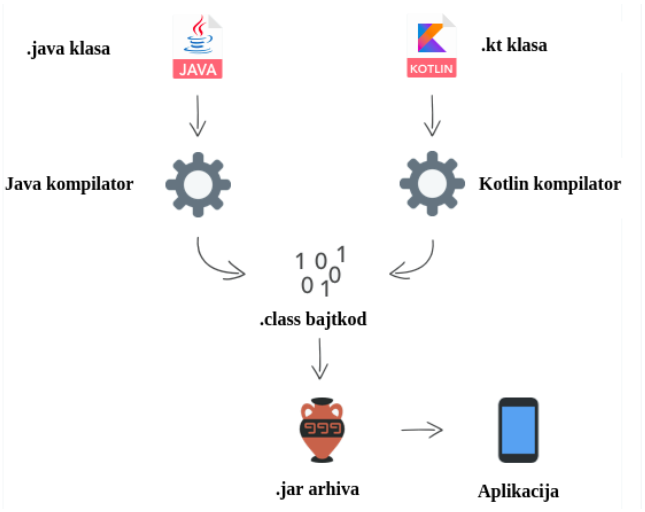
3.4. Poređenje sa Javom

Kada se program piše u *Javi*, klase (*.java*) se kompiliraju od strane *Java* kompilatora (*javac*) koji ih transformiše u datoteke (*.class*) koje sadrže bajtkod. Tako kompiliran kôd se može izvršavati na bilo kojoj platformi koja podržava *JVM* ("piši jednom, pokreći svuda"). Međutim, i dalje je potrebna *JVM* koja je specifična za platfomu. Kôd pisan u *Kotlinu* obično se nalazi u datotekama sa ekstenzijom *.kt*. Kompilator za *Kotlin*, *kotlinc*, analizira izvorni kôd i generiše datoteke .class. Dakle, pri kompilaciji kôda pisanog u *Kotlinu*, dešava se gotovo ista situacija, kao što je i prikazano na slici 3.12. Ono u čemu se *Kotlin* razlikuje je to što se može izvršavati na različitim platformama kao što je *JVM*, *JS* ili direktno na nativnim platformama [11, 16].

Kotlin je statički tipiziran programski jezik, što znači da je tip svake promenljive poznat u vreme kompilacije. Ovo omogućava razvojnom okruženju da upozori na potencijalne greške, na primer dodeljivanje brojevne vrednosti promenljivoj koja je tipa *String*. Statička tipiziranost jezika ima i prednosti i mane u odnosu na dinamički tipizirane jezike kod kojih se tip promenljivih određuje tek prilikom izvršavanja. Kod dinamički tipiziranih jezika je kôd kraći, ali je potrebno više memorije za određivanje

tipa promenljive prilikom izvršavanja. Međutim, za razliku od *Jave*, u *Kotlinu* nije neophodno uvek eksplicitno navesti tip promenljive i to u slučajevima kada se promenljiva odmah i inicijalizuje. Tada se tip može zaključiti na osnovu vrednosti dodeljene promenljivoj. Dakle, osnovne prednosti statički tipizarnih jezika su:

1. **Performanse** – kako je tip poznat u vreme kompilacije, nema potrebe da se on određuje kasnije u vreme izvršavanja.
2. **Pouzdanost** – kompilator prijavljuje potencijalne greške u vreme kompilacije što smanjuje mogućnost grešaka pri izvršavanju.
3. **Efikasnost** – jednostavnije je refaktorisati kôd i moguće je koristiti neke alate u okviru razvojnog okruženja kao što je automatsko dopunjavanje kôda.



Slika 3.12: Proces kompilacije kôda pisanog u *Javi* i *Kotlinu*

Kao i *Java*, *Kotlin* je objektno-orijentisan programski jezik, međutim, *Kotlin* je istovremeno i funkcionalni programski jezik. To znači da se funkcije mogu koristiti kao promenljive (mogu se čuvati kao promenljive, prosleđivati kao parametri ili biti povratne vrednosti drugih funkcija), kao što je slučaj u programskom jeziku *JavaScript* [27]. Funkcionalni pristup donosi i "nepromenljive promenljive" čije se vrednosti ne menjaju nakon što su jednom napravljene. Njihovo korišćenje obezbeđuje pouzdanost i bolje performanse programa, čineći ga otpornim na probleme vezane za konkurentno programiranje (istovremeni pristup i izmena vrednosti promenljive iz različitih niti).

Neke od ključnih razlika između *Jave* i *Kotlina* su [28]:

1. **Konciznost** – *Java* prati tradicionalni pristup pisanja opširnog kôda, dok *Kotlin* primenjuje principe konciznosti. Zahvaljujući konciznosti, kôd je kraći, čitljiviji i manje podložan greškama.
2. **Sigurnost** – *Kotlin* ne dozvoljava dodeljivanje vrednosti *null* promenljivim koje nisu označene kao promenljive kojima je moguće dodeliti vrednost *null*. Ovim *Kotlin* nudi veću stabilnost u odnosu na *Javu* kod koje je moguće proizvesti *NullPointerException*.
3. **Funkcije proširenja** – Kada je u *Javi* potrebno proširiti neku funkciju, mora se naslediti odgovarajuća klasa i onda *override*-ovati funkcija, dok *Kotlin* dolazi sa opcijom proširivanja funkcija bez nasleđivanja klasa.
4. **Zaključivanje tipova** – U *Kotlinu* nije uvek neophodno navesti tip promenljive pri deklaraciji. Međutim, u takvoj situaciji, potrebno je da promenljiva bude inicijalizovana kako bi se tokom kompilacije mogao odrediti njen tip.
5. **Kastovanje** – U *Javi* je potrebno proveriti tip objekta da bi kastovanje u željeni tip prošlo bez izuzetka. Međutim u *Kotlinu*, nakon provere tipa, nije potrebno eksplicitno kastovati objekat.
6. **Funkcionalno programiranje** – *Java* je objektno-orijentisani jezik, a *Kotlin* ima osobine i objektno-orijentisanih i funkcionalnih programskih jezika.
7. **Korutine** – *Java* dozvoljava pravljenje više niti koje se u pozadini dugo izvršavaju i obavljaju složene operacije, dok *Kotlin*, pored niti, podržava i korutine zahvaljujući čemu je moguće obustaviti izvršavanje blokirajućih funkcija.
8. **Zapečaćene klase** – Od samog početka *Kotlin* podržava zapečaćene (*sealed*) klase, dok su u *Javi* prisutne tek od verzije 15. U *Javi* samo klase iz istog modula mogu naslediti zapečaćenu klasu, a zapečaćena klasa mora da definiše koje klase je mogu naslediti pomoću ključne reči *permits*. U *Kotlinu*, zapečaćenu klasu može naslediti bilo koja klasa koja je u istom faju.
9. **Vreme kompilacije** – *Java* se kompilira za 15-20% brže od *Kotlina*.
10. **Kompatibilnost sa platformama** – *Java* se kompilira u bajtkod za *JVM*, a *Kotlin* u bajtkod za *JVM*, *JS* ili mašinski kôd za specifičnu platformu.

11. **Slučajevi upotrebe** – *Java* je jedan od najpopularnijih jezika opšte namene pa se često koristi za samostalne ili za *back-end* aplikacije. Kotlin je takođe popularan jezik opšte namene, ali najčešće se koristi za *Android* aplikacije.

3.5. Kotlin i višeplatformsko programiranje

Jedan od ključnih benefita korišćenja *Kotlina* je podrška višeplatformskom programiranju. Time se smanjuje vreme utrošeno na pisanje i održavanje istog kôda za različite platforme, a pritom se zadržavaju fleksibilnost i benefiti nativnog programiranja.

Postoji nekoliko slučajeva upotrebe *Kotlina* za višeplatformsko programiranje:

1. ***Android* i *iOS* aplikacije** – Deljenje kôda između mobilnih platformi je jedan od najčešćih slučajeva upotrebe višeplatformskog programiranja u *Kotlinu*. Pomoću *KMM* (*Kotlin Multiplatform Mobile*), mogu se pisati višeplatformske aplikacije, kao i kôd koji je zajednički za *Android* i *iOS*.
2. ***Full-stack* veb aplikacije** – Još jedan scenario u kome deljenje kôda može doneti benefite je povezana aplikacija kod koje se logika može koristiti i na klijentskoj i na serverskoj strani. *KMM* ovo takođe pokriva.
3. **Višeplatformske biblioteke** – Višeplatformsko programiranje u *Kotlinu* je korisno i za one koji se bave pisanjem biblioteka. Moguće je napraviti višeplatformsku biblioteku sa deljenim kôdom i kôdom specifičnim za *JVM*, *JS* ili nativnu platformu. Ovakve biblioteke se mogu koristiti u drugim višeplatformskim aplikacijama.
4. **Deljeni kôd za mobilne i veb aplikacije** – Još jedan popularan slučaj upotrebe višeplatformskog programiranja u *Kotlinu* je pisanje zajedničkog kôda za *Android*, *iOS* i veb aplikacije.

Kako bi izvršavanje kôda pisanog na standardnom *Kotlinu* bilo moguće na različitim platfomama, *Kotlin* obezbeđuje kompilatore koji su specifični za platforme, kao i biblioteke kao što su *Kotlin/JVM*, *Kotlin/JS* i *Kotlin/Native* (slika 3.13).

Standardni *Kotlin* uključuje sam jezik i osnovne biblioteke i alate. Kôd napisan na standardnom *Kotlinu* se izvršava na svim platformama. Pomoću višeplatformskih *Kotlin* biblioteka, višeplatformska logika se može iznova koristiti u standardnom kôdu i u kôdu specifičnom za platformu. Standardni kôd se može osloniti na skup biblioteka koje pokrivaju uobičajene zadatke (*HTTP*, serijalizacija, korutine i drugo). Da bi se kôd prilagodio specifičnoj platformi, koriste se verzije *Kotlin/JVM*, *Kotlin/JS* i *Kotlin/Native* koje uključuju dodatke standardnom jeziku i biblioteke i alate specifične za platformu. Kroz ove verzije *Kotlina* moguće je pristupiti kôdu nativnom za platformu i iskoristiti sve njegove mogućnosti. Mehanizmom očekivanih (*expect*) i pravih (*actual*) deklaracija se iz deljenog/zajedničkog kôda može pristupiti *API*-jima specifične platforme.

Slika 3.13: Kompilacija kôda pisanog na *Kotlinu* za različite platforme

Trenutna *Kotlin/JS* implementacija se odnosi na *ES5* i može se koristiti za pisanje *front-end*, *back-end* i *serverless* aplikacija, kao i za biblioteke.

Kotlin/Native je tehnologija za kompilaciju kôda napisanog u *Kotlinu* u binarni koji se može izvršavati bez virtualne mašine. Namenjen je uređajima na kojima virtualne mašine nisu moguće ili poželjne (*embeded* ili *iOS*) [29].

3.6. KMM (Kotlin Multiplatform Mobile)

KMM je *SDK* (*Software Development Kit*) napravljen da pojednostavi pisanje višeplatformskih mobilnih aplikacija, tako što se kôd koji je specifičan za platformu, piše samo kada je to neophodno [30].

Zajednički kôd se kompilira u *JVM* bajtkod pomoću *Kotlin/JVM* za *Android*, odnosno u nativni *iOS* kôd pomoću *Kotlin/Native*. Dakle, *KMM* moduli se mogu koristiti kao bilo koja mobilna biblioteka. *KMM* je dizajniran tako da pomogne programerima da iznova koriste poslovnu logiku među mobilnim platformama, a da kôd specifičan za platformu pišu samo kada je neophodno, na primer, da bi se implementirao nativni korisnički interfejs ili kada se radi sa *API*-jem koji je specifačan za platformu. Za razvoj korisničkog interfejsa se mogu koristiti, na primer, *Jetpack Compose* za *Android*, a sa *iOS SwiftUI*. Moguće je deliti i samo sloj podataka, koristeći popularne biblioteke kao što su *Ktor* ili *SQLDelight*, ili sloj podataka i poslovne logike. Da bi se razvijale *iOS* aplikacije, potreban je *Xcode IDE*, tj. *MacOS*, dok se deo kôda za *Android* može razvijati i na operativnim sistemima *Windows* i *Linux*. *AndroidStudio* je potreban za razvoj *Android* i višeplatformskog modula aplikacije. Za bildovanje modula, potreban je *Gradle*, i svi *KMM* projekti koriste *KMM Gradle plugin*. Da bi se koristio *Gradle*, potrebno je instalirati *JDK* (*Java Development Kit*). Za *AndroidStudio* čak postoji i *KMM plugin* koji nudi i *iOS Emulator*, ali u njemu se može pokretati i testirati samo kôd deljenog modula [31].

3.6.1. Struktura KMM projekata

Nakon što se napravi novi *KMM* projekat, dobijemo *Gradle* projekat koji sadrži deljeni modul i *Android* aplikaciju kao potprojekte koji su povezani pomoću višeplatformskog mehanizma *Gradle* i ova dva modula su uključena u settings.gradle. Modul za *iOS* je u odvojenom folderu u *root* projetku. On koristi poseban *build* sistem i zato nije povezan sa ostalim delovima *KMM* projekta preko *Gradle*-a. Konfiguracija

samog *Gradle*-a se nalazi u okviru datoteke *build.gradle* na nivou projekta. Prikaz strukture *KMM* projekta je na slici 3.14.

Modul za *Android* je najjednostavniji zato što je po strukturi upravo ono na šta su programeri *Andoid* aplikacija već navikli. U njemu se, u okviru foldera *src*, nalazi *AndroidManifest* u koji se mogu uneti dozvole (na primer za koriscenje interneta i lokacije), aktivnosti, servisi i drugo. Pored toga, tu su paketi *java* i *res*. U *res* su resursi (na primer definicije ekrana - ukoliko se koristi *xml*, vrednosti nekih predefinisanih niski i boja), a u *java* je sam kôd i tu se uvek nalazi klasa *MainActivity*. U okviru modula *androdApp* se nalazi sav kôd koji je specifičan za *Android* i ne mora biti implementiran u okviru platforme *iOS*. Na primer *Activity* je koncept koji ne postoji u *iOS* aplikacijama, a za *Android* je potrebno da postoji bar *MainActivity*, tako da se to smešta u ovaj modul. Iz istog razloga se i implementacija korisničkog interfejsa smešta u ovaj modul. Bilo da se koristi *xml*, kao podrazumevani način definisanja korisničkog interfejsa za *Android* aplikacije (taj kôd se smešta u *res/layout*) ili *Jetpack Compose*. Taj kôd se smešta u ovaj modul, jer je korisnički interfejs specifičan za platformu i mora biti odrađen za svaku posebno. Takođe, u okviru modula za *Android* se nalazi i datoteka *build.gradle* u koju se uključuju biblioteke koje su potrebne samo za ovaj modul, na primer *Jetpack Compose*.

Folder *iosApp* je jedini u okviru projekta koji se može menjati samo iz razvojnog okruženja *Xcode*.

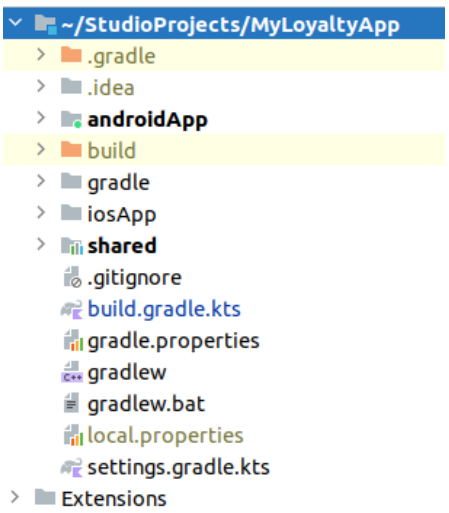
Najkompleksniji je modul *shared*. U njemu se nalazi paket *src* i datoteka *build.gradle*. U okviru foldera *src* se nalaze tri nova modula, tj. podmodula (*androidMain*, *commonMain* i *iosMain*).

Modul *commonMain* je modul koji sadrži sav deljeni kôd koji nema ništa specifično za bilo koju platformu. Na primer, ako bi se pisala klasa koja radi validaciju unete šifre, to bi bila čista poslovna logika aplikacije i ni na koji način ne zavisi od platforme. Zbog toga bi bilo poželjno pisati je samo jednom, dakle u ovom modulu. Na taj način, njoj će direktno moći da pristupe oba modula *androidApp* i *iosApp*.

Oba modula *androidMain* i *iosMain* koriste *Kotlin*. Dakle, *iosMain*, za razliku od modula *iosApp* ne koristi *Swift*. U ova dva modula se smeštaju klase koje su potrebne na obe platforme, ali se nešto u okviru njih razlikuje u zavisnosti od platforme. Na primer, kada se pravi novi *KMM* projekat, dobije se klasa *Platfrom* u sva tri ova modula

u okviru modula *shared*. U modulu *commonMain* je ova klasa deklarisana kao klasa *expect*, što je nešto poput već poznatog koncepta apstraktnih klasa. *Expect* je koncept koji je specifičan za *KMM* i svaka klasa *expect* dolazi sa odgovarajućim klasama *actual*. Ono što ključna reč *expect* označava jeste da se na obe platforme očekuje implementacija odgovarajućih klasa *actual*. Te klase se definišu upravo u ova dva modula, *androidMain* i *iosMain*. Ukoliko se na jednoj od platformi ne napravi implemetacija odgovarajuće klase *actual*, dolazi do greske.

U datoteci *build.gradle* u okviru modula *shared*, za sve podmodule se može dodati odgovarajuća biblioteka. Ovde se mogu koristiti samo biblioteke koje su napisane u *Kotlinu*, a ako je neka biblioteka pisana u *Javi*, neće biti moguće njeno korišćenje unutar podmodula *commonMain*. Na primer *Retrofit* i *Ktor Client* su biblioteke koje služe za pravljenje *HTTP* zahteva. *Ktor Client* je moguće koristiti pošto je cela napisana u *Kotlinu*, ali *Retrofit* nije. Međutim, moguće je praviti *Retrofit* pozive iz modula *androidMain* zato što će se ovo koristiti samo na *Android* platformi, pa se može koristiti i biblioteka pisana na *Javi*. Problem koji se u ovom slučaju javlja je taj što se mora pronaći i način za pravljenje *HTTP* zahteva iz modula *iosMain*. Ovo nije poželjno jer pišemo dve funkcije za jedan isti *HTTP* zahtev. Uvek se treba truditi da je što veći deo kôda zajednički i ne ponavlja se, odnosno, uvek treba pokušati sa čistim *Kotlin* bibliotekama, ukoliko postoje [32].



Slika 3.14: Struktura *KMM* aplikacije

3.7. MVVM (Model-View-ViewModel)

MVVM, takođe poznat i kao *Model-View-Binder*, predstavlja arhitekturalni softverski šablon koji odvaja programsku logiku od kontrola korisničkog interfejsa. Nastao je 2005. godine, a njegovi tvorci su *Ken Cooper* i *John Gossman*, arhitekte iz kompanije *Microsoft*. *MVVM* pomaže u organizaciji kôda i odvajanju programa u module, čineći razvoj, izmene i ponovno korišćenje kôda jednostavnijim i bržim. Može se koristiti sve dok su aplikacije dovoljno male tako da se odgovornosti različitih *ViewModel*-a mogu lako razdvojiti.

Implementacija šablona *MVVM* omogućava razdvajanje razvoja korisničkog interfejsa i poslovne logike aplikacije. To donosi odgovarajuće benefite, kao što je lakše testiranje aplikacije i ponovno korišćenje kôda. Takođe, omogućen je paralelan rad programera i dizajnera, pošto korisnički interfejs nije usko povezan sa kôdom.

Ključna stvar kod implementacije ovog šablona je razumevanje načina podele kôda u odgovarajuće klase, kao i razumevanje načina komunikacije tih klasa. Tri glavna dela u šablonu *MVVM* su *Model*, *View* i *ViewModel*.

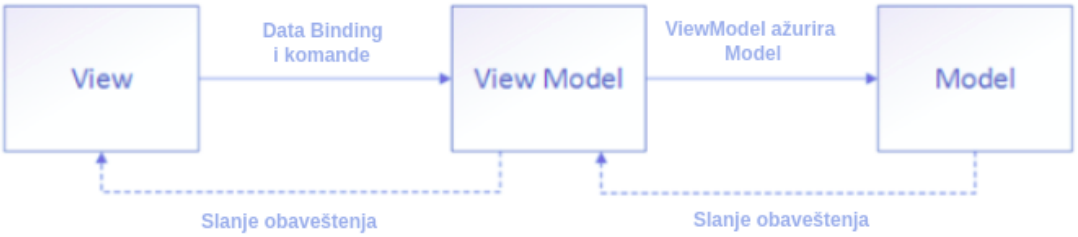
Između *Model*-a i *ViewModel*-a obavlja se upravljanje podacima, a između *ViewModel*-a i *View*-a, povezivanje podataka (*Data Binding*), kao što je prikazano na slici 3.15 [33].

Data Binding je tehnika jednosmernog ili dvosmernog povezivanja objekata sa korisničkim interfejsom. Kod jednosmernog povezivanja, kad god dođe do promene objekata u *ViewModel*-u, odgovarajuća svojstva *View*-a se ažuriraju (*One-Way*). Međutim, jednosmerno povezivanje može ići i u suprotnom smeru, od *View*-a ka *ViewModel*-u (*One-Way-To-Source*). Dvosmerno povezivanje omogućava da i *View* i *ViewModel* mogu međusobno posmatrati promene i automatski se ažurirati u skladu sa njima (*Two-Way*). Pored ova tri navedena tipa, postoji i četvrti koji od *ViewModel*-a šalje vrednosti ka *View*-u, i to se dešava samo jednom (*One-Time*) [34].

Model – Klase modela enkapsuliraju podatke aplikacije. Obično se koriste u kombinaciji sa servisima ili repozitorijumima koji sadrže obradu i pristup podacima. Kôd ovih klasa se može deliti među različitim platformama.

View – Odgovornost ovih klasa obuhvata definisanje strukture i izgleda onoga što će korisnicima biti prikazano na ekranu i sa čim će imati interakciju. Ove klase ne bi trebalo da sadrže tragove poslovne logike aplikacije. Često je kôd ovih klasa specifičan za platformu. Postoji više načina za izvršavanje kôda na *ViewModel*-u u zavisnosti od akcija koje se dese u *View* klasama. Na primer, dodir dugmeta ili odabir neke opcije.

ViewModel – Klase koje implementiraju svojstva i komande na koje *View* može da veže podatke (*Data Binding*). Takođe, *ViewModel* obaveštava *View* o eventualnim promenama stanja podataka. Svojstva i komande, koje *ViewModel* obezbeđuje, definišu funkcionalnosti koje se nude na korisničkom interfejsu, ali *View* određuje način na koji će one biti prikazane. *ViewModel* povezuje *View* sa potrebnim klasama *Model*-a i to najčešće jedan *View*, a više *Model*-a. Nekada je podatke potrebno izmeniti pre prikazivanja korisniku i to bi trebalo raditi u klasama *ViewModel*-a. Kôd ovih klasa se može iznova koristiti [33].



Slika 3.15: *Model-View-ViewModel*

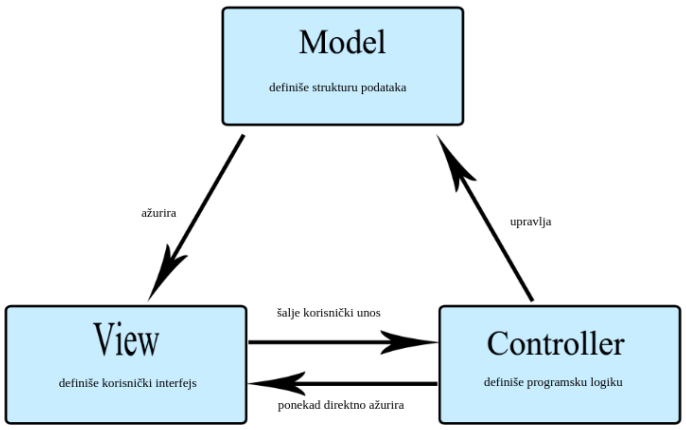
3.7.1. Poređenje sa arhitekturalnim softverskim šablonom MVC

Jedan od prvih arhitekturalnih softverskih šablona je *MVC* (*Model-View-Controller*). Prvi put je predstavljen 1979. godine od strane norveškog profesora *Trygve Reenskaug* koji je hteo da pronađe rešenje za razdvajanje složenih korisničkih aplikacija na manje komponente. *MVC* je prvi put korišćen u programskom jeziku *Small Talk*. Dugo se koristio za desktop aplikacije, a danas je zastupljen i među popularnim radnim okvirima za razvoj veb aplikacija (*Ruby on Rails*, *ASP.NET*, *Laravel*, *Angular*) [35].

Komponente ovog šablona su:

1. ***Model*** – odgovoran za podatke.
2. ***View*** – ono što krajnji korisnik aplikacije vidi.
3. ***Controller*** – obavlja programsku logiku i komunicira sa *Model*-om i *View*-om.

Razlika između šablona *MVC* i *MVVM*, ogleda se u komunikaciji između komponenti. Kod šablona *MVC*, nakon interakcije korisnika, *Controller* to obrađuje i onda komunicira sa *Model*-om i/ili *View*-om. Ukoliko dođe do promene podataka, *Model* o tome obaveštava *View* direktno [36]. Dakle, u šablonu *MVC, View* i *Model* mogu komunicirati i direktno, za razliku od šablona *MVVM* gde se njihova komunikacija dešava isključivo preko *ViewModel*-a. Na slici 3.16 se može videti način komunikacije komponenti šablona *MVC*.



Slika 3.16: *Model-View-Controler*

Pored već navedene razlike, *MVC* razdvaja aplikaciju na tri glavne logičke komponente, dok *MVVM* olakšava razdvajanje razvoja grafičkog korisničkog interfejsa. Takođe, kod šablona *MVC* jedan *Controller* može biti povezan sa više *View*-ova, a kod šablona *MVVM*, jedan *View* može biti povezan sa više *ViewModel*-a [37].

MVC bi trebalo koristiti kada je potrebno razdvojiti odgovornosti podataka (*Model*), upravljanja podacima (*Controller*) i prikaza podataka (*View*) unutar aplikacije. *MVVM* bi trebalo koristiti kada je potrebno deliti projekat sa dizajnerom i kada je

moguće razdvojiti dizajniranje i razvoj aplikacije, kada je potrebno imati komponente koje se mogu iznova koristiti ili kada je potrebno *unit* testiranje rešenja [38].

Zbog cirkularne prirode, šablon *MVC* je pogodan za razvoj monolitnih ili centralizovanih aplikacija, a naročito za desktop. Sa druge strane, šablon *MVVM* je pogodniji za razvoj slojevitih ili necentralizovanih aplikacija, kao što je slučaj sa veb ili aplikacijama za *Android*.

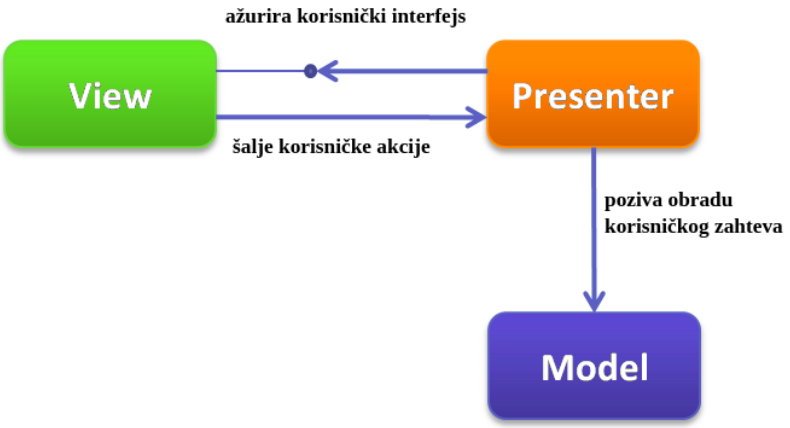
3.7.2. Poređenje sa arhitekturalnim softverskim šablonom MVP

MVP je arhitekturalni softverski šablon koji je nastao početkom devedesetih godina prošlog veka. Kompanija *Taligent* ga je prvo koristila u okviru programskog jezika *C++* [39]. *MVP* je sličan šablonu *MVC*, s tim što je *Controller* zamenjen *Presenter*-om [40].

Komponente ovog šablona su:

1. ***Model*** – skup klasa koje opisuju poslovnu logiku i podatke.
2. ***View*** – komponenta sa kojom korisnik ima direktnu interakciju.
3. ***Presenter*** – posrednik između *Model*-a i *View*-a. Prima korisnički ulaz od *View*-a, a uz pomoć *Model*-a obrađuje dobijene podatke i rezultat vraća *View*-u.

Presenter i *View* komuniciraju putem korisničkog intefejsa koji je definisan u *Presenter*-u, a implementiran u okviru komponente *View* (*Activity* ili *Fragment*, na primer). Dakle, *View* i *Presenter* su potpuno odvojeni jedan od drugog i komuniciraju samo preko interfejsa, zahvaljujući čemu je moguće *unit* testiranje aplikacije bez komponente *View*. Uvek je *View* komponenta koja započinje komunikaciju sa *Presenter*-om. *Presenter* obrađuje odgovarajuće korisničke akcije i podatke koje dobije od *View*-a, čeka odgovor od *Model*-a i kada odgovor stigne, na odgovarajući način ažurira *View* [40]. Na slici 3.17 prikazan je tok komunikacije komponenti šablona *MVP*.



Slika 3.17: *Model-View-Presenter*

Razlika šablona *MVP* i *MVVM* se ogleda u načinu komunikacije komponenti. *Presenter* ažurira *View* pozivanjem metoda, dok *ViewModel* šalje strimove podataka. U šablonu *MVP* jedan *View* može biti povezan samo sa jednim *Presenter-om*, a *View* u šablonu *MVVM* može biti povezan sa više *ViewModel*-a. *Presenter* ima refencu za *View*, a *ViewModel* i ne zna za *View* [41].

MVP ne bi trebalo koristiti kada je potrebno ažurirati korisnički interfejs bez prethodne akcije korisnika aplikacije. U ovom šablonu je ulazna tačka aplikacije *View*, tako da u slučaju kada je zbog promena na *Model*-u potrebno menjati *View*, bolje je koristiti *MVVM* [42].

Glava 4

4. Aplikacija MyLoyaltyApp

4.1. Opis aplikacije

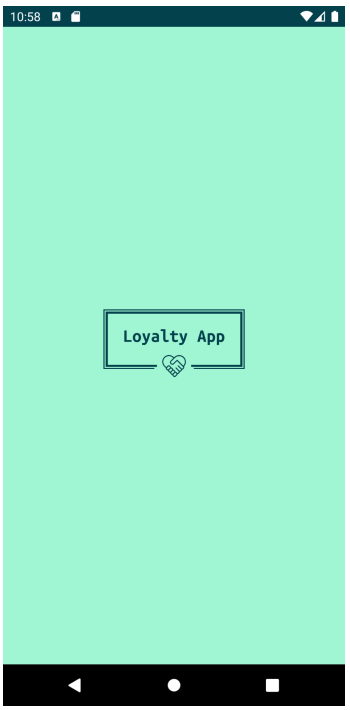
Aplikacija *MyLoyaltyApp* zamišljena je kao aplikacija za ostvarivanje programa lojalnosti za različite apotekarske ustanove. U apotekarstvu je konkurencija velika i svako želi da zadrži kupce nudeći im razne pogodnosti. Neko to realizuje kroz davanje popusta, a neko kroz sakupljanje poena koji se kasnije mogu iskoristiti na kupovinu. Kako i potrošači žele da uštede i ostvare sve pogodnosti koje im se nude, ovo direktno vodi nagomilavanju plastičnih kartica u njihovim novčanicima. Danas kada gotovo svi imaju pametne mobilne telefone, jednom aplikacijom bi se moglo zameniti mnoštvo kartica.

Pravljenjem korisničkog naloga preko aplikacije, korisnici dobijaju jedinstveni kôd koji se može iskoristiti za sakupljanje bodova, kao i za ostvarivanje popusta pri kupovini u apotekama koje podržavaju aplikaciju. Načini sakupljanja i korišćenja bodova mogu biti različiti. Svaka apoteka može da odredi način funkcionisanja svog programa lojalnosti.

Aplikacija bi mogla ponuditi i dodatne mogućnosti kao što je provera stanja poena ili radnog vremena neke apoteke. Takođe, mogao bi se proveriti i način sakupljanja i iskorišćavanja sakupljenih poena, za svaku apoteku ponaosob.

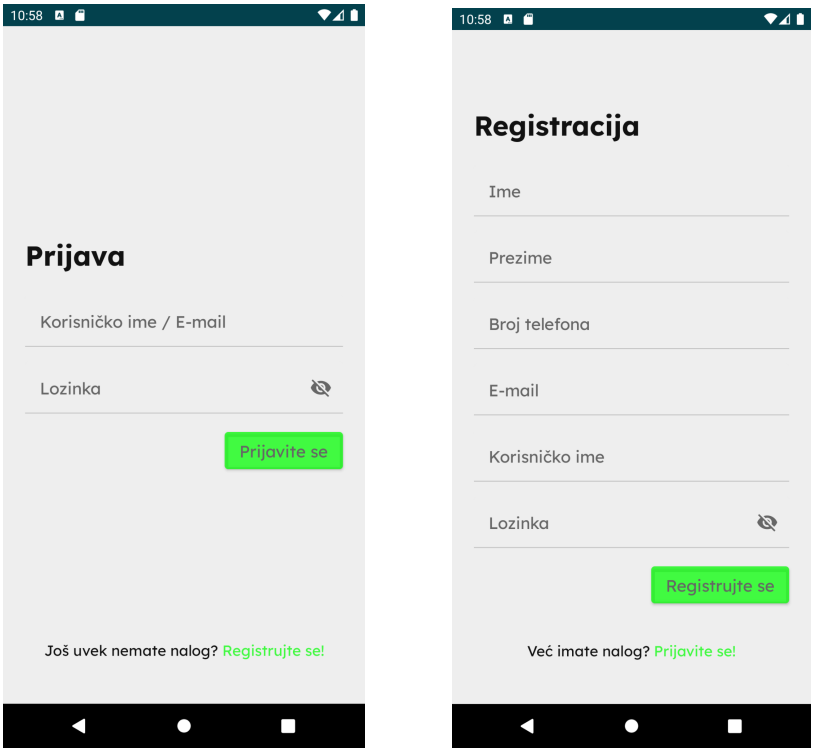
4.2. Korisnički interfejs i uputstvo za upotrebu

Prilikom pokretanja aplikacije, uvek se prvo prikaže početni ekran sa logom aplikacije prikazan na slici 4.1 (*Splashscreen*). Logo koji se nalazi na ovom ekranu ima i animaciju, na kratko se poveća i smanji, a prikaz ovog ekrana je postavljen na dve sekunde. Nakon isteka te dve sekunde, u zavisnosti od toga da li je korisnik već prijavljen ili nije, prikazuje se početna stranica ili stranica za prijavljivanje.



Slika 4.1: *Splashscreen* sa logom

Ukoliko korisnik već ima nalog, može se prijaviti pomoću korisničkog imena i lozinke. Umesto korisničkog imena se pri prijavi može korisiti imejl adresa. Na ekranu za prijavu se nalazi i link koji vodi ka stranici za registraciju. Ukoliko je aplikacija prvi put instalirana i korisnik nema već otvoren nalog, mora se registrovati. Za registraciju je potrebno uneti ime, prezime, broj telefona, imejl adresu, korisničko ime i lozinku. Nakon uspešne registracije, zahteva se prijavljivanje pomoću korisničkog imena ili imejl adrese i lozinke. Na slici 4.2 su prikazane stranice za prijavljivanje i registraciju korisnika.

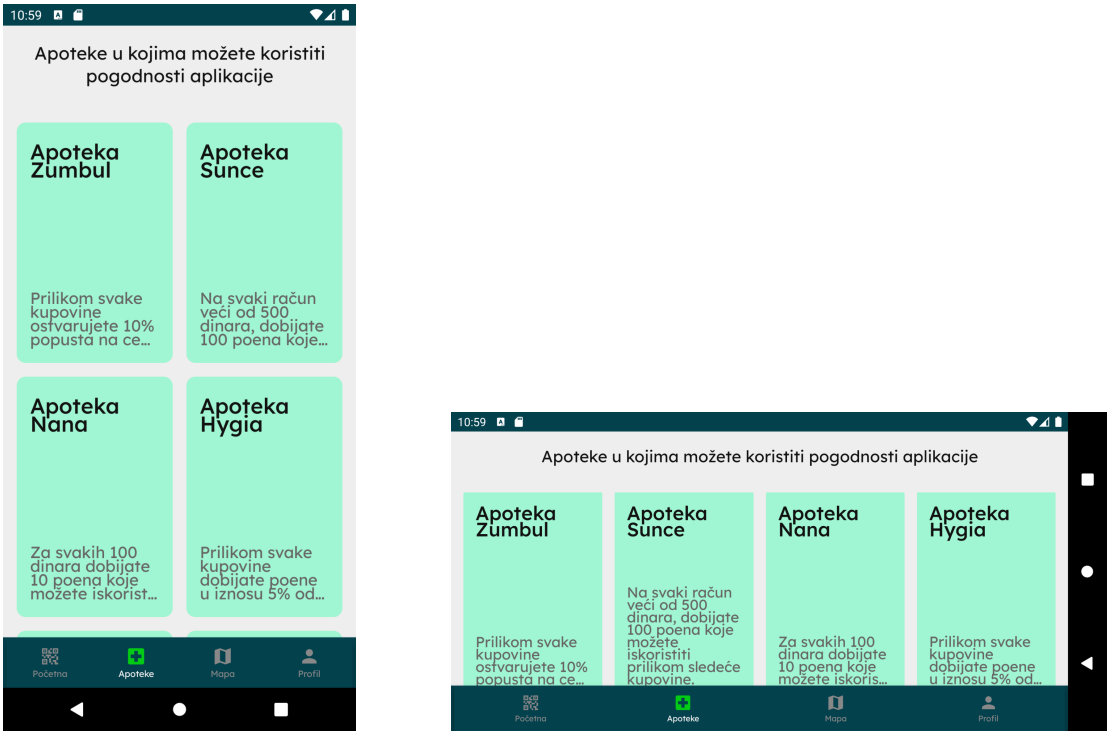


Slika 4.2: Ekrani za prijavu i registraciju korisnika

Nakon uspešne prijave, otvara se početna stranica na kojoj je prikazan jedinstveni identifikacioni *QR* kôd korisnika. Skeniranjem ovog kôda je obezbeđena identifikacija kupca prilikom kupovine u apoteci koja podržava aplikaciju. Na taj način će korisniku biti omogućeno ostvarivanje pogodnosti koje odgovarajući program lojalnosti nudi. Takođe, ova stranica se otvara odmah nakon početne stranice sa slike 4.1, ako je korisnik već prijavljen. Podatak na osnovu koga je poznato da li je korisnik već prijavljen je jedinstveni identifikacioni kôd korisnika na osnovu koga se generiše odgovarajući *QR* kôd. Prikaz ovog ekrana je dat na slici 4.3.

Odabirom odgovarajućeg polja u navigaciji na dnu ekrana, otvara se stranica na kojoj su prikazane sve apoteke koje podržavaju aplikaciju. Pored samog spiska apoteka, ovde je moguće videti i detaljne uslove i pravila korišćenja programa lojalnosti svake od njih. Na ekranima različitih veličina (na primer na telefonima i tabletima, ili pri promeni orijentacije na telefonu) postoji razlika u organizaciji elemenata ovog ekrana. Na slici 4.4 je prikaz ovog ekrana za male i srednje ekrane.

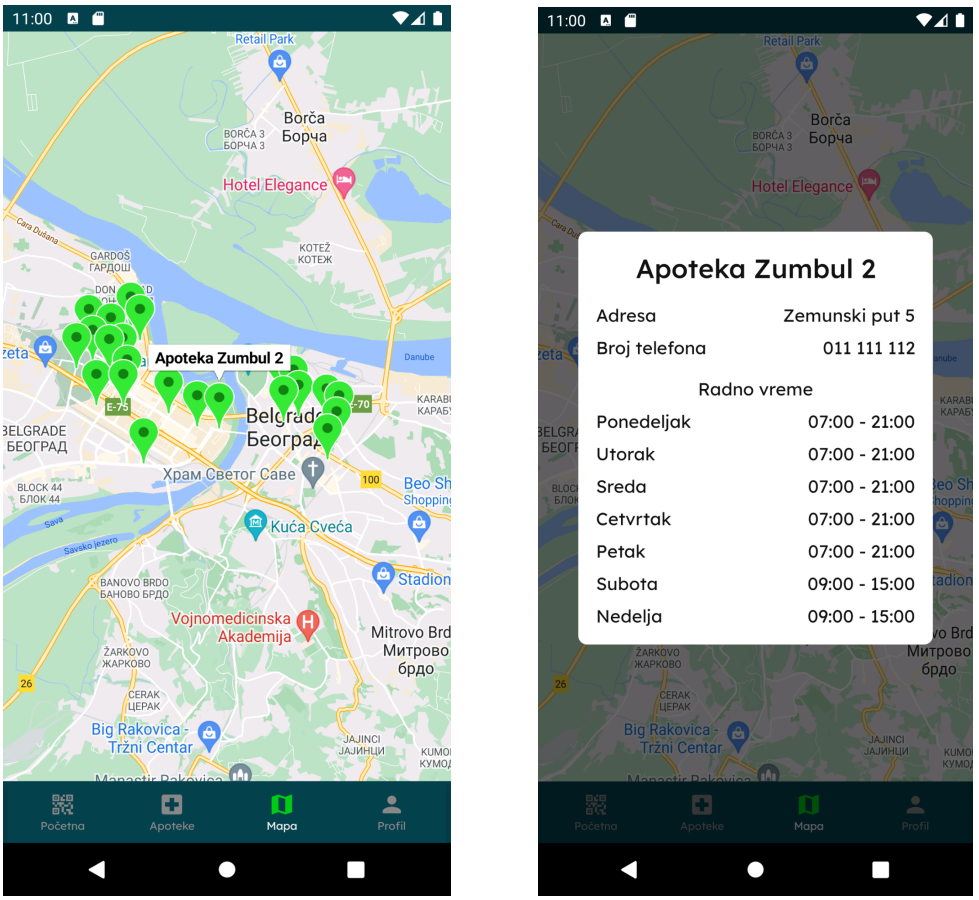
Slika 4.3: Početni ekran aplikacije sa *QR* kôdom



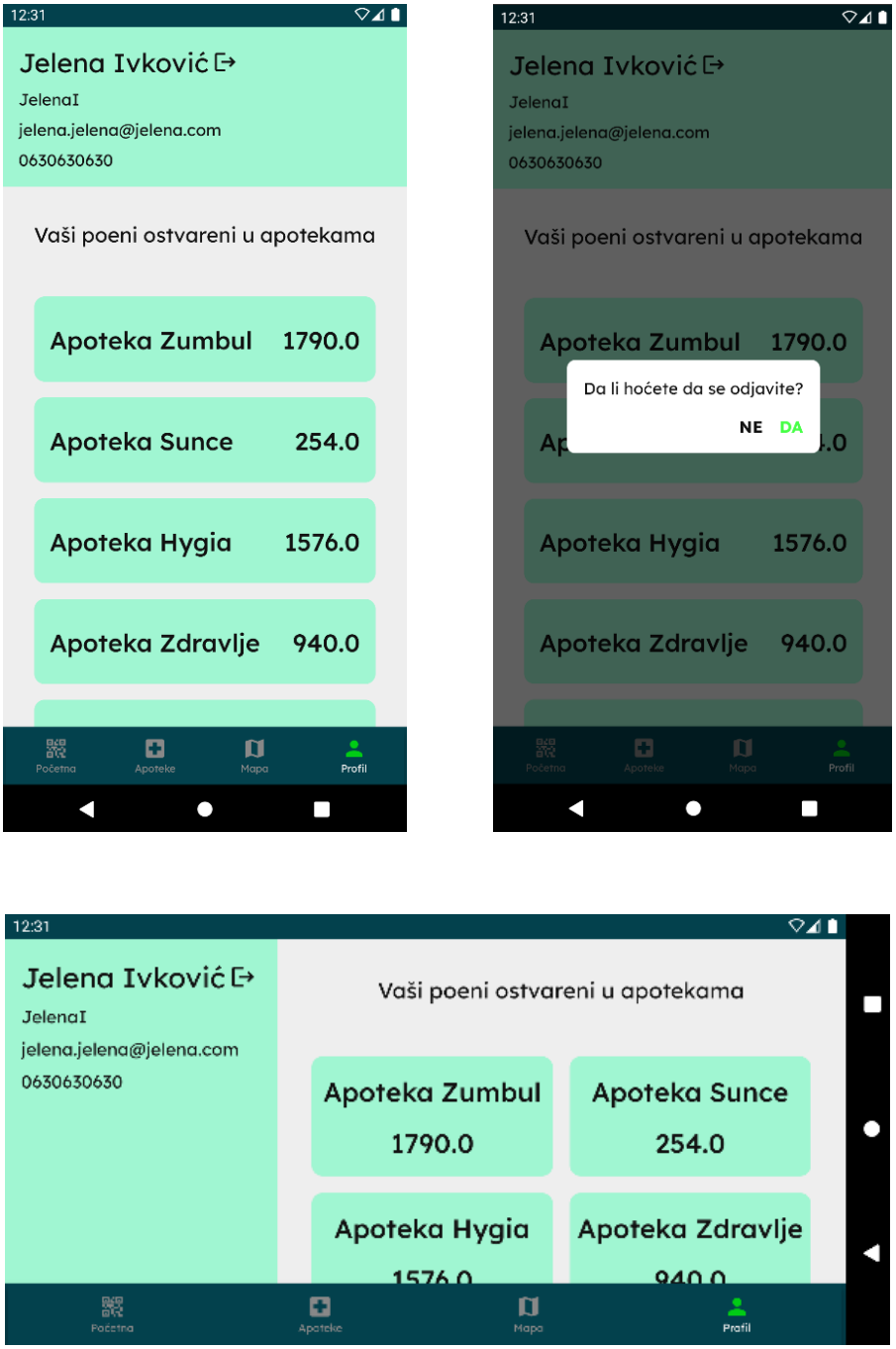
Slika 4.4: Spisak apoteka – mali i srednji ekran

Naredno polje u navigaciji vodi na mapu na kojoj su označena sva prodajna mesta svih apoteka koje podržavaju aplikaciju. Kada se dodirne oznaka na ekranu, pojavi se prozorčić sa nazivom konkretnog prodajnog mesta. Kada se on dodirne, otvara se novi prozor na kome su prikazani dodatni podaci: tačna adresa, broj telefona i radno vreme. Na slici 4.5 je prikazan izgled ovog ekrana.

Poslednje polje u navigaciji vodi na stranicu sa korisničkim profilom. Pored osnovnih podataka o prijavljenom korisniku (ime, prezime, korisničko ime, imejl i broj telefona), na ovoj stranici se nalazi i spisak sa trenutnim stanjem ostvarenih poena po apotekama. Takođe, ovde korisnik može da se odjavi iz aplikacije. Prikaz ovog ekrana dat je na slici 4.6.



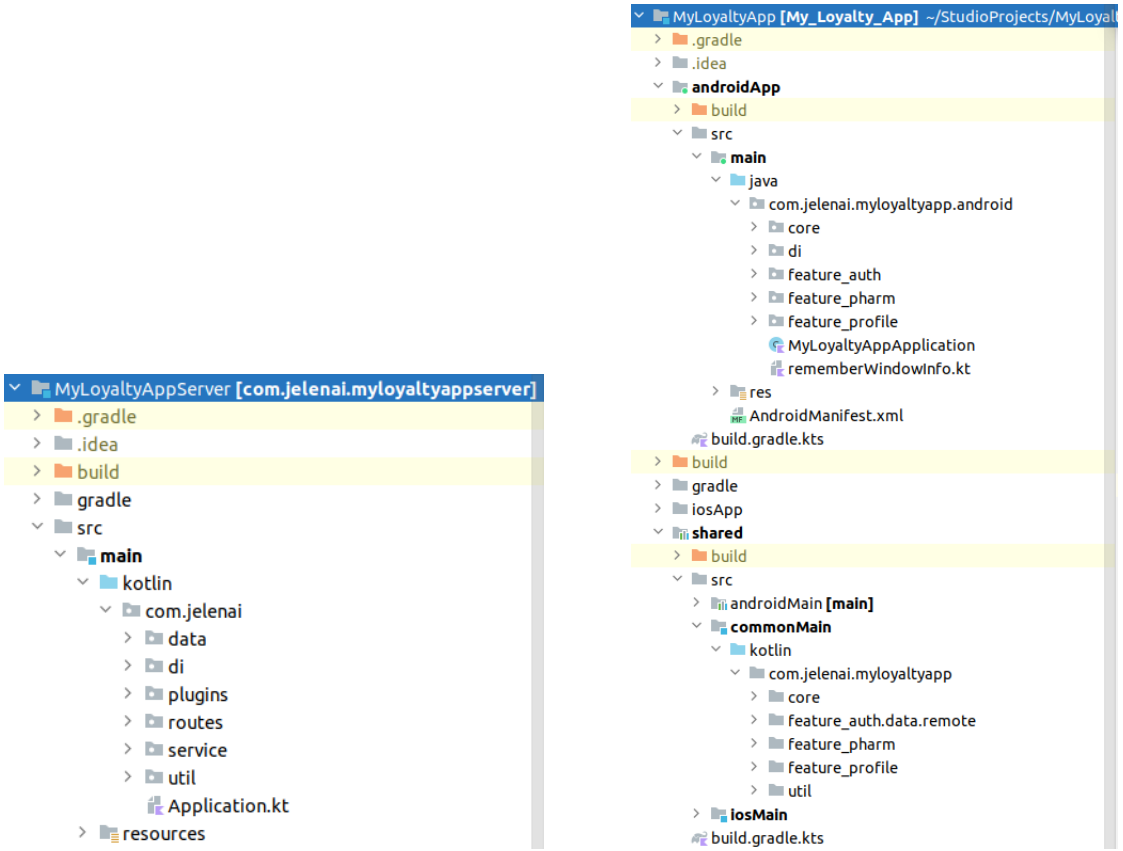
Slika 4.5: Mapa sa svim prodajnim mestima apoteka i detaljima



Slika 4.6: Korisnički profil – mali i srednji ekran

4.3. Struktura projekta

Aplikacija razvijena u okviru ovog rada se sastoji iz dva dela. *MyLoyaltyApp* predstavlja klijentsku višeplatformsku aplikaciju, s tim da je razvijena samo verzija za platformu *Android*. Drugi deo je *back-end* aplikacija sa bazom podataka, *MyLoyaltyAppServer*. Obe aplikacije su u potpunosti pisane na programskom jeziku *Kotlin*. Za čuvanje podataka na *back-end* delu se koristi baza podataka *MongoDB*, dok *front-end* aplikacija za *Android* ne koristi baze podataka, već se potrebni podaci čuvaju u *SharedPreferences*. Korisnički interfejs *Android* aplikacije je takođe napisan u *Kotlinu* koristeći *Jetpack Compose*. Na adresi https://github.com/JelenaI/MyLoyaltyApp se nalazi kôd aplikacije *MyLoyaltyApp*, a na https://github.com/JelenaI/MyLoyaltyAppServer je *MyLoyaltyAppServer*. Na slici 4.7 prikazane su strukture oba dela projekta.

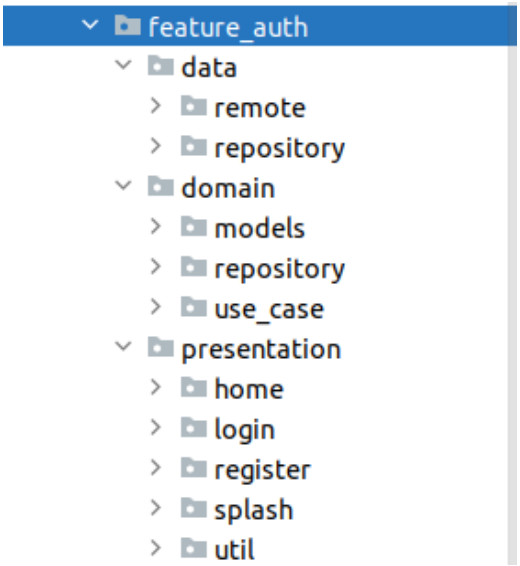


Slika 4.7: Struktura aplikacije *MyLoyaltyAppSever* i *MyLoyaltyApp*

Aplikacija *MyLoyaltyApp* je napravljena kao *KMM*-aplikacija, kombinujući jednostavnu i *MVVM* arhitekturu. Standardnim elementima šablona *MVVM*, *Model*, *View* i *ViewModel*, dodaje se još jedan sloj, a to su slučajevi upotrebe. Unutar ovog sloja se nalazi jedan deo logike aplikacije, a drugi deo je unutar *ViewModel*-a. Struktura modula *shared* i *androidApp* je ista, s tim da se u modulu *shared* nalaze sve klase modela, koje je bilo moguće smestiti u deljeni modul.

Kôd je grupisan u nekoliko paketa, tako da svaki predstavlja po jedan deo (*feature*) aplikacije, a to su autentikacija, apoteke i profil. Pored ove tri celine, neke opšte klase se nalaze u paketima *core* u modulu *androidApp* i *util* u modulu *shared*. Takođe, u okviru modula *androidApp* se nalazi paket *di* u kome su definisani moduli koji se koriste za umetanje zavisnosti (*Dependency Injection*).

Na primeru jedne celine (autentikacija – registracija) biće objašnjen način rada i tok podataka unutar aplikacije. U okviru paketa *feature_auth* nalaze se klase potrebne za obavljanje prijave i registracije korisnika podeljene u tri paketa. To su *presentation* (u kome se nalaze *View*, *ViewModel*), *domain* (slučajevi upotrebe, repozitorijumi, *Model*) i *data* (implementacija repozitorijuma, interfejsi za *API*, klase podataka *DTO*). Ova celina sadrži stranice za prijavu, registraciju, početnu stranicu (*splashscreen*) i početnu stranicu sa *QR* kôdom. Na slici 4.8 je prikazana struktura paketa *feature_auth*.



Slika 4.8: Struktura jedne celine aplikacije

U okviru paketa *data* se nalaze paketi *remote* i *repository*. U paketu *remote* se nalazi interfejs *AuthApi* u kome su sve funkcije koje se koriste za slanje *HTTP* zahteva *back-end* aplikaciji, a tiču se ove celine (slika 4.9). Funkcije ovog interfejsa su označene ključnom rečju *suspend*, što znači da se mogu pozivati samo iz neke druge funkcije obeležene sa *suspend* ili u okviru korutine. Pored toga, interfejs *AuthApi* ima i jedan prateći objekat (*companion object*). U okviru pratećih objekata se definišu promenljive i funkcije koje se mogu koristiti bez postojanja instance odgovarajuće klase, dakle, poput polja i metoda obeleženih ključnom rečju *static* u *Javi*. Klase koje se koriste za slanje zahteva (*CreateAccountRequest, LoginRequest, BasicApiResponse, AuthResponse*) deo su modula *shared*. Za pravljenje *HTTP* zahteva je korišćena biblioteka *Retrofit* [43]. U paketu *repository* se nalaze implementacije repozitorijuma definisanih u paketu *domain/repository*.

```
interface AuthApi {
    @POST("/user/create")
    suspend fun register(
        @Body request: CreateAccountRequest
    ): BasicApiResponse<Unit>

    @POST("/user/login")
    suspend fun login(
        @Body request: LoginRequest
    ): BasicApiResponse<AuthResponse>

    @GET("/user/authenticate")
    suspend fun authenticate()

    companion object {
        const val BASE_URL = "http://10.0.2.2:8100/"
    }
}
```

Slika 4.9: Interfejs *AuthApi*

U okviru paketa *domain* nalaze se paketi *models*, *repository* i *use_case*. U paketu *repository* je definicija repozitorijuma, tj. interfejs *AuthRepository* prikazan na slici 4.10. U okviru njega se za svaku funkciju iz interfejsa *AuthApi* definiše po jedna funkcija. Implementacija ovog interfejsa se nalazi u paketu *data/repository*. U klasi

AuthRepositoryImpl postoje dva polja, jedno je tipa *AuthApi*, a drugo *SharedPreferences*. Deo kôda ove klase, koji služi za pozivanje funkcije za registraciju korisnika, prikazan je na slici 4.11.

```
interface AuthRepository {
    suspend fun register(
        firstName: String,
        lastName: String,
        phoneNumber: String,
        email: String,
        username: String,
        password: String
    ): SimpleResource

    suspend fun login(
        email: String,
        password: String
    ): SimpleResource

    suspend fun authenticate(): SimpleResource
}
```

Slika 4.10: Interfejs *AuthRepository*

```
class AuthRepositoryImpl(
    private val api: AuthApi,
    private val sharedPreferences: SharedPreferences
) : AuthRepository {
    override suspend fun register(
        firstName: String,
        lastName: String,
        phoneNumber: String,
        email: String,
        username: String,
        password: String
    ): SimpleResource {
        val request =
            CreateAccountRequest(firstName, lastName, phoneNumber, email,
username, password)
        return try {
            val response = api.register(request)
            if (response.successful) {
                Resource.Success(Unit)
            } else {
                response.message?.let { msg ->
```

```
                    Resource.Error(UiText.DynamicString(msg))
                } ?:
Resource.Error(UiText.StringResource(R.string.error_unknown))
            }
        } catch (e: IOException) {

Resource.Error(UiText.StringResource(R.string.error_couldnt_reach_server))
        } catch (e: HttpException) {

Resource.Error(UiText.StringResource(R.string.something_went_wrong))
        }
    }
    /*...*/
}
```

Slika 4.11: Deo klase *AuthRepositoryImpl*

Rezultati koje vraćaju funkcije iz repozitorijuma se koriste u slučajevima upotrebe koji su definisani u okviru paketa *domain/use_case*. Slučajevi upotrebe imaju samo po jednu javnu funkciju koja izvršava sam slučaj upotrebe, tako da u slučaju ove celine postoje tri slučaja upotrebe: *AuthenticateUseCase*, *LoginUseCase* i *RegisterUseCase*. Na slici 4.12 je prikazan kôd slučaja upotrebe za registraciju.

```
class RegisterUseCase(
    private val repository: AuthRepository
) {
    suspend operator fun invoke(
        firstName: String,
        lastName: String,
        phoneNumber: String,
        email: String,
        username: String,
        password: String
    ): RegisterResult {
        val firstNameError = ValidationUtil.validateFirstName(firstName)
        val lastNameError = ValidationUtil.validateLastName(lastName)
        val phoneNumberError =
ValidationUtil.validatePhoneNumber(phoneNumber)
        val emailError = ValidationUtil.validateEmail(email)
        val usernameError = ValidationUtil.validateUsername(username)
        val passwordError = ValidationUtil.validatePassword(password)

        if (firstNameError == null
```

```
            && lastNameError == null
            && phoneNumberError == null
            && emailError == null
            && usernameError == null
            && passwordError == null
        ) {
            val result = repository.register(
                firstName.trim(),
                lastName.trim(),
                phoneNumber.trim(),
                email.trim(),
                username.trim(),
                password.trim()
            )

            return RegisterResult(result = result)
        }

        return RegisterResult(
            firstNameError = firstNameError,
            lastNameError = lastNameError,
            phoneNumberError = phoneNumberError,
            emailError = emailError,
            usernameError = usernameError,
            passwordError = passwordError
        )
    }
}
```

Slika 4.12: Klasa *RegisterUseCase*

Podatke koje dobiju od odgovarajućih repozitorijuma, slučajevi upotrebe prosleđuju dalje ka *ViewModel*-ima. Ono što je bitno kod slučajeva upotrebe je to da se mogu pozivati iz bilo kog *ViewModel*-a, što dodatno smanjuje pisanje istog kôda u različitim klasama.

ViewModel-i se nalaze u okviru paketa *presentation* koji je podeljen na dodatne pakete, po jedan za svaku stranicu koja je vezana za ovu celinu (*home*, *login*, *register*, *splash*). U paketu *register* se nalaze klase *RegisterViewModel*, *RegisterState*, *RegisterEvent* i *RegisterScreen*. Svrha *ViewModel*-a je održavanje stanja aplikacije, na primer pri rotaciji ekrana. Za svaki ekran čije stanje je potrebno pratiti postoji odgovarajuća *State* klasa, u ovom slučaju *RegisterState*. U okviru klase *ViewModel* su definisane funkcije koje odgovarajuće podatke dohvataju od slučajeva upotrebe, kao i

funkcije koje reaguju na događaje sa korisničkog interfejsa. Deo kôda klase *RegisterViewModel* je prikazan na slici 4.13.

```
@HiltViewModel
class RegisterViewModel @Inject constructor(
   private val registerUseCase: RegisterUseCase
) : ViewModel() {
   private val _firstNameState = mutableStateOf(StandardTextFieldState())
   val firstNameState: State<StandardTextFieldState> = _firstNameState

   private val _lastNameState = mutableStateOf(StandardTextFieldState())
   val lastNameState: State<StandardTextFieldState> = _lastNameState
   /*...*/

   fun onEvent(event: RegisterEvent) {
       when (event) {
           is RegisterEvent.Register -> {
               viewModelScope.launch {
                val registerResult = registerUseCase(
                  firstName = firstNameState.value.text,
                  lastName = lastNameState.value.text,
                  phoneNumber = phoneNumberState.value.text,
                  email = emailState.value.text,
                  username = usernameState.value.text,
                  password = passwordState.value.text
                )
               /*...*/
             }
           }
       /*...*/
    }
}
```

Slika 4.13: Deo klase *RegisterViewModel*

Poslednji deo predstavlja sam korisnički interfejs do koga podaci dolaze od *ViewModel*-a. Za implementaciju korisničkog interfejsa je korišćena biblioteka *Jetpack Compose* [44], pomoću koje se komponente korisničkog interfejsa predstavljaju kao funkcije pisane u *Kotlinu*. Deo kôda klase *RegisterScreen* je prikazan na slici 4.14. Klasa *Screen* sadrži po jedan objekat svake stranice aplikacije. Njima su pritom definisane i putanje koje se koriste prilikom navigacije sa jedne na drugu stanicu.

```
@Composable
fun RegisterScreen(
   navController: NavController,
   scaffoldState: ScaffoldState,
   onPopBackStack: () -> Unit,
   viewModel: RegisterViewModel = hiltViewModel()
) {
   val firstNameState = viewModel.firstNameState.value
   val lastNameState = viewModel.lastNameState.value
   val context = LocalContext.current
   /*...*/

   Box(
       modifier = Modifier
           .fillMaxSize()
           .padding(
               start = SpaceLarge,
               end = SpaceLarge,
               top = SpaceLarge,
               bottom = SpaceHuge
           )
   ) {
       /*...*/
       Row(
          Modifier.fillMaxWidth(),
          horizontalArrangement = Arrangement.End,
          verticalAlignment = Alignment.CenterVertically
       ) {
         Button(
           onClick = {
             viewModel.onEvent(RegisterEvent.Register)
           }
         ) {
           Text(
             text = stringResource(id = R.string.register_button_text)
             )
         }
      }
      /*...*/
   }
}
```

Slika 4.14: Deo funkcije *RegisterScreen*

Za umetanje zavisnosti (*Dependecy Injection*) korišćena je biblioteka *Dagger Hilt* [45]. Same zavisnosti su definisane u objektima koje su u okviru paketa *di*. Za

svaku celinu postoji po jedan objekat *AuthModule*, *PharmacyModule* i *ProfileModule*, kao i jedan objekat sa opštim zavisnostima *AppModule*. Na slici 4.15 je prikazan objekat *AuthModule*. Pre objekta *AuthModule* je potrebno navesti anotacije *@Module* i *@InstallIn(SingletonComponent::class)* koja označavaju da se radi o modulu i da će sve zavisnosti iz ovog modula postojati sve dok postoji i trenutna instanca aplikacije. Same funkcije modula se obeležavaju anotacijama *@Provides i @Singleton* kada je potrebno da postoji samo jedna instanca odgovarajuće zavisnosti. Ovde se mogu obezbediti zavisnosti *API*-ja, repozitorijuma, slučajeva upotrebe i drugo.

```
@Module
@InstallIn(SingletonComponent::class)
object AuthModule {
   @Provides
   @Singleton
   fun provideAuthApi(client: OkHttpClient): AuthApi {
       return Retrofit.Builder()
           .baseUrl(AuthApi.BASE_URL)
           .client(client)
           .addConverterFactory(GsonConverterFactory.create())
           .build()
           .create(AuthApi::class.java)
   }

   @Provides
   @Singleton
   fun provideAuthRepository(api: AuthApi, sharedPreferences:
SharedPreferences): AuthRepository {
       return AuthRepositoryImpl(api, sharedPreferences)
   }

   @Provides
   @Singleton
   fun provideRegisterUseCase(repository: AuthRepository): RegisterUseCase
{
       return RegisterUseCase(repository)
   }
   /*...*/
}
```

Slika 4.15: Objekat *AuthModule*

4.4. Migracija na iOS

Modul *iosApp*, odnosno deo aplikacije koji se odnosi na platformu *iOS* nije razvijen. Međutim, u okviru datoteke *buid.gradle* u modulu *shared*, može se videti da se ovaj modul kompilira u biblioteku za *Android* i u radni okvir (*framework*) za *iOS*. Za konfiguraciju biblioteke za *Andorid* se koristi dodatak (*plugin*) *android-library*.

Za *iOS* aplikaciju se deljeni modul kompilira u datoteku sa ekstenzijom *.framework* pomoću kompilatora *Kotlin/Native*. Konfiguracija ove datoteke se nalazi u *build.gradle* u deljenom modulu. Tu je definisan tip datoteke (*framework*) i njen naziv (*basename*). Pored ovoga, potrebno je napisati i zadatak za *Gradle* koji će datoteku *.framework* učiniti dostupnom razvojnom okruženju *Xcode*. Kada bi se razvijala aplikacija za *iOS*, radno okruženje bi pri svakoj kompilaciji dohvatalo najnoviju verziju datoteke *.framework*, koristeći ovaj zadatak za *Gradle*. Drugi način za integraciju aplikacije za *iOS* i deljenog modula je korišćenje dodatka za integraciju *CocoaPods* [46] koji enkapsulira detalje integracije i dozvoljava dodavanje zavisnosti deljenog modula kao bilo koje druge biblioteke.

U slučaju razvoja aplikacije za *iOS*, neki delovi aplikacije bi se morali implementirati posebno za ovu platformu. Kako se u deljenom modulu aplikacije *MyLoyltyApp* nalaze samo klase modela, pored korisničkog interfejsa bi bilo potrebno implementirati i komunikaciju sa serverskom aplikacijom i akcije na korisničke unose. Zamenom nekih od biblioteka, korišćenih u okviru modula *androidApp,* bibliotekama koje su napisane u Kotlinu, bilo bi moguće izdvajanje dodatnih delova kôda u deljeni modul. Na primer, biblioteka *Retrofit* bi se mogla zameniti bibliotekom *Ktor Client* [47], biblioteka *Dagger Hilt* bibliotekom *Koin* [48], svako korišćenje klase *IOException* bi se moralo zameniti klasom *RuntimeException* i drugo. Samim tim, potreba za dodatnom implementacijom za *iOS* bi se svela na najmanji mogući deo, a to je korisnički interfejs.

Serverska aplikacija *MyLoyaltyAppServer* je nezavisna u odnosu na aplikaciju *MyLoyaltyApp*. Samim tim, one ne zahteva nikakve dodatne izmene u slučaju razvoja aplikacije za *iOS*.

Glava 5

5. Zaključak

Pri donošenju odluke za način razvoja neke aplikacije, uvek se cilj aplikacije stavlja na prvo mesto. Dakle, odluka se donosi u zavisnosti od toga šta se želi postići. Ako je cilj napraviti aplikaciju tako da za kraće vreme bude dostupna za što veći broj platformi, onda je najbolje rešenje pisati višeplatformsku aplikaciju. Međutim, ako je cilj napraviti aplikaciju samo za *Android*, *iOS* ili bilo koju drugu mobilnu platformu, verovatno je bolja opcija pravljenje nativne aplikacije.

Ukoliko odluka bude pravljenje višeplatformske aplikacije, to je svakako dobra prilika za učenje novih tehnologija, kao što je i *KMM*. Upravo mogućnost pisanja višeplatfomskih aplikacija na *Kotlinu*, predstavlja jedan od ključnih benefita ovog programkog jezika. Za razvoj višeplatformskih aplikacija je dovoljan jedan tim programera, međutim, samo znanje *Kotlina* nije dovoljno. Potrebno je i poznavanje razvoja aplikacija za *iOS*. U modulu za *iOS* je potrebno implementirati korisnički interfejs, a po potrebi i dodatne opcije koje su specifične za platformu.

Neke od prednosti korišćenja *KMM*-a za razvoj višeplatformskih aplikacija su:

1. Veliki deo kôda može biti zajednički za sve platforme, samim tim nema potrebe za ponavljanjem kôda koji se tiče logike same aplikacije. Takođe, ovo vodi i lakšem održavanju kôda.
2. Kako se korisnički interfejs i integracija sa *API*-jima specifičnim za platformu pišu posebno za svaku platformu, dobijaju se bolje performanse i korisničko iskustvo.

3. Brzina razvoja je veća, a naknadno se lako može obezbediti podrška dodatnim platformama ako bude potrebe.

Neki od nedostataka korišćenja *KMM*-a za razvoj višeplatformskih aplikacija su:

1. Iako veliki deo kôda može biti zajednički za platforme *Android* i *iOS*, i dalje je potrebno poznavanje obe ove platforme kako bi se razvijali delovi aplikacije koji su specifični za svaku platformu. Da bi programeri modula za *iOS* mogli da rade na zajedničkom ili modulu za *Android*, moraju poznavati *Kotlin*, kao i radno okruženje *AndroidStudio*. Takođe, kako bi programeri modula za *Android* mogli da rade na modulu za *iOS*, potrebno je da poznaju *Swift*, kao i radno okruženje *Xcode*.
2. Sama konfiguracija ovakvog projekta može biti komplikovana. Na primer, dešava se da *AndroidStudio* ne prepoznaje instalirani dodatak za integraciju *CocoaPods*. Pored toga, dešava se da radno okruženje predloži korišćenje nove verzije neke biblioteke, a ukoliko se to prihvati, može doći do niza grešaka koje nastaju zbog nekompatibilnosti nove verzije biblioteke i verzije *Koltina* ili *Gradle*-a.

Dalji razvoj aplikacije *MyLoyaltyApp* bi mogao uključiti dodatno izmeštanje slojeva u deljeni modul i razvoj aplikacije za *iOS*. Zatim, dodatne opcije kao što su potvrda registracije, promena lozinke, pregledi akcijskih kataloga apoteka, pregled ostvarenih kupovina i drugo. Takođe, nema razloga da se pored apoteka, u aplikaciju ne uključe marketi ili knjižare, na primer.

6. Reference

[1] Andrey Breslav, **“Kotlin 1.0 Released: Pragmatic Language for the JVM and Android”**, *blog.jetbrains.com*, https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/ (26.08.2022.)

[2] Maxim Shafirov, **“Kotlin on Android. Now official”**, *blog.jetbrains.com*, https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/ (26.08.2022.)

[3] JetBrains, **“Get started with Kotlin”**, *kotlinlang.org*, https://kotlinlang.org/docs/getting-started.html (26.08.2022.)

[4] JetBrains, **“Kotlin”**, *jetbrains.com*, https://www.jetbrains.com/lp/devecosystem-2021/kotlin/ (26.08.2022.)

[5] Google Developers, **“Apps built with Kotlin”**, *developer.android.com*, https://developer.android.com/kotlin (26.08.2022.)

[6] John Callaham, **“From Android Market to Google Play: a brief history of the Play Store”**, *androidauthority.com*, https://www.androidauthority.com/android-market-google-play-history-754989/ (26.08.2022.)

[7] Steven Winkelman, **“Appy birthday: A brief history of the App Store’s first 10 years”**, *digitaltrends.com*, https://www.digitaltrends.com/news/apple-app-store-turns-10/ (26.08.2022.)

[8] Laura Ceci, **“Number of apps available in leading app stores Q2 2022”**, *statista.com*, https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (26.08.2022.)

[9] Dmitry Jemerov, **“Hello World”**, *blog.jetbrains.com*, https://blog.jetbrains.com/kotlin/2011/07/hello-world-2/ (26.08.2022.)

[10] Amit Raja Naik, **“Ten Years Of Kotlin Programing Language”**, *analyticsindiamag.com*, https://analyticsindiamag.com/ten-years-of-kotlin-programming-language/ (26.08.2022.)

[11] Bill Wixted, Boisney Philippe, Stanley Bogode, **“Discover the History of Kotlin”**, *openclassrooms.com*, https://openclassrooms.com/en/courses/5774406-learn-kotlin/5930526-discover-the-history-of-kotlin (26.08.2022.)

[12] Google Developers, **“Android’s Kotlin-first approach”**, *developer.android.com*, https://developer.android.com/kotlin/first (26.08.2022.)

[13] StackOverflow, **“2022 Developer Survey”**, *survey.stackoverflow.co*, https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages (26.08.2022.)

[14] StackOverflow, **“Stack Overflow Trends”**, *insights.stackoverflow.com*, https://insights.stackoverflow.com/trends?tags=kotlin%2Cjava (26.08.2022.)

[15] JetBrains, **“Kotlin Evolution”**, *kotlinlang.org*, https://kotlinlang.org/docs/kotlin-evolution.html (26.08.2022.)

[16] Dmitry Jemerov, Svetlana Isakova, **“Kotlin in Action”**, *Manning*, 2017. ISBN 9781617293290, *livebok.manning.com*, https://livebook.manning.com/book/kotlin-in-action/chapter-1/83 (26.08.2022.)

[17] JetBrains, **“Unsigned integer types”**, *kotlinlang.org*, https://kotlinlang.org/docs/unsigned-integer-types.html (10.09.2022.)

[18] JetBrains, **“Numbers”**, *kotlinlang.org*, https://kotlinlang.org/docs/numbers.html (26.08.2022.)

[19] JetBrains, **“Booleans”**, *kotlinlang.org*, https://kotlinlang.org/docs/booleans.html (26.08.2022.)

[20] JetBrains, **“Characters”**, *kotlinlang.org*, https://kotlinlang.org/docs/characters.html (26.08.2022.)

[21] JetBrains, **“Conditions and loops”**, *kotlinlang.org*, https://kotlinlang.org/docs/control-flow.html (26.08.2022.)

[22] JetBrains, **“Classes”**, *kotlinlang.org*, https://kotlinlang.org/docs/classes.html (26.08.2022.)

[23] JetBrains, **“Inheritance”**, *kotlinlang.org*, https://kotlinlang.org/docs/inheritance.html (26.08.2022.)

[24] JetBrains, **“Visibility modifiers”**, *kotlinlang.org*, https://kotlinlang.org/docs/visibility-modifiers.html (26.08.2022.)

[25] JetBrains, **“Functions”**, *kotlinlang.org*, https://kotlinlang.org/docs/functions.html (26.08.2022.)

[26] JetBrains, **“Operator overloading”**, *kotlinlang.org*, https://kotlinlang.org/docs/operator-overloading.html (26.08.2022.)

[27] Varun Kumar, **“Kotlin vs Javascript – Learning by comparison”**, *varunon9.medium.com*, https://varunon9.medium.com/kotlin-vs-javascript-learning-by-comparison-3835f5d94c1f (10.09.2022.)

[28] Anshul Bansal, **“Java vs. Kotlin”**, *bealdung.com*, https://www.baeldung.com/kotlin/java-vs-kotlin (26.08.2022.)

[29] JetBrains, **“Kotlin Multiplatform”**, *kotlinlang.org*, https://kotlinlang.org/docs/multiplatform.html (26.08.2022.)

[30] JetBrains, **“Get started with Kotlin Multiplatform Mobile”**, *kotlinlang.org*, https://kotlinlang.org/docs/multiplatform-mobile-getting-started.html (26.08.2022.)

[31] JetBrains, **“Setup an environment”**, *kotlinlang.org*, https://kotlinlang.org/docs/multiplatform-mobile-setup.html (26.08.2022.)

[32] JetBrains, **“Understand mobile project structure”**, *kotlinlang.org*, https://kotlinlang.org/docs/multiplatform-mobile-understand-project-structure.html (26.08.2022.)

[33] Microsoft, **“The Model-Vew-ViewModel Pattern”**, *docs.microsoft.com*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm (26.08.2022.)

[34] MVVM Cross, **“Data Binding”**, *mvvmcross.com*, https://www.mvvmcross.com/documentation/fundamentals/data-binding (26.08.2022.)

[35] Jessica Wilkins, **“MVC Architecture - What is a Model View Controller Framework?”**, *frecodecamp.org*, https://www.freecodecamp.org/news/mvc-architecture-what-is-a-model-view-controller-framework/ (26.08.2022.)

[36] Mozilla, **“MVC”**, *developer.mozilla.org*, https://developer.mozilla.org/en-US/docs/Glossary/MVC (26.08.2022.)

[37] Matthew Martin, **“MVC vs MVVM: Key Differences with Examples”**, *guru99.com*, https://www.guru99.com/mvc-vs-mvvm.html (26.08.2022.)

[38] Rosa Tiara Galuh, **“MVC vs MVVM: Key Differences with Examples”**, *makeuseof.com*, https://www.makeuseof.com/mvc-mvp-mvvm-which-choose/ (26.08.2022.)

[39] Lars Gyrup Brink Nielsen, **“The history of Model-View-Presenter”**, *dev.to*, https://dev.to/this-is-learning/the-history-of-model-view-presenter-420h (26.08.2022.)

[40] Ankit Sinhal, **“MVC, MVP and MVVM Design Pattern”**, *medium.com*, https://medium.com/@ankit.sinhal/mvc-mvp-and-mvvm-design-pattern-6e169567bbad (26.08.2022.)

[41] Manh Phan, **“MVP architectural pattern”**, *ducmanhphan.github.io*, https://ducmanhphan.github.io/2019-08-05-MVP-architectural-pattern/ (26.08.2022.)

[42] Anupam Chugh, **“Android MVVM Design Pattern”**, *digitalocean.com*, https://www.digitalocean.com/community/tutorials/android-mvvm-design-pattern (26.08.2022.)

[43] Square, **“Retrofit – A type-safe HTTP client for Android and Java”** , *square.github.io*, https://square.github.io/retrofit/ (10.09.2022.)

[44] Google Developers, **“Build better apps faster with Jetpack Compose”**, *developers.android.com*, https://developer.android.com/jetpack/compose (10.09.2022.)

[45] Dagger, **“Hilt”**, *dagger.dev*, https://dagger.dev/hilt/ (10.09.2022.)

[46] CocoaPods, **“What is CocoaPods”**, *cocoapods.org*, https://cocoapods.org/ (10.09.2022.)

[47] JetBrains, **“Creating a client application”**, *ktor.io*, https://ktor.io/docs/getting-started-ktor-client.html (10.09.2022.)

[48] Kotzilla, **“Koin – a smart Kotlin injection library to keep you focused on your app, not on your tools”**, insert-*koin.io*, https://insert-koin.io/ (10.09.2022.)