Napredni koncepti razvoja mobilnih aplikacija u Kotlinu

Listeš, Daniela

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:881122

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Daniela Listeš

NAPREDNI KONCEPTI RAZVOJA MOBILNIH APLIKACIJA U KOTLINU

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Daniela Listeš

JMBAG: 0016118400

Studij: Informacijsko i programsko inženjerstvo

NAPREDNI KONCEPTI RAZVOJA MOBILNIH APLIKACIJA U KOTLINU

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2022.

Daniela Listeš

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovim radom će se sistematizirati i opisati napredni koncepti razvoja mobilnih aplikacija u programskom jeziku Kotlin. Nakon uvoda u Kotlin, rad će uključiti analizu najboljih praksi i pristupa u dizajnu arhitekture, strukture i ponašanja mobilnih aplikacija, primjenu servisno orijentiranih arhitektura i mikroservisa, kao i najboljih praksi u radu s relacijskim i NoSQL podacima. U praktičnom dijelu rada bit će prikazan dizajn i implementacija jednog mobilnog programskog proizvoda koji će uključivati odabrane koncepte obrađene u ovom radu.

Ključne riječi: Kotlin; Android; Mobilna apliakcija; Programski jezik; Android Studio; Sql; NoSQL; Arhitektura

Sadržaj

1. Uvod

Tržište mobilnih aplikacija kontinuirano raste iz godine u godinu, a posebno snažan rast zabilježen je tijekom globalne pandemije, kada su razna statistička istraživanja zabilježila porast korisnika mobilnih aplikacija. Osim pandemije, novi trendovi na tehnološkom tržištu, poput 5G mreža također stvaraju sve veće prilike na tržištu.

Iz perspektive tehnoloških kompanija fokus je svakako na izradi samih aplikacija što uključuje različite pristupe te raznovrsne tehnologije. Kako raste potražnja za mobilnim aplikacijama te broj samih korisnika, raste i konkurencija kod izrade. Najviše profitiraju oni koji na vrijeme uoče trendove te ih iskoriste, ali osim brzine jako bitan faktor je i kvaliteta proizvoda. Cilj je da aplikacija dizajnom prati najnovije trendove, odnosno najbolju praksu, da pruži ugodno i intuitivno korisničko iskustvo, ali da i backend aplikacije tu ne zaostaje. Ono sto je u backendu mobilnih aplikacija bitno su arhitektura, ali i odabir stoga tehnologija. U oba područja dostupno je više opcija, neke su bolje od drugih, a nekad odabir može ovisiti i o domeni aplikacije, veličini, roku izrade, iskustvu razvojnog tima i slično.

Ovaj rad će se najprije baviti jezikom Kotlin kroz prikaz samih funkcionalnosti. Provest će se usporedba nekih bitnih funkcionalnosti koje smo već vidjeli u jeziku Java, zatim će se predstaviti neki novi koncepti koje donosi Kotlin te isto tako istaknuti i obraditi odabrani napredniji koncepti istog jezika. Nakon toga će se baviti različitim arhitekturama i tehnologijama koje se mogu koristiti u izradi mobilnih aplikacija s fokusom na servisno orijentirane arhitekture kao i mikroservise. Sve će biti zaokruženo i bazama podataka, odnosno najboljom praksom upotrebne SQL kao i NoSQL tehnologija, a isto tako neki od istraženih koncepata će biti primijenjeni u praktičnom projektu koji je dio ovog rada.

Motivacija za ovaj rad i temu proizlazi iz entuzijazma za Android razvoj te želje da se prouči trenutna Kotlin tehnologija koja je službeni standard tržišta te nadogradi postignuto znanje Android razvoja na fakultetu..

2. Metode i tehnike rada

Ovaj diplomski rad se sastoji od teorijskog i praktinčnog dijela. Teorijski dio obuhvaća proučavanje tematike te tehnologije koje se dotiče rada što je rezultiralo tekstualnim dijelom rada. Praktični dio obuhvaća izradu pripadne aplikacije temeljem teorijskog istraživanja. Najprije je započela prva faza, a u određenom periodu se izvedba obje preklapala. Nakon što je tekstualni dio rada završen još je uloženo dodatno vrijeme u praktični rad.

Prilikom izvedbe obje faze korišteni su dostupni izvori, a to su stručne knjige, znanstveni radovi, internetski izvori te do sada sveobuhvatno znanje s fakulteta. Svaka referenca na korištene literature navedena je u tekstu kao i na samom popisu literature. Za vrijeme pisanja ovog rada služila sam se konzultiranjem s tvrtkom Five.

Kroz cijeli rad prakticirane su određene istraživačke metodologije. Najprije analiza literature koja je korištena prilikom pisanja svakog poglavlja, zatim komparativna analiza u poglavljima gdje je provedena usporedba Kotlin funkcionalnosti koje su već viđene u Javi, također je korištena i u poglavljima o Android arhitekturi te kod usporedbe tehnologija za baze podataka. Može se izdvojiti i metodologija apstrakcije koja je naprimjer prisutna kod opisivanja tehnologije za rad s bazama podataka, a zatim se nastavak poglavlja usmjerio na konkretizaciju, samim praktičnim primjerom upotrebe tehnologije. Osim toga, prilikom proučavanja obujma korištenja tehnologija za baze podataka koristila se i statistička metoda u radu.

Programski alati i apliakcije koje su korištene su: Android Studio (Kotlin aplikacija), InteliJ (Ktor mikroservis i programski primjeri koji prate rad), Postman (testiranje servisa), Figma (Mockup i Wireframe), Firebase, MongoDB te LaTeX editor.

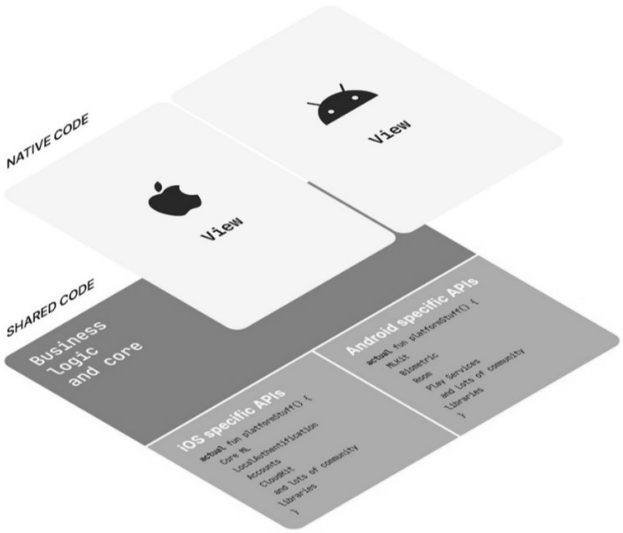
3. Kotlin

U ovom poglavlju će najprije biti ukratko opisana kratka povijest Kotlin jezika, a zatim će se detaljno predstaviti i opisati odabrane funkcionalnosti jezika. Najprije u usporedbi s jezikom Java, a zatim neke nove i napredne funkcionalnosti Kotlina.

Kotlin je besplatan jezik opće namjene, otvorenog koda te statički programski jezik koji je prvobitno dizajniran za JVM (Java Virtual Machine) i Android sustave. Nastao je u tvrtki JetBrains 2010. godine, a projekt otvorenog koda postao je 2012. godine. Godine 2017. Google je najavio Kotlin kao jedan od službeno podržanih jezika za Android razvoj. Do tada su to samo bili jezici Java i C++. Kotlin je tada ugrađen kao razvojni skup alata u Android Studio 3.0. Nakon te verzije, osim u novijim verzijama, Kotlin se može dodati i u starije verzije Android Studija s kasnije razvijenim dodacima [1].

Kombinira objektno orijentirane i funkcionalne programske značajke te je usredotočen na interoperabilnost, sigurnost te jasnoću. Kotlin se kompilira u isti bajtni kod kao Java te koristi Java virtualnu mašinu. U interakciji je s Java klasama na prirodan način i dijeli svoje alate s Javom. Sintaksa i način pisanja koda razlikuje od onoga u Javi, odnosno podudara se s funkcionalnim stilom programiranja. Sličnost Kotlina jezicima poput Scale i JavaScript donosi mogućnost bržeg programiranja te jednostavnijeg koda. Jedan od prepoznatljivih značajki funkcionalnog pisanja koda su lambda izrazi radi kojih je jednostavnije koristiti funkcije, obzirom da su ovako anonimne, odnosno bez identifikatora. Uobičajeno vrijeme za Java developera da nauči Kotlin je nekoliko sati, jezik omogućuje funkcije proširenja te dodavanje Kotlin rutina te smanjuje broj redaka koda za otprilike 40 posto [1].

Nova verzija Kotlina omogućuje nativni razvoj, odnosno kreirana aplikacija preko odgovarajućih apija može raditi na iOS, Android te web sustavima.



Slika 1: Vizualni prikaz nativnog razvoja, [1]

U nastavku će biti opisane neke odabrane funkcionalnosti Kotlina kroz tri potpoglavlja. Prvo potpoglavlje će se baviti unaprijeđenim funkcionalnostima Kotlina u odnosu na Javu, zatim nekim novim funkcionalnostima Kotlina koje je donio sam jezik te na kraju još nekoliko naprednih funkcionalnosti. Bitno je napomenuti da je u vrijeme pisanja ovog rada zadnja verzija Kotlina bila Kotlin 1.7.0.

3.1. Unaprijeđene funkcionalnosti u Kotlinu

Ovo potpoglavlje će se baviti unaprijeđenim funkcionalnostima kako sam naslov kaže. Dobro su nam poznate i do sada su se koristile u jeziku Java, no dolaskom Kotlina, izmijenjene su te postaju naprednije, kraće ili jednostavnije za korištenje. Najprije slijedi statički uvoz layout klasa koji će biti ukratko definiran a zatim će biti prikazan programski kod.

3.1.1. Statički uvoz layout klasa

Do sada je u razvoju Android aplikacija u Javi bila standardna upotreba metode *findViewById()* za dohvaćanje referenci `View` klasa u `Activity` ili `Fragment` klasama, odnosno *data binding*. U međuvremenu su razvijene neke biblioteke s kojima se moglo postići smanjenje takvog šablonskog koda, a najpoznatija je *Butterknife* biblioteka. Kotlin ovdje donosi inovativni pristup kojim omogućuje da se učitaju sve reference za View klase jednim uvozom. U nastavku je prikazan praktični primjer opisane funkcionalnosti[2].

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingBottom="@dimen/activity_vertical_margin"https://www
      .overleaf.com/project/62afddb1e84b1e537fdf4f24
   android:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   tools:context="co.dlistes.kotlinprimjeri.MainActivity">

   <TextView
      android:id="@+id/firstTextView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
</RelativeLayout>
```

Programski kod 1: Primjer statičkog uvoza layout klasa (XML)

```
package co.dlistes.kotlinprimjeri
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        firstTextView.text = "Hello World!"
    }
}
```

Programski kod 2: Primjer statičkog uvoza layout klasa (Kotlin)

U nastavku slijedi potpoglavlje gdje će biti prikazano kako se često korištene POJO klase, već poznate iz Java jezika, pušu u jeziku Kotlin. Također bit će prikaza usporedba programskog koda oba jezika.

3.1.2. Kreiranje POJO klasa

U Javi se koriste takozvane *POJO ("Plain Old Java Object")* klase za spremanje podataka. Najprepoznatljivija primjena ovakvih klasa su *REST* servisi gdje se *POJO* klase koriste kao dio odgovora i zahtjeva na servis. Kotlin značajno olakšava korištenje klase za istu svrhu, a razlika u količini koda prikazana je ispod u primjeru.

Java kod

```
public class Korisnik {
   private String ime;
   private String prezime;

   public String getIme() {
       return ime;
   }
   public void setIme(String ime) {
       this.ime = ime;
   }
   public String getPrezime
       return prezime
   }
   public void setPrezime(String prezime) {
       this.prezime = prezime;
```

```
    }
}
```

Programski kod 3: Primjer POJO klase (Java)

Kotlin kod

```
class Korisnik {
    var ime: String? = null
    var prezime: String? = null
}
```

Programski kod 4: Primjer POJO klase (Kotlin)

U prikazanim primjerima 3 i 4 jasno vidimo da u Kotlin POJO klasi nema get i set metoda, a razlog tomu je taj što ih Kotlin ima auto generirane te omogućuje izostavljanje autorovog pisanja takvih metoda kao što je to moguće jedino uz korištenje određenih biblioteka i u Javi. Nakon POJO klasa u nastavku slijedi potpoglavlje gdje će biti prikazani tipovi konstruktora u Kotlinu te kako se korite.

3.1.3. Konstruktor klasa

Kotlin klase imaju primarni konstruktor te jedan ili više sekundarnih konstruktora. U nastavku će biti prikazani primjeri istih te će biti kratko objašnjeni.

3.1.3.1. Primarni konstruktor

U ovom potpoglavlju najprije će biti opisani priparni konstruktori u Kotlinu. Ispod u programskom kodu broj 5 je prikazan klasični način pisanja konstruktora u Kotlinu, navođenjem ključne riječi *constructor* nakon naziva klase te navođenjem parametara konstruktora unutar zagrade, a nakon toga je prikazan i klasičan primjer pisanja konstruktora u Javi. Ključna razlika je u upotreni ključne riječi, ali i navođenju tipova parametara što je značajka pojedinog jezika[2].

```
//Kotlin
class Korisnik constructor(ime: String, prezime: String) {
}

//Java
public Korisnik(String ime, String prezime) {
}
```

Programski kod 5: Sintaksa primarnog konstruktora u Kotlinu i Javi

Drugi primjer odnosno programski kod bod probjem 6 prikazuje strukturu primarnog konstruktora koja se koristi kada ne postoje anotacije ili modifikacije vidljivosti. Također bitno

je spomenuti da je prikazan *init* blok koda. Naime, konstruktor u sebi ne može imati bilo kakav kod, sva inicijalizacija ili neki drugi kod koji želimo dodati se mora desiti upravo unutar tog bloka[2].

```
class Korisnik (ime: String) {
    init {
            //TO DO: inicijalizacija primarnog konstruktora
        }
}
```

Programski kod 6: Sintaksa primarnog konstruktora kada ne postoje anotacije i modifikacije vidljivosti

Bitno je napomenuti da se primarni konstruktor može koristiti i za definiranje te inicijaliziranje atributa klase, tako da se vrijednosti prosljeđuju putem parametara konstruktora, kao u jeziku Java. U nastavku je prikazana sintaksa takvog konstruktora. Možemo vidjeti da se za postavljanje vrijednosti koristi ključna riječ *"this"* te da Kotlin ima mehanizam za određivanje promjenjivosti parametara. Ako se ispred naziva parametra stavi ključna riječ *"val"* vrijednost je nepromjenjiva, a ako se stavi ključna riječ *"var"*, vrijednost je promjenjiva. Na primjeru broj 7 se vidi i logična primjena, recimo u slučaju da jedan korisnik promjeni prezime to bi bilo moguće ažurirati u sustavu[2].

```
class Korisnik(var ime: String, var prezime: String) {
  // ...
}
```

Programski kod 7: Sintaksa primarnog konstruktora uz var tip

Nakon primarnog konstruktora u nastavku slijedi potpoglavlje gdje će biti opisan sekundarni konstruktor, njegova primjena te programski kodovi kako bi se prikazala sintaksa.

3.1.3.2. Sekundarni konstruktor

Razlika u sintaksi sekundarnog i primarnog konstruktora jest pozicija pisanja. Dok je primarni konstruktor u zaglavlju klase, sekundarni konstruktori se navode nakon definiranja klase. U nastavku je prikazana sintaksa pisanja sekundarnog konstruktora u slučaju kada klasa nema definiran primarni konstruktor[2].

```
class Korinsik {
    constructor(ime: String)
}
```

Programski kod 8: Sintaksa sekundarnog konstruktora kada nema primarnog

Sljedeći primjer prikazuje sintaksu pisanja sekundarnog konstruktora u slučaju kada je definiran primarni konstruktor. Ovdje sekundarni konstruktor delegira primarnom putem ključne

riječi *"this"*. U ovom konkretnom primjeru vidimo kako sekundarni konstruktor delegira parametar ime primarnom konstruktoru, te uvodi novi parametar Odjel.

```
class Korinsik(var ime: String, var prezime) {
    constructor(ime: String, parent: Odjel) : this(ime) {
        parent.zaposelnici.add(this)
    }
}
```

Programski kod 9: Primjer delegiranja parametra primarnom konstruktoru

Prilikom proučavanja jezika Kotlin uočeno je nekoliko zanimljivih "ponašanja" jezika: kod primarnog konstruktora izvršava se uvijek prije koda iz sekundarnog konstruktora i kada kreiramo klasu bez konstruktora, Kotlin sam generira prazni primarni konstruktor. Prilikom automatskog generiranja konstruktora Kotlin dodaje modifikator *"public"*, a ako želimo postići da se klasa ponaša kao da nema konstruktor potrebno je modifikator*public* zamijeniti s modifikatorom *private*. U nastavku slijedi potpoglavlje koje će se baviti nasljeđivanjem i nadjačavanjem te usporediti navede funkcionalnosti s istima jeziku Java.

3.1.4. Nasljeđivanje i nadjačavanje

U Kotlinu sve klase nasljeđuju klasu *Any* te su time automatski zadano zatvorene poput *final* klasa u Javi. Kako bi se omogućilo daljnje proširivanje klase, klasu je potrebno deklarirati kao otvorenu ili apstraktnu tako što se prije naziva klase nadopiše ključna riječ *open*[2]. U primjeru programskog koda pod brojem 10 prikazana je sintaksa i upotreba ključne riječi *open*.

```
open class Korisnik(val ime, val prezime)
class Moderator(val ime, val prezime) : Korisnik(ime, prezime)
```

Programski kod 10: Upotreba ključne riječi *open*

Dakle, u Kotlinu su klasu zadano zatvorene te ih treba otvoriti, dok su u javi zadano otvorene te ih je potrebno zatvoriti ključnom riječi. U slučaju kada klasa nema primarni konstruktor, nadklasu inicijalizira u sekundarnim konstruktorima upotrebom ključne riječi "super" što je prikazano u primjeru broj 11 koji slijedi u nastavku rada.

```
open class Korisnik(val ime, val prezime)
class Moderator : Korisnik {
    constructor(ime:String) : super(ime)
}
```

Programski kod 11: Upotreba ključne riječi *super*

U Kotlinu, metoda koja se želi nadjačati mora biti specificirana ključnom riječju *open*, odnosno klasa koja nasljeđuje neku klasu treba nadjačati sve metode iz roditeljske klase koje su označene s *open*, a upravo to je prikazano u primjeru programskog koda 12 koji slijedi u nastavku.

```kotlin
//Kotlin
open class Korinsik(var ime: String) {
    open fun ispisi() {
        print("Ovo je korisnik.")
    }
}

open class Moderator : Korisnik {
    constructor(ime: String) : super(ime)
    override fun ispisi() {
        print("Ovo je moderator.")
    }
}

//Java
public class Korinsik(String ime) {
    void ispisi() {
        print("Ovo je korisnik.")
    }
}

public class Moderator extends Korisnik {
    void ispisi() {
        print("Ovo je moderator.")
    }
}
```

Programski kod 12: Primjer nadjačavanja Java i Kotlin

Ako usporedimo Kotlin nadjačavanje s Java nadjačavanjem možemo uočiti dvije osnovne razlike u sintaksi. Prvo metoda koju je potrebno nadjačati u Javi nije specificirana ključnom riječju *open*, a druga razlika je ta što se prilikom nadjačavanja u Kotlinu dodaje *override* oznaka ispred oznake za funkciju *fun* te naziva same funkcije, dok se u Javi to ne koristi. Također u Kotlinu za proširivanje klase koristi dvotočje, a u Javi ključna riječ *extends*. U nastavku slijedi potpoglavlje koje će se baviti sučeljima u jeziku Kotlin te na koji način se razlikuje od Java sučelja.

3.1.5. Kotlin sučelja

U Kotlinu se sučelja definiraju putem ključne riječi *interface*, a implementiraju se putem dvotočke. Metode sučelja su javne, a sučelje omogućuje i definiranje varijabli i nepromjenjivih svojstava. Pravila sučelja navode da se kod implementacije sučelja s nepromjenjivim svojstvima ista nadjačavaju, a ako se implementira više sučelja potrebno je kod poziva metode sljedećom sintaksom označiti iz kojeg sučelja pozivamo metodu: *super<ime sučelja>.metoda()*.

U nastavku primjerom programskog koda broj 13 je prikazana sintaksa Kotlin sučelja .

```
interface Moderator {
    val prezime: String
    fun ispisi() {
        print("Ovaj korisnik je moderator.")
    }
}

interface Admin {
    fun ispisi() {
        print("Ovaj korisnik je admin.")
    }
}

class Korinsik: Moderator, Admin {
    override val prezime: String
        get() = "Sunic"

    override fun ispisi()
        super<Admin>.ispisi();

}
```

Programski kod 13: Primjer sintakse sučelja

Dakle, u programskom kodu broj 13 vidimo upotrebu Kotlin sučelja, a ako usporedimo sintaksu s jezikom Java možemo zaključiti da je sve vrlo slično, osim što se sučelje implementira dvotočkom dok se u Javi koristi ključna riječ. *implements*. No ipak bitnija razlika koja se da zamijetiti jest implementirana metoda unutar sučelja, naime Kotlin omogućuje kod unutar metoda u sučelju te time zapravo implementacijom sučelja klasa nasljeđuje i njegovo ponašanje.

Nakon opisa Kotlin sučelja, u posljednjem potpoglavlju ovog djela opisat će se takozvani *companion* objekti te prikazati kako se koriste u Kotlinu umjesto java *static* objekata.

3.1.6. Companion objekti

Poznata nam je upotreba *static* ključne riječi za deklariranje statičkih članova klase u jeziku Java, no u Kotlinu to ne postoji. Kako bi se omogućilo pristupanje funkciji bez instanciranja klase u Kotlinu se koriste takozvani "*companion* objekti". Unutar tog objekta možemo definirati metode i svojstva te im pristupati preko naziva klase i to bez implicitnog navođenja imena objekta. *Companion* objekti još omogućuju definiranje konstanti u Kotlinu[2]. U nastavku rada slijedi primjer jednog takog objekta u programskom kodu broj 14.

```
class Korisnik {
    companion object {
        var ime: String = "Ana"
        val prezime: String = "Sunic"
        const val spol: String = "F"
        fun spavaj() {}
    }
}

Korisnik.spavaj()
```

Programski kod 14: Primjer *Companion* objekta

Nakon što su opisane odabrane funkcionalnosti Kotlina koje su već viđene u Javi te i uspoređene s Java jezikom, u nastavku slijedi sljedeće poglavlje koje će na sličan način obraditi nekoliko odabranih novih funkcionalnosti, odnosno funkcionalnosti koje je donio Kotlin.

3.2. Nove funkcionalnosti u Kotlinu

U ovom poglavlju bit će opisano nekoliko novih funkcionalnosti koje donosi Kotlin. U usporedbi s Javom, Kotlin je svojim dolaskom donio puno noviteta, a novijim verzijama jezika funkcionalnosti su još više obogaćene. Ovdje su odabrane one funkcionalnosti koje su u korištenoj literaturi okarakterizirane kao dio naprednog Kotlina, ali osim toga poglavlje se dotiče i funkcionalnosti kao što je *Null safety* koje su vrlo zanimljive za spomenuti, posebno iz gledišta Java developera. I ovdje je bitno napomenuti da je u vrijeme pisanja ovog rada zadnja verzija Kotlina bila Kotlin 1.7.0. Poglavlje najprije kreče s lambda izrazima u Kotlinu. Najprije rad započinje opis te primjerima korištenja lambda izraza koji prije pojave Kotlinu nisu bili dostupni u Android razvoju.

3.2.1. Lambda izrazi

Do sada su Lambda izrazi iz Jave 8 bili nedostupni u Android razvoju, a do Kotlina su se koristila neka zaobilazna rješenja poput *Retrolambda* biblioteka. Kotlin donosi Lambda izraze koji se mogu usporediti s anonimnim funkcijama u Javi. U nastavku je prikazan primjer koda u oba jezika[2].

Java

```
view.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(v.getContext(), "Kliknuto!").show();
    }
};
```

Programski kod 15: Anonimne funkcije u Javi

Kotlin

```
//Lambda sintaksa
view.setOnClickListener({ view -> toast("Kliknuto!") })
//Lambda sintaksa za slučaj da se ne koriste parametri
view.setOnClickListener({ toast("Kliknuto!") })
//Lambda sintaksa za slučaj kada je funkcija zadnji
//parametar pa se može izostaviti iz zagrada
view.setOnClickListener() { toast("Kliknuto!") }
//Lambda sintaksa za slučaj kada funkcija ima
//samo jedan parametar koji je funkcija
view.setOnClickListener { toast("Kliknuto!") }
```

Programski kod 16: Lambda izrazi u Kotlinu

Dakle u programskom kodu broj 16 vidimo kako se u Android razvoju, gdje su upravo prije pojave Kotlina nedostajali lambda izrazi, koriste anonimne funkcije u Javi za istu svrhu kao i lambda funkcije u Kotlinu. Ovime su omogućene neke prednosti poput lakšeg iteriranja kroz kolekcije. Nakon lambda izraza u nastavku slijedi opis i primjena ekstenzijskih funkcija u Kotlinu.

3.2.2. Ekstenzijske funkcije

Kotlin pruža mogućnost proširenja klase ili sučelja novom funkcionalnošću bez potrebe za nasljeđivanjem ili korištenjem obrazaca dizajna kao što je Decorator. Ovaj mehanizam proširenja omogućuje i recimo pisanje novih funkcija za klase ili sučelja iz biblioteka koje koristimo. Za dodavanje funkcije klasi potrebno je nakon ključne riječi *fun* te imena klase koju proširujemo, dodati točku a zatim napisati ime funkcije s pripadnim parametrima, ako je ima. U nastavku je prikazan primjer dodavanja funkcije, za brisanje prvog i zadnjeg znaka u stringu, klasi String[2].

```
fun String.obrisiPrviZadnjiZnak(): String =  this.substring(1, this.length - 1)

fun main(args: Array<String>) {
    val primjerString= "Ej Ana!"
    val rezultantniString = primjerString.obrisiPrviZadnjiZnak()
    println("Rezultat: $rezultantniString")
}
izlaz:
Rezultat: j Ana
```

Programski kod 17: Primjer dodavanja funkcije klasi

Nakon kratkog opisa ekstenzijskih funkcija te primjera programskog koda u nastavku slijedi potpoglavlje koje će se baviti *null safety* problematikom u Kotlinu.

3.2.3. *Null safety*

NullPointerException je česta pojava u Javi te je kodiranje često orijentirano tako da se pokriju svi mogući slučajevi pojave ove iznimke. Funkcionalnost *Null-safety* je integrirana u Kotlin i upravo nam omogućuje da programiramo bez brige o spomenutoj iznimci. No, ni u Kotlinu nismo sigurno u sto posto slučajeva. Prema službenoj dokumentaciji Kotlina postoje sljedeći slučajevi kada je moguća pojava spomenute iznimke: kod eksplicitnog poziva za "izbacivanje" iznimke *NullPointerException*, kod upotrebe vanjskog Java koda, kada se koristi operator "!!" te kada se svojstvu *lateinit* pristupa u konstruktoru prije nego što se inicijalizira[2].

Prema zadanim postavkama, sve varijable i svojstva u Kotlinu smatraju se "ne null" (ne mogu zadržati null vrijednost) ako nisu izričito deklarirane kao *nullable*. Ako želimo definirati varijablu koja može prihvatiti *null* vrijednost, nakon tipa varijable dodajemo upitnik te navodimo *null* vrijednost[2]. Upravo to je prikazuje programski kod 18 u nastavku rada.

```
val dob: Int? = null
```

Programski kod 18: Sintaksa za definiranje null vrijednosti

Kako bi izveli neke operacije s *null* varijablom koju smo sami definirali, potrebno je obaviti provjeru *null* vrijednosti kao u Java jeziku. Ukoliko to ne napravimo, kod će se kompilirati s greškom jer prevodilac izvodi *null* provjere[2]. Ispravan način kodiranja bi bio sljedeći:

```
val dob: Int? = null

if(dob != null) {
    dob.toString();
}
```

Programski kod 19: Primjer provjere null vrijednosti

Provjera se može obaviti i s jednostavnijom sintaksom te upotrebom takozvanog *Elvis* operatora što je prikazano primjerom u programskom kodu broj 20.

```
val dob: Int? = null
dob?.toString()

//Elvis operator
val dob Int? = null
val dobString = dob?.toString() ?: "Vrijednost varijable dob je null"
```

Programski kod 20: Primjer pojednostavljene provjere null vrijednosti

Nakon *null safety* problematike sljedeće potpoglavlje u radu se bavi funkcijama konteksta koje su vrlo koristan novitete jezika Kotlin u odnosu na Javu.

3.2.4. Funkcije konteksta

Prije svega, funkcije konteksta možemo definirati kao funkcije koje izvršavaju programski kod unutar konteksta objekta. U Kotlinu ih koristimo za lakše pristupanje objektima pod drugačijim kontekstom[2], a te funkcije su: *let*, *run*, *with*, *apply* i *also*. Za potrebe programskih primjera za svih 5 funkcija konteksta koristit će se klasa koja je prikazana u primjeru programskog koda broj 21 u nastavku.

```
class Korisnik() {

    var ime: String = "Ana"
    var prezime: String = "Sunic"
    var adresa: String = "Ruzicnjak 7"

    fun ispisi()  = print(
            "Ime korisnika: $ime\n " +
            "Prezime korisnika: $prezime\n " +
            "Adresa korisnika: $adresa"
            )
}
```

Programski kod 21: Klasa Korisnik

Najprije slijedi opis te programski primjer korištenja funkcije konteksta *let* te primjeri provjera *null* vrijednosti kod korištenja ove funkcije.

3.2.4.1. Funkcija *let*

Funkciju *let* koristimo kada želimo izvršiti neku operaciju nad objektom te vratiti pojedinačnu vrijednost ovisno o potrebi, a ne cijeli objekt kao kod funkcija *apply* i *also* što će biti pokazano malo kasnije u radu[2]. U nastavku slijedi primjer korištenja ove funkcije unutra programskog koda broj 22.

```
private fun izvediLetOperaciju() {
    val korisnik = Korisnik().let {
        return@let "Ime korisnika je ${it.ime}"
    }
    print(korisnik)
}

izlaz:
Ime korisnika je Ana
```

Programski kod 22: Sintaksa *let* funkcije

Također, funkcija omogućuje provjeru *null* vrijednosti, a na primjeru ispod u programskom kodu broj 23 je vidljivo koliko se to obavlja lakše u Kotlinu te koliko je značajno manje koda nego u Javi.

```
var ime: String? = "Ana"
private fun izvediLetOperaciju() {
    val ime = Korisnik().ime?.let {
        "Ime korisnika je $it"
    }
    print(ime)
}
```

Programski kod 23: Sintaksa provjere *null* vrijednosti

Nije potrebno pisati *return@let*, ali je preporučljivo radi čitljivosti koda. U Kotlinu, ako je posljednja naredba u *let* bloku napisana bez ključne riječi *return*, ono se prema zadanim postavkama tretira kao da je[2]. Prema tome možemo napisati kao što pokazuje sljedeći primjer:

```
private fun izvediLetOperaciju() {
    val korisnik = Korisnik().let {
        "Ime korisnika je ${it.ime}"
    }
    print(korisnik)
}
izlaz:
Ime korisnika je Ana
```

Programski kod 24: Sintaksa pojednostavljene provjere *null* vrijednosti

Moguće je ključnu riječ *it* koju koristimo za referenciranje na objekt zamijeniti s čitljivim lambda parametrom, što može biti vrlo korisno za neke slučajeve kao recimo kada imamo ugniježđene *let* blokove, te želimo pristupiti specifičnom objektu[2]. Primjer korištenja ove ključne riječi nalazi se uz programskom kodu broj 25.

```
private fun izvediLetOperaciju() {
    val korisnik = Korisnik().let { korisnikData ->
        korisnikData.ime = "Valentina"
    }
    print(korisnik)
}
```

Programski kod 25: Primjer korištenja ključne riječi *it*

Još jedna korisna primjena ove funkcije jest kada želimo obaviti operaciju nad dohvaćenim rezultatom kod poziva u lancu[2]. To se može prikazati putem jednostavnog primjera kao u programskom kodu broj 26, gdje želimo iz liste korisnika pronaći Anu, odnosno korisnika s 3 slova u imenu te zatim ime ispisati.

```
private fun gdjeJeAna() {
    val korisnici = mutableListOf("Ana","Anna","Valentina",
    "Daneila","Tena","Andrea")
    korisnici.map { it.length }.filter { it == 3 }.let {
        print(it)
    }
}
izlaz: Ana
```

Programski kod 26: Primjer poziva u lancu

Nakon funkcije *let*, slijedi opis te programski primjer korištenja funkcije konteksta *run* te primjeri provjera *null* vrijednosti kod korištenja ove funkcije.

3.2.4.2. Funkcija *run*

Funkciju *run*, slično kao i *let* koristimo kada želimo izvršiti operaciju nad nekim objektom te vratiti neku povratnu vrijednost koja nije objekt. Ovdje koristimo ključni riječ *this* te funkciju možemo koristiti za inicijalizaciju objekta[2]. U nastavku u programskom kodu pod brojem 27 prikazano je kako se ova funkcija koristi.

```
private fun izvediRunOperaciju() {
    Korinsik().run {
        name = "Valentina"
        prezime = "Mindoljevic"
        adresa = "Splitska Ulica 7"
        return@run ispisi()
    }
}
izlaz:
Ime korisnika: Valentina
Prezime korisnika: Mindoljevic
Adresa korisnika: Splitska Ulica 7
```

Programski kod 27: Primjer korištenja funkcije *run*

Nakon funkcije *run*, slijedi opis te programski primjer korištenja funkcije konteksta *with* te primjeri provjera *null* vrijednosti kod korištenja ove funkcije.

3.2.4.3. Funkcija *with*

Funkcija *with* je sintaksom kao i razlogom korištenja gotovo ista funkciji *run*. Ovdje se postavlja pitanje u kojem slučaju odnosno kada odabrati jednu ili drugu. Obavljenim istraživanjem ustanovljen je slučaj gdje je jasna prednost upotrebe jedne funkcije na drugom, a to je kod provjere *null* vrijednosti[2]. Sljedeći primjer će to prikazati.

```
private fun provjeriNullVrijednost() {
    val korisnik: Korisnik? = null
    with(korisnik) {
        this?.ime = "Ana"
        this?.prezime = "Sunic"
        this?.adresa = "Ruzicnjak 7"
        this?.ispisi()
    }
}
```

Programski kod 28: Provjera *null* vrijednosti korištenjem funkcije *with*

Nakon funkcije *with*, slijedi opis te programski primjer korištenja funkcije konteksta *apply* te primjeri provjera *null* vrijednosti kod korištenja ove funkcije.

```
private fun provjeriNullVrijednost() {
    val korisnik: Korinsik? = null
    korisnik?.run {
        ime = "Ana"
        prezime = "Sunic"
        adresa = "Ruzicnjak 7"
        ispisi()
    }
}
```

Programski kod 29: Provjera *null* vrijednosti korištenjem funkcije *run*

3.2.4.4. Funkcija *apply*

Ova funkcija vraća cijeli objekt nad kojim se poziva te se koristi u svrhu promjena nad objektom. Jako koristan slučaj upotrebe u Android razvoju je naprimjer izvršavanje promjena nad UI elementom ekrana gdje u slučaju dugačkih naziva elementa dobivamo puno kraći kod, ali i jednostavniji za napisati s obzirom na to da bi se izmjene vršile preko ključne riječi *this*, a ne naziva elementa. Pruža provjeru *null* vrijednosti te ne prihvaća izraz *return*, odnosno uvijek vraća objekt na koji se odnosi[2]. U nastavku slijedi programski primjer korištenja ove funkcije.

```
private fun izvediApplyOperaciju() {
    val korisnik: Korisnik? = null
    korisnik?.apply {
        ime = "Valentina"
        prezime = "Mindoljevic"
        adresa = "Splitska Ulica 7"
        ispisi()
    }
```

```
}
izlaz:
Ime korisnika: Valentina
Prezime korisnika: Mindoljevic
Adresa korisnika: Splitska Ulica 7
```

Programski kod 30: Primjer korištenja funkcije *apply*

Nakon funkcije *apply*, slijedi opis te programski primjer korištenja funkcije konteksta *also* te primjeri provjera *null* vrijednosti kod korištenja ove funkcije.

3.2.4.5. Funkcija *also*

Funkcija *also* se poput funkcije *apply* koristi kada želimo napraviti promjene na nekom objektu. Pruža provjeru *null* vrijednosti te ne prihvaća izraz *return*, odnosno uvijek vraća objekt na koji se odnosi[2]. U nastavlu sliejdi programski primejr korištenja ove funkcije.

```
private fun izvediAlsoOperaciju() {
    val ime = Korisnik().also {
        print("Korisnicko ime je ${it.ime}\n")
        it.ime = "Anna"
    }.run {
        "Novo korisnicko ime je $ime\n"
    }
    print(ime)
}
izlaz:
Korisnicko ime je Ana
Novo korisnicko ime je Anna
```

Programski kod 31: Primjer korištenja funkcije *also*

Nakon opisanih funkcija konteksta slijedi poglavlje koje će se baviti delegatima u Kotlinu. Bit će opisano delegiranje svojstava klase te zasebno dva tipa delegata u Kotlinu, a to su delegat *Observable* i delegat *Vetoable*.

3.2.5. Delegati

Kotlin podržava obrazac dizajna "delegiranja" uvođenjem ključne riječi *by*. Koristeći ovu metodologiju, Kotlin omogućuje izvedenoj klasi pristup svim implementiranim javnim metodama sučelja kroz određeni objekt. Sljedeći primjer pokazuje sučelje i njegovu implementaciju, te zatim kako iz treće klase koristimo implementaciju metode korištenjem ključne riječi *by*[2].

```
interface Korisnik {
   fun ispisi()
```

```
}
class KorisnikImpl(val ime: String) : Base {
   //implementacija metode sucelja
   override fun ispisi() {
       println(ime)
    }
}
class Derived(korisnik: Korisnik) : Korisnik by korisnik
    // delegiranje metode

fun main(args: Array<String>) {
   val korisnik = KorisnikImpl("Ana")
   Derived(korinsik).ispisi()
}

izlaz: Ana
```

Programski kod 32: Primjer delegiranja u Kotlinu *also*

Nakon prikaza sintakse delegiranja i njegovog osnovnog korištenja u nastavku slijedi prikaz delegiranja svojstava klase.

3.2.5.1. Delegiranje svojstava klase

U prethodnom poglavlju je objašnjena upotreba obrazac dizajna delegiranja, a u ovom poglavlju će biti opisane 3 standardne metode delegiranja svojstava iz Kotlin biblioteke, a to su: *Lazy*, *Observable* i *Vetoable*. Najprije u nastavku slijedi opis *Lazy* metode delegiranja.

Lazy

Lazy je lambda metoda koja uzima svojstvo kao ulaz i zauzvrat daje instancu *Lazy<T>*, gdje je *<T>* u osnovi tip svojstava koje koristi. Sljedeći primjer pod brojem 33 pokazuje kako funkcionira[2].

```
class KorisnikLazy {
    init {
        println("Lazy korisnik je inicijaliziran.")
    }
}

class Korisnik {
    val korisnikLazy by lazy { KorisnikLazy() }
}
```

```
fun main() {
    val korisnikLazy = Korisnik()
}

izlaz:
```

Programski kod 33: Primjer delegiranja svojstava u Kotlinu

U primjeru iznad vidimo da je rezultat prazan, a to se desilo zato što objektu KorisnikLazy nismo pristupili, a instancirali smo ga *lazy* operatorom kojim postižemo izvršavanje *init* koda tek nakon pristupa instanci. U nastavku je ažuriran kod metode *main()* tako da smo pristupili instanci objekta, a u rezultatu je vidljivo da se izvršila metoda *inti* klase KorisnikLazy.

```
fun main() {
    val korisnik = Korisnik()
    korisnik.korisnikLazy
    korisnik.korisnikLazy
}

izlaz:
Lazy korisnik je inicijaliziran.
```

Programski kod 34: Primjer delegiranja s pristupanjem instanci objekta

Zanimljivo je napomenuti da se instanciranje objekta obavlja samo jedanput, čak i ako se linija koda gdje pristupamo objektu pozove više puta. To vidimo u gornjem primjeru gdje je namjerno treća linija koda napisana dva puta.

Lijena inicijalizacija je nešto što je od ranije poznato u računalnoj znanosti, a Kotlin omogućuje jednostavan način korištenja tog mehanizma. Mehanizam lijene inicijalizacije je vrlo koristan i često korišten u stvarnom svijetu razvoja aplikacija, a razlog je taj što omoguće uštedu memorije, samim time što odgađamo proces inicijalizacije nečega. Ako se radi o nekakvim velikim količinama podataka koji nisu potrebni u svakom trenutku rada aplikacije radi se o značajnim uštedama. Jedan takav primjer je inicijalizacija baze podataka, koja je odvija uglavnom prilikom pokretanja neke aplikacije. Ovim mehanizmom ju možemo odgoditi za konkretniji trenutak u kojem nam treba pristup podacima iz baze te time dobijemo brže pokretanje same aplikacije[2]. U nastavku slijedi opis i primjer *observable* delegata.

Observable

Observable delegat nam omogućuje da otkrijemo promjene na nekom objektu kojeg promatramo. Ovo je također lambda funkcija ili izraz koja se izvršava kod svakog poziva te daje staru i novu vrijednost[2]. U nastavku slijedi programski kod broj 35 koji pokazuje kako se koristi ovaj delegat.

```
class Korisnik {
    val korisnikLazy by lazy { KorisnikLazy() }
```

```
    var place: Int by Delegates.observable(70000) {property,
            oldValue, newValue ->
        println("Stara vrijednost $oldValue")
        println("Nova vrijednost $newValue")
    }
}

fun main() {
    val korisnikLazy = Korisnik()
    korisnikLazy.ocjene = 100000
    korisnikLazy.ocjene = 120000
}

izlaz:
Stara vrijednost 70000
Nova vrijednost 100000
Stara vrijednost 100000
Nova vrijednost 120000
```

Programski kod 35: Primjer korištenja *Observable* delegata

Primjer gore je nadogradnja na prethodni primjer, dodana je nova varijabla "place" s inicijalnom vrijednošću od 100000. Zatim je dodana je funkcija delegata s lambda izrazom i pripadnim parametrima. Možemo reći da je *observable* funkcija zapravo već poznata *OnCreate()* funkcija. U rezultatu vidimo da je ispisana stara i nova vrijednost plaće kod svake promjene vrijednosti plaće. Nakon *observable* delegata slijedi opis i programski primjer upotrebe *vetoable* delegata.

Vetoable

Vetoable delegat nam također omogućuje da otkrijemo promjene na nekom objektu kojeg promatramo poput prethodnog delegata, no ovim delegatom možemo postaviti uvijte pod kojim se promijenjena vrijednost sprema. Sljedeći primjer također je nadogradnja prethodnog, a dodan je *vetoable* delegat koji sprema vrijednost radnog staža korisnika isključivo ako je novo uneseni staž veći ili jednak broju 13[2].

```
class Korisnik {
    val korisnikLazy by lazy { KorisnikLazy() }

    var place: Int by Delegates.observable(70000) {property, oldValue, newValue
        println("Stara vrijednost $oldValue")
        println("Nova vrijednost $newValue")
    }
```

```
    var godineStaza: Int by Delegates.vetoable(10) {property, oldValue, newValue
        println("Nova vrijednost staza $oldValue")
        println("Stara vrijednost staza $newValue")
        newValue >= 13
    }
}

fun main() {
    val korisnikLazy = Korisnik()
    korisnikLazy.godineStaza = 12
    korisnikLazy.godineStaza = 13
    korisnikLazy.godineStaza = 14
}

izlaz:
Nova vrijednost staza 12
Stara vrijednost staza 10
Nova vrijednost staza 13
Stara vrijednost nova 10
Nova vrijednost staza 14
Stara vrijednost nova 13
```

Programski kod 36: Primjer korištenja *Vetoable* delegata

U rezultatu iz primjera programskog koda broj 36 vidimo da se stara vrijednost resetirala tek kada je odabrana godina staža bila jednaka broju 13.

Ovo poglavlje se bavilo nekim odabranim funkcionalnostima Kotlina koje do sada nisu bile dostupne u Android razvoju odnosnu u Javi za Android, dok su u prethodnom poglavlju viđene funkcionalnosti koje se koriste u Javi i Android razvoju s Javom no Kotlin ih je na neki način unaprijedio, izmijenio ili pojednostavio. Odabrane funkcionalnosti prema korištenoj literaturi spadaju pod napredne koncepte Kotlina ili su pak korištene u praktičnom radu pa su time odabrani kako bi bili dio ovog poglavlja. U nastavku slijedi još jedno potpoglavlje trećeg poglavlja, a koje će se baviti naprednim funkcionalnostima Kotlina. Neke su izabrane zbog značaja, a neke jer su korištene u praktičnom radu.

3.3. Napredne funkcionalnosti u Kotlinu

Ovo poglavlje rada bavit će se još nešto naprednijim konceptima, alatima i tehnikama koje donosi jezik Kotlin. U odnosu na prethodna dva potpoglavlja ovi koncepti zahtijevaju prethodno znanje o osnovama jezika te je potrebno više vremena da se svladaju od jednostavnih koncepata. Bit će najprije opisane Kotlin korutine koje omogućuju upravljanje radom s više dretvi, biblioteka *Koin* koja omogućuje implementiranje *dependency injection* tehnike, zatim

Live Data kao zbirka biblioteka koja omogućuje razvoj robusnih aplikacija i na kraju Kotlin *Flow* odnosno funkcionalnost kotlina za upravljanje više tokova podataka.

3.3.1. Kotlin korutine

Korutina (engl. *coroutin*) je uzorak dizajna paralelnosti dodan u Kotlin verziju 1.3 kao API. Koristi se u razvoju Android aplikacija kako bi se pojednostavio kod koji se izvršava asinkrono, a pomažu u upravljanju dugotrajnim zadacima koji bi inače mogli blokirati glavnu dretvu i uzrokovati situacije u kojima aplikacija ne reagira. Prema informaciji sa službene Android stranice, Više od 50% profesionalnih programera koji koriste Kotlin korutine prijavilo je povećanje produktivnosti u radu. Pomoću Kotlin korutina dakle možemo pisati asinkroni programski kod tradicionalno pisan pomoću *Callback* obrasca, ali kako bi ih koristili, najprije je potrebno objasniti kako ovaj mehanizam funkcionira[1].

Kako bismo lakše razumjeli Kotlin korutine, možemo reći da su korutine poput dretvi, ali bolje. Prvi razlog je taj što nam korutine omogućuju sekvencijalno pisanje asinkronog koda, a drugi je taj da su učinkovitije jer se nekoliko korutina mogu pokrenuti pomoću iste dretve. Dakle, iako je broj dretvi koje se mogu pokrenuti u aplikaciji prilično ograničen, granica za korutine je gotovo beskonačna[1].

Obustavljanje funkcija

Kotlin korutine temelje se na ideji obustave funkcija. To su funkcije koje mogu zaustaviti izvršavanje korutine u bilo kojem trenutku i zatim vratiti kontrolu korutini nakon što je rezultat spreman. Dakle, korutinama ostvarujemo da obustavljanje funkcija ne blokira trenutnu dretvu. To znači da bi kod mogao prestati s izvršavanjem u trenutku kada pozove funkciju obustavljanja te nastaviti kasnije, no ne znamo što će trenutna dretva raditi u međuvremenu. Mogla bi se vratiti na izvršavanje druge korutine u tom trenutku, a kasnije bi mogla nastaviti s izvršavanjem korutine koju smo napustili. Sve je to kontrolirano načinom na koji se obustavna funkcija poziv, a asinkrone su samo ako se koriste na asinkron način. Kako bi detaljnije definirali korutine možemo reći da jedna korutina nije točno jedan "zadatak", već niz "podzadataka" koje treba izvršiti određenim i zajamčenim redoslijedom. Pozivanje obustavnih funkcija uzrokuje podjelu zadatka na pod zadatke unutar korutine. Čak i ako se čini da je kod u jednom sekvencijalnom bloku, svaki poziv obustavne funkcije ograničava početak novog "podzadatka" unutar korutine[1].

Obustavne funkcije nemaju posebni tip povratnog podatka, deklariraju se kao i obične funkcije. Same po sebi nisu asinkrone te unutar njih možemo sekvencijalno pozivati druge funkcije na uobičajen način pozivanja funkcija. Moramo pričekati da se pozvana funkcija izvrši, prije nego što dobijemo povratnu vrijednost i izvršimo ostatak koda. Upravo ovo omogućuje da se kompleksi asinkroni kod u Kotlinu piše na jednostavniji način. Sljedeći kod prikazuje primjer pozivanja jedne obustavne funkcije unutar druge[1].

```
suspend fun preuzmiPodatke(): Podatak {
    // neka operacija koja preuzima podatke s mreze i koriste obustavni mehaniza
```

```
}
suspend fun josJednaObustavnaFunkcija(): Unit {
    delay(1000L)
    println("Prosla je jedna sekunda.")
    val podatak: Podatak = preuzmiPodatke()
    println("Pristigli podatak s mreze: $something")
}
```

Programski kod 37: Primjer obustave funkcije

Izravno pozivanje obustavne funkcije iz "normalne" funkcije ne može se kompilirati. Uobičajeno objašnjenje je "zato što se samo korutine mogu obustaviti", i iz toga zaključujemo da moramo nekako stvoriti korutinu iz koje ćemo pokretati funkciju obustave, a korutine se stvaraju korištenjem funkcija koje se nazivaju graditelji korutina (*engl. coroutine builders*)[1].

U nastavku će biti opisani graditelji korutina te će biti priloženi programski kodovi koji prikazuju način korištenja. Prikazat će se *runBlocking*, *launch* te *async* graditelji.

Najprije slijedi opis i primjer korištenja *runBlocking* graditelja. Ovom funkcijom na jednostavan način možemo obustavnu funkciju pozvati iz klasične funkcije tako što blokiramo trenutnu dretvu. Ovo se često koristi iz funkcije main(), a u nastavku slijedi vrlo jednostavan primjer[3].

```
fun main() {
    println("Hallo,")
    runBlocking {
        // ulazak u korutinu
        delay(2000L) // suspenzija korutine na dvije sekunde
    }
    // ostatak ispisa koji se izvrsava nakon dvije sekunde
    println("Ana!")
}
```

Programski kod 38: Primjer korištenja *runBlocking* funkcije

Nakon *runBlocking* graditelja slijedi *launch* graditelj. Rijetko kada postoji potreba za blokiranjem dretve, puno češća je potreba za pokretanjem asinkronog zadatka, a upravo to nam omogućuje korutina *launch*. Zove se još i glavni graditelj, a i najjednostavniji je način za stvaranje korutina. Uvijek koristi klasu *GlobalScope*, te vraća objekt klase *Job* koja implementira *CoroutineContext*[3], a to je prikazano programskim kodom 39 u nastavku.

```
GlobalScope.launch(Dispatchers.Main) {
    ...
}
```

Programski kod 39: Primjer korištenja *launch* funkcije

Klasa *Job* ima korisne funkcije koje pomažu u radu s korutinama te postoji mogućnost da objekt te klase ima roditeljsku klasu istog tipa. Roditeljska klasa ima određenu kontrolu nad "svojom djecom", a vršu ju putem sljedećih funkcija *job.join* te *job.cancel*.

Funkcijom *join* možemo blokirati korutinu koja je povezana s *job* objektom, sve dok djeca poslovi nisu završeni. Sve obustavne funkcije koje se pozivaju unutar te korutine su zapravo vezane za isti *job*, stoga je moguće saznati kada su podređeni poslovi završeni te nastaviti s izvršavanjem[3]. Upotreba ove funkcije prikazana je u sljedećem primjeru.

```
val job = GlobalScope.launch(Dispatchers.Main) {
    odradiZadatakaKorutine()
    val res1 = obustavniZadatak1()
    val res2 = obustavniZadatak2()
    process(res1, res2)
}
job.join()
```

Programski kod 40: Primjer korištenja *job.join* funkcije

Nadalje slijedi opis funkcije *job.cancel*. Ova funkcija će otkazati sve poslove povezane s djecom klase *Job*. U nastavku je primjer jednak prethodnom, no u ovom slučaju ako se pozove *cancel()* dok se izvodi *obustavniZadatak1()*, u *res1* se neće spremiti povratna vrijednost funkcije, a metoda *obustavniZadatak2()* se nikad neće ni izvršiti[3].

```
val job = GlobalScope.launch(Dispatchers.Main) {
    odradiZadatakaKorutine()
    val res1 = obustavniZadatak1()
    val res2 = obustavniZadatak2()
    process(res1, res2)
}
job.cancel()
```

Programski kod 41: Primjer korištenja *job.cancel* funkcije

async

Nakon *launch* graditelja slijedi *async* graditelj. Ova funkcija omogućuje paralelno pokretanje nekoliko pozadinskih zadataka. Uvijek ju treba pozvati unutar druge korutine i vraća specijalizirani *job* koji se zove *Deferred* te koji ima specijalnu funkciju *await()*. Funkciju *await* koristimo kada trebamo rezultat. Ako rezultat nije spreman tada se korutina suspendira, a ako smo tada imali rezultat, on se vraća i korutina se nastavlja[3].

Slijedi primjer s tri funkcije u programskom kodu broj 42, druge dvije za izvođenje zahtijevaju rezultat prve, stoga se one mogu izvesti paralelno nakon rezultata prve funkcije. Ovo je moguće riješiti korištenjem *withContext*, no u tom slučaju gubimo vrijeme. Recimo da za izvođenje jedne funkcije trebamo 3 sekunde, ukupno bi potrošili devet sekundi.

```
GlobalScope.launch(Dispatchers.Main) {

    val korisnik = withContext(Dispatchers.IO) {
        korisnikService.login(korisnickoIme, lozinka)
    }
    val nadredjeni = withContext(Dispatchers.IO) {
        korisnikService.dohvatiNadredjenog(korisnik)
    }
    val odjel = withContext(Dispatchers.IO) {
        korisnikService.dohvatiOdjel(korisnik)
    }

    val zaposlenik = korisnik.copy(pripadnost = odjel + nadredjeni)
}
```

Programski kod 42: Primjer korištenja *withContext* funkcije

Ispod se nalazi kod kao u prethodnom primjeru no ovaj put koristimo funkciju *async* kojom postižemo da se druge dvije funkcije izvode paralelno. Ako uzmemo istu pretpostavku da za izvođenje svake funkcije trebamo 3 sekunde, ovaj kod bi se izveo unutar 6 sekundi[3].

```
GlobalScope.launch(Dispatchers.Main) {

    val korisnik = withContext(Dispatchers.IO) {
        korisnikService.login(korisnickoIme, lozinka)
    }
    val nadredjeni = async(Dispatchers.IO) {
        korisnikService.dohvatiNadredjenog(korisnik)
    }
    val odjel = async(Dispatchers.IO) {
        korisnikService.dohvatiOdjel(korisnik)
    }

    val zaposlenik = korisnik.copy(pripadnost.await() = odjel + nadredjeni.await
}
```

Programski kod 43: Primjer korištenja *async* funkcije

Nakon opisanih korutina te samih graditelj korutina slijedi potpoglavlje koje će se baviti drugim naprednim konceptom, a to je upravo biblioteka *Koin* koja olakšava implementaciju *dependency injection* tehnike u Kotlinu.

3.3.2. *Koin*

Naziv ovog poglavlja je *Koin*, a to je Kotlin biblioteka za implementiranje *dependency injection* tehnike odnosno za naknadno dodavanje ovisnosti. *Dependency injection* je tehnika

programiranja koja klasu čini neovisnom o svojim ovisnostima, a to se postiže odvajanjem uporabe objekta od njegove kreacije. Ovo je već dobro poznata tehnika, a sljedeći primjer prikazuje kako to izgleda u kodu[4].

```
class Odjel {
    lateinit var sef: Sef

    fun start() {
        sef.pokreniSe()
    }
}

fun main(args: Array) {
    val odjel = Odjel()
    odjel.sef = Sef()
    odjel.pokreniSe()
}
```

Programski kod 44: Primjer *dependency injection* korištenja

U nastavku slijedi primjer korištenja *Koin* biblioteke, najprije jednostavan primjer u Kotlinu u programskom kodu broj 45.

```
//Dodavanje klasa
class Controller(val service: BusinessService)
class BusinessService()

// Deklariranje modula
val myModule = module {
  singleOf(::Controller)
  singleOf(::BusinessService)
}

//Pokretanje Koin-a metodom startKoin()
fun main(vararg args: String) {
  // start Koin!
  startKoin {
    // deklariranje modula
    modules(myModule)
  }
}
```

Programski kod 45: Primjer korištenja *Koin* biblioteke

Nakon primjera korištenja biblioteke, slijedi prikaz korištenja u tipičnoj Android klasi *Activity* odnosno samnom Android razvoju kojim se bavi ovaj rad.

```
class MyActivity() : AppCompatActivity() {
  // dodavanje BusinessService u svojstvo service
  val service: BusinessService by inject()
}
```

Programski kod 46: Primjer korištenja *Koin* biblioteke u Android razvoju

U nastavku slijedi opis i programski primjer treće odabrane napredne funkcionalnosti a to je *Live Data*. Bit će prikazano postavljanje instance, postavljenje podataka te promatranje izmjena.

3.3.3. *Live Data*

LiveData je dio komponenti Android arhitekture koje su u osnovi zbirka biblioteka koje omogućuju razvoj robusnih aplikacija koje se lako testiraju. To je *observable* klasa koja je svjesna životnog ciklusa aplikacije, što znači da poštuje životni ciklus drugih komponenti aplikacije, kao što su aktivnosti, fragmenti ili servisi. Ova svjesnost životnog ciklusa omogućuje da se promatraju samo aktivne komponente životnog ciklusa. S obzirom na to da je *observable* klasa, može biti "promatrana" od strane drugih komponenti što bi značilo da u *activity/fragment* klasama postoje reference na *ViewModel*. Ovo je poznati mehanizam koji se koristio ranije u Java razvoju, a sada kada je Kotlin standard, pojavile su se bolje opcije. Stoga ovaj rad neće ulaziti duboko u temu *LiveData*, a u nastavku slijedi primjer[5].

Kreiranje *LiveData* instance

```
private val pocetnaVrijednost = "Bok Ana!"
private val _liveData = MutableLiveData(pocetnaVrijednost)
val liveData: LiveData<String> = _liveData
```

Programski kod 47: Primjer *LiveData* instance

Postavljanje podataka u *LiveData*

Klasa *MutableLiveData* javno izlaže dvije metode za postavljanje podataka, a to su *setValue* i *postValue*. Ako se pozivaju u glavnoj dretvi, tada će i *setValue* i *postValue* raditi na isti način, tj. ažurirat će vrijednost i obavijestiti promatrače, no ako se pozivaju u nekoj pozadinskoj dretvi, tada je potrebno koristiti *postValue*[5], a to je prikazanu u sljedećem primjeru.

```
fun updateLiveData() {
    _liveData.postValue("Kako si spavala?")
}
```

Programski kod 48: Primjer korištenja *postValue* metode

Promatranje izmjene podataka

Ako dođe do promjene podataka, ti će se podaci odraziti na sve promatrače koji su s njima povezani, ali o promjenama će biti obavješteni samo oni promatrači koji su u *live* ili u *active* stanju, a u nastavku slijedi prikaz promatranja izmjene podataka.

```
viewmodel.liveData.observe(this@MainActivity) { data ->
    liveDataT.text = data
}
```

Programski kod 49: Primjer promatranja izmjene podataka

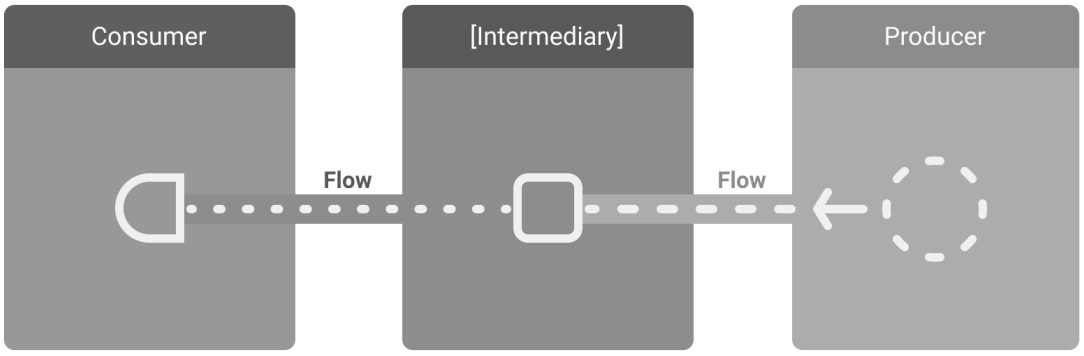
Neke bitne prednosti korištenja *LiveData* su: UI komponente odgovaraju stanju podataka, aplikacija se ne urušava radi zaustavljenih *activity-a*, pravilne izmjene konfiguracije te prikaz najnovijih podataka. Kao i u svemu, postoje i neki nedostaci, a oni značajniji su problemi s dretvama te činjenica ne radi s Kotlin *coroutines*, a i postojanje dosta *"Boiler Plate"* koda. Ovi nedostaci su nadvladani novim mehanizmom kojeg je donio Kotlin, a to je Kotlin *flow* kojim se bavi sljedeće poglavlje[5].

U nastavku slijedi opis i programski primjer četvrte odabrane napredne funkcionalnosti a to je Kotlin *Flow*. Bit će prikazano kreiranje *flow* objekata te korištenje *StateFlow* i *SharedFlow* funkcija.

3.3.4. *Kotlin Flow*

U korutinama, *flow* može emitirati više vrijednosti uzastopno, za razliku od suspendirajućih funkcija koje vraćaju samo jednu vrijednost. U nastavku rada će se koristi izraz "tok". Tokovi su izgrađeni na korutinama te mogu dati više vrijednosti, a to je zapravo konceptualno tok podataka koji može raditi asinkrono. Emitirane vrijednosti moraju biti iste vrste. Na primjer, *Flow* je tok koji emitira cjelobrojne vrijednosti. Može se reći da su tokovi slični iteratorima, ali razlika je što koriste funkcije obustave za asinkronu proizvodnju i potrošnju vrijednosti. To znači, na primjer, da tok može sigurno uputiti zahtjev bez blokiranja glavne dretve[1].

Slika 2 prikazuje tri entiteta koji su uključeni u tok podataka (promatra se s lijeva na desno).



Slika 2: *Entiteti toka podataka*, [1]

Prateći sliku 2, najprije je proizvođač, on proizvodi podatke koji se dodaju u tok. Zahvaljujući korutinama, tokovi također mogu proizvoditi podatke asinkrono. Drugi entitet je posrednik

koji modificira vrijednosti koje se emitiraju u toku podataka i ovaj entitet je opcionalan. Zadnji entitet je potrošač koji konzumira vrijednosti iz toka[1].

Iz gledišta Androida, repozitorij bi bio tipičan proizvođač koji proizvodi podatke za UI, a UI komponenta bi bila potrošač koji u konačnici prikazuje podatke. S druge strane, sloj korisničkog sučelja je proizvođač događaja korisničkog unosa, a drugi slojevi hijerarhije ih konzumiraju. Slojevi između proizvođača i potrošača u Androidu također obično djeluju kao posrednici koji modificiraju tok podataka kako bi ga prilagodili zahtjevima sljedećeg sloja[1].

U nastavku je prikazan primjer broj 50. gdje prva linija prikazuje kreiranje *Flow* objekta, a kako *flow* počinje emitirati podatke kada se pozove funkcije *collect()* na stream, to pokazuje druga linija koda iz primjera ispod.

```
val myIntFlow: Flow<Int> = flow { emit(1) }
myIntFlow.collect { intVrijednost -> /* intVrijednost = 1 */}
```

Programski kod 50: Primjer kreiranja *flow* objekta

U sljedećem potpoglavlju slijedi opis i prikaz korištenja *StateFlow* i *SharedFlow* funkcija. Bit će prikazano korištenje kreiranja instance, a zatim emitiranje te dohvaćanje podataka.

3.3.4.1. *StateFlow* i *SharedFlow*

StateFlow i *SharedFlow* su *Flow API-ji* koji omogućuju tokovima optimalno emitiranje ažuriranih stanja te emitiranje vrijednosti većem broju potrošača. Najprije slijedi opis *StateFlow* toka.

StateFlow je *observable* tok *state-holder* koji emitira trenutna i nova ažurirana stanja svojim kolektorima. Vrijednost trenutnog stanja također se može očitati kroz svojstvo *value*. Da bismo ažurirali stanje i poslali ga u tok, u primjeru je dodana nova vrijednost svojstvu *value* klase *MutableStateFlow*. *StateFlow* vraća samo vrijednost koja je ažurirana, odnosno uvijek daje najnoviju vrijednost. *StateFlow* je *read-safe* jer uvijek sadrži vrijednost, a i zbog toga zahtjeva početnu vrijednost[1]. U nastavku slijedi prikaz kreiranja *StateFlow* instance.

```
private val pocetnaVrijednost = "Bok Ana!"
private val _stateFlow = MutableStateFlow(pocetnaVrijednost)
val stateFlow = _stateFlow.asStateFlow();
```

Programski kod 51: Primjer kreiranja *StateFlow* instance

Nakon primjera u kojem se kreira instanca, programski kod 52, koji slijedi, prikazuje emitiranje podataka u *StateFlow* toku.

```
fun updateStateFlow() {
    viewModelScope.launch {
        _stateFlow.emit("Jesi li se naspavala?")
    }
}
```

Programski kod 52: Primjer emitiranja podataka u *StateFlow-u*

Nakon primjera u kojem je prikazano emitiranje podataka, slijedi programski kod 53 koji prikazuje dohvaćanje promjena podataka u *StateFlow* toku.

```
lifecycleScope.launchWhenStarted {
    viewmodel.stateFlow.collectLatest { data ->
        stateFlowT.text = data
    }
}
```

Programski kod 53: Primjer dohvaćanja promjena podataka

Nakon opisa *StateFlow* toka, slijedi *SharedFlow* tok koji je vrlo konfigurabilna generalizacija *StateFlow-a*. Glavna im je razlika što *StateFlow* uzima zadanu vrijednost kroz konstruktor i emitira je odmah kada ju netko počne prikupljati, dok *SharedFlow* ne uzima nikakvu vrijednost i ne emitira ništa prema zadanim postavkama[1]. U nastavku programski kod 54 prikazuje primjer kreiranja *SharedFlow* instance.

```
private val pocetnaVrijednost = "Bok Ana!"
private val _sharedFlow = MutableStateFlow(pocetnaVrijednost)
val sharedFlow = _sharedFlow.asSharedFlow();
```

Programski kod 54: Primjer kreiranja *SharedFlow* instance

Nakon primjera u kojem se kreira instanca, slijedi programski kod 55 koji prikazuje emitiranje podataka u *SharedFlow* toku.

```
fun updateSharedFlow() {
    viewModelScope.launch {
        _sharedFlow.emit("Jesi li se naspavala?")
    }
}
```

Nakon primjera emitiranja podataka, slijedi programski kod 56 koji prikazuje dohvaćanje podataka u *SharedFlow* toku.

```
lifecycleScope.launchWhenStarted {
    viewmodel.sharedFlow.collectLatest { data ->
        sharedFlowT.text = data
    }
}
```

Programski kod 56: Primjer dohvaćanja promjena podataka u *SharedFlow-u*

Ovime završava potpoglavlje naprednih koncepata Kotlina čime je pokriven jedan od ciljeva rada a to su napredni koncepti Kotlina. Kotlin korutine, *Live Data* koncept te *Koin* biblioteka su odabrani jer su korišteni u praktičnom radu, a Kotlin *Flow* radi zanimljivosti tematike.

Nakon ovog poglavlja slijedi novo veliko poglavlje koje će se baviti Android arhitekturom, najpoznatijim arhitekturalnim uzorcima dizajna u Android razvoju te servisno orijentiranim i mikroservisnim arhitekturama.

4. Arhitektura mobilnih aplikacija

Ovo poglavlje bavit će se arhitekturom mobilnih aplikacija. Najprije će se opisati uzorci dizajna arhitekture, a nakon toga tipovi arhitektura na strani poslužitelja. U nastavku će biti objašnjeni neki od uzoraka dizajna koji se koriste prilikom izrade mobilnih aplikacija, a određuju na koji način određeni slojevi aplikacije komuniciraju odnosno kako su raspoređene odgovornosti[6].

4.1. Uzorci dizajna arhitekture

Arhitektura mobilne aplikacije odnosi se na skup pravila, tehnika, procesa i obrazaca za razvoj mobilne aplikacije. Ova pravila pomažu razvojnim programerima da kreiraju aplikacije koje zadovoljavaju i poslovne zahtjeve i industrijske standarde. U nastavku će biti analizirana tri najpoznatija uzorka arhitekture, kako se oni prakticiraju u razvoju android apliakcija, te koji od ta tri uzorka bi bio najbolja praksa u razvoju aplikacija[6]. Najprije slijedi uzorak MCV.

4.1.1. *Model-view-controller (MVC)*

MVC odnosno *Model-View-Controller* je jedan od prvih uzoraka koji se bavi odgovornostima pojedinih aplikacijskih slojeva te je ujedno i jedan od najpoznatijih uzoraka dizajna arhitekture. Prvi ga je predstavio Trygve Reenskaug 1970-ih, gdje ga je upotrijebio u jeziku *SmallTalk-76*. Od tada se samo razvijao, pomažući programerima na različitim jezicima i platformama. Uglavnom se koristi za razvoj web i desktop aplikacij[6].

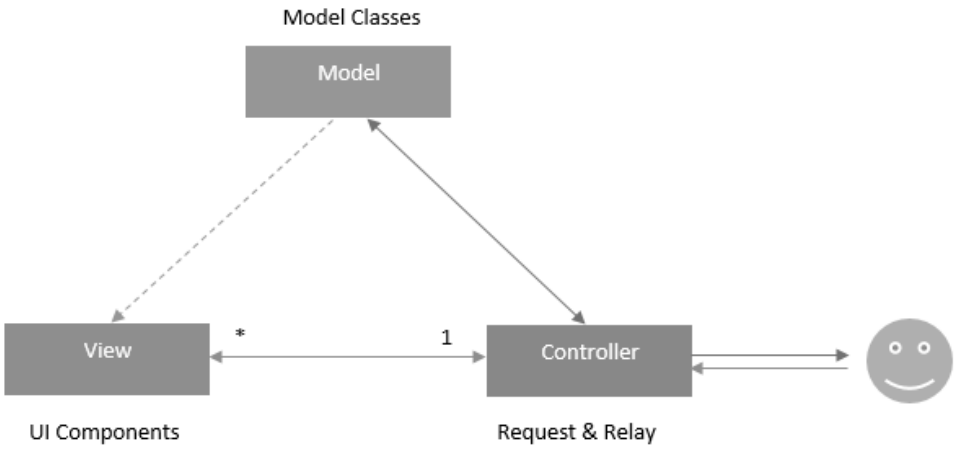
Ovim uzorkom klasificiramo odgovornosti aplikacije u tri neovisne komponente, a to su *model*, *view* i *controller*. U ovom uzorku dizajna arhitekture više *view* slojeva dijeli jedan *controller* te on upravlja a aktivnim *view* slojevima[6].

Model je ovdje podatkovni dio aplikacije te upravlja podacima, logikom te pravilima. Sadrži set klasa koje opisuju poslovnu logiku, poslovni model te sloj za pristup podacima. Također, definira i poslovna pravila koja određuju kako i pod kojim uvjetima podaci mogu biti promijenjeni i manipulirani. Odgovoran je za stanja, odnosno odgovara na zahtjeve o stanju koji obično dolaze od pogleda te je odgovoran za zahtjeve promjene stanja, koji uglavnom dolaze od kontrolera[6].

View je sloj korisničkog sučelja, to su uglavnom *UI* komponente, *CSS*, *JavaScript* te *HTML* kod. Služi za prikazivanje podataka koji dolaze iz sloja *controller*[6].

Controller sadrži logiku koja odgovara na akcije korisnika odnosno procesira dolazeće zahtjeve. Upravlja s inputima koji dolaze od strane korisnika putem sloja *view*, prilikom toga, procesira korisničke podatke najčešće uz pomoć *model-a*, a rezultate koje dobije od *view* šalje natrag na *view*. Možemo reći da se ponaša kao koordinator između sloja *view* i sloja *model* ili da zapravo informira *model* i *view* o korisničkim akcijama[6].

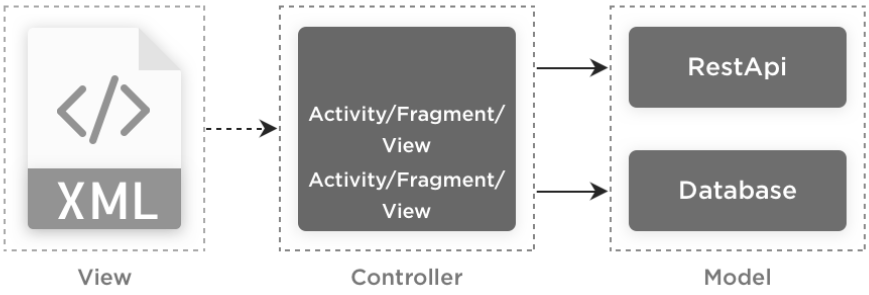
Na slici pod rednim brojem 3 vidimo kako funkcionira ta međusobna interakcija i komunikacija između tri sloja. Vidimo da su *controller* i korisnik u obostranoj interakciji, dakle on odgovara na akcije korisnika. *Controller* također komunicira sa slojem *model* tako da mu i šalje i prima podatke, no primljenim podacima samo manipulira te ih ne predaje u sloj *view*, odnosno nije pretplaćen na promjene *modela*[6].



Slika 3: *MVC arhitektura, [13]*

Možemo vidjeti da je veza između sloja *model* i *view* prikazana isprekidanom strjelicom što nam govori da su zapravo ta dva sloja u ovoj arhitekturi povezana i to s indirektnom vezom. Naime, u ovoj arhitekturi sloj *view* je pretplaćen na promjene *model-a*, tako da se u slučaju promjene automatski i sam mijenja bez potrebe za slanjem podataka nazad preko *controller-a*, odnosno *model* ne vraća podatke u sloj poslovne logike nego ga predaje sloju *view*[6].

Ako sagledamo ovaj uzorak dizajna iz perspektive razvoja android aplikacija to izgleda kao što je prikazano na slici pod brojem 4. Uobičajeni pristup u implementaciji *MVC-a* u Androidu je pisanje oko preko 1000 linija koda u jednoj aktivnosti ili fragmentu. U ovom pristupu, svaka klasa aktivnosti i fragmenata s pripadnim promatračima, slušačima, okidačima na događaje i slično su dio sloja *controller-a*. Sloj *model-a* bi bile klase koje sadrže pozive na bazu podataka, pozive na *REST API-e*, poslovna logika, model podataka i slično. I naravno *XML layout-i* bi bili dio sloja *view*[6].



Slika 4: *MVC android arhitektura, [12]*

Ovakva vrsta implementacije vrlo lako može postati podložna greškama te je vrlo teško

izvesti *UNIT* testove nad ovime, a i gubi se princip ponovne iskoristivosti koda. U praksi kada bi pokušali implementirati *Activity* i *Fragment* klase da se ponašaju kao *controller*, dobili bismo vrlo kompliciranu strukturu klasa te čestu pojavu redundantnog koda. Nadalje, ovaj uzorak nalaže da *model* ažurira *view* u pozadini. Kada bismo imali veliku aplikaciju s puno pogleda ponekad bi bilo kompliciranо pratiti koji pogled je ažuriran modelom. Prema tome možemo lako zaključiti da ovaj uzorak i nije najbolja praksa za razvoj android aplikacija, no bez obzira na to ima svoju primjenu i još uvijek se koristi[6]. U sljedećem potpoglavlju slijedi programski primjer ove arhitekture.

4.1.1.1. Primjer *MVC* arhitekture u Kotlinu

Najprije je prikazan kod za sučelje klase *View* s dvije pripadne metode te xml fajlica sa poljima s formom za prijavu korisnika.

***View* paket: IPrijavaView.kt (*Interface*), MainActivity.xml (*Class*)**

```
interface IPrijavaView {
    fun OnPrijavaSuccess(message: String?)
    fun OnPrijavaError(message: String?)
}
```

Programski kod 57: IPrijavaView.kt klasa

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android
    "
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"">

        <androidx.appcompat.widget.AppCompatEditText
            android:id="@+id/inputEmail"
            android:layout_width="match_parent"
            android:layout_height="55dp"
            android:inputType="text"/>

        <androidx.appcompat.widget.AppCompatEditText
            android:id="@+id/inputPassword"
            android:layout_width="match_parent"
```

```
            android:layout_height="55dp"
            android:inputType="textPassword"/>

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="50dp">

            <com.google.android.material.button.MaterialButton
                android:id="@+id/buttonPrijava"
                android:layout_width="match_parent"
                android:layout_height="match_parent"/>
        </RelativeLayout>

    </LinearLayout>
</ScrollView>
```

Programski kod 58: MainActivity.xml fajl

Nadalje slijedi sučelje za podatkovnu klasu Korisnik kao i sama klasa Korisnik. Korisnik sadrži email i lozinku da implementira metode sučelja za dohvat emaila, lozinke te provjeru ispravnosti korisnika.

***Model* paket: IKorisnik.kt (*Interface*), Korisnik.kt (*Class*)**

```
interface IKorisnik {
    fun getEmail(): String?
    fun getPassword(): String?
    fun isValid(): Int
}
```

Programski kod 59: IKorisnik.kt sučelje

```
class Korisnik(
    private val email: String?,
    private val password: String?
) : IKorisnik {

    override fun getEmail(): String? {
        return email
    }

    override fun getPassword(): String? {
        return password
    }
```

```
    override fun isValid(): Int {
        if(TextUtils.isEmpty(getEmail()))
            return  0
        else if(!Patterns.EMAIL_ADDRESS.matcher(getEmail()).matches())
            return  1
        else if(TextUtils.isEmpty(getPassword()))
            return 2
        else if(getPassword()?.length!! <=7)
            return 3
        else
            return -1;
    }
}
```

Programski kod 60: Korisnik.kt klasa

Kreirana je i *Controller* klasa sa svojim pripadnim sučeljem gdje je implementirana metoda za prijavu korisnika. Također sadrži i referencu na *View* klasu, a i vidljivo je da komunicira s *Model* slojem.

***Controller* paket: IPrijavaController.kt (*Interface*), PrijavaController.kt (*Class*)**

```
interface IPrijavaController {
    fun OnLogin(email: String?, password: String?)
}
```

Programski kod 61: IPrijavaController.kt sučelje

```
class PrijavaController(
     private val prijavaView: IPrijavaView
):IPrijavaController {

    override fun OnPrijava(email: String?, password: String?) {
        val user = User(email, password)
        val loginKod = user.isValid()

        when (loginKod) {
            0 -> {
                loginView.OnLoginError("Molimo unesite e-mail adresu!");
            }
            1 -> {
                loginView.OnLoginError("Molimo unesite ispravnu e-mail
                    adresu!");
            }
```

```
            2 -> {
                loginView.OnLoginError("Molimo unesite lozinku!");
            }
            3 -> {
                loginView.OnLoginError("Molimo unesite lozinku koja ima
                    više od 7 znakova!");
            }
            else -> {
                loginView.OnLoginSuccess("Uspješna prijava!");
            }
        }

    }
}
```

Programski kod 62: PrijavaController.kt klasa

Na kraju je još kreiran programski kod glavne klase gdje se vidi kako *View* sloj komunicira s *Controller* slojem.

MainActivity.kt (*Class*)

```
class MainActivity : AppCompatActivity(), IPrijavaView {

    var email: EditText? = null
    var password: EditText? = null
    var loginButton: Button? = null
    var loginPresenter: ILoginController? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        email = findViewById(R.id.inputEmail)
        password = findViewById(R.id.inputPassword)
        loginButton = findViewById(R.id.buttonPrijava)
        loginPresenter = LoginController(this)

        loginButton?.setOnClickListener {
            (loginPresenter as LoginController).OnPrijava(
                email?.text.toString(),
                password?.text.toString().trim()
            )
        }
```

```
    }

    override fun OnPrijavaSuccess(message: String?) {
        Toast.makeText(this,message, Toast.LENGTH_LONG).show()
    }

    override fun OnPrijavaError(message: String?) {
        Toast.makeText(this,message,Toast.LENGTH_LONG).show()
    }
}
```

Programski kod 63: MainActivity.kt klasa

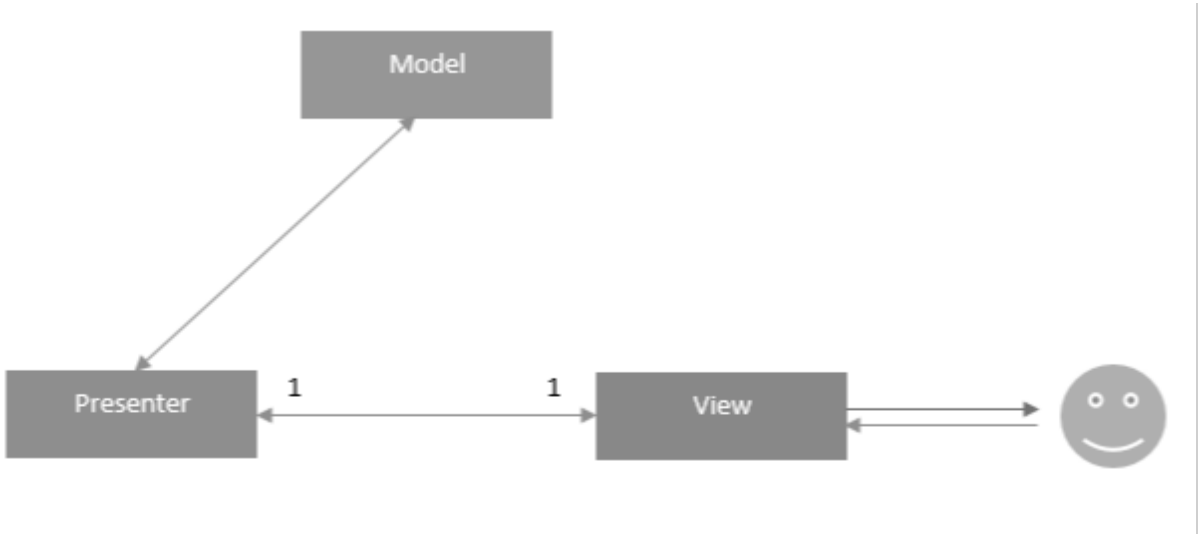
Nakon programsko primjera korištenja *MVC* uzorka arhitekture, slijedi dio rada gdje će na isti način biti prikazana *MVP* arhitektura, odnosno najprije teorijski, a zatim također s programskim primjerom.

4.1.2. *MVP*

MVP obrazac sličan je MVC uzorku, no ovdje je kontroler zamijenjen presenterom, a veze funkcioniraju malo drugačije. Osim toga, u MVC uzorku dizajna više view slojeva djele jedan controller, dok ovdje svaki view sloj ima svoj presenter sloj. Dakle ovdje se aplikacija dijeli na tri sloja, a s to su model, pogled i prezenter. Model i ovdje predstavlja skup klasa koje opisuju poslovnu logiku i podatke te definira poslovna pravila koja govore o tome kako se podaci mijenjaju i manipuliraju. Pogled je također gotovo isti, predstavlja komponente korisničkog sučelja, kao i CSS, ks i HTML kod. Isto kao i kod MVC obrasca prikazuje rezultate i podatke korisniku odnosno transformira modele u UI[6].

Prezenter je ovdje odgovoran za upravljanje događajima na korisničkom sučelju, prima input od korisnika putem pogleda, procesira korisničke podatke u suradnji s modelom te rezultate prosljeđuje natrag na pogled. Razlika MVP i MVC uzorka leži u tome što su ovdje prezenter i pogled potpuno odvojeni jedan od drugoga te međusobno komuniciraju putem sučelja. Uz to, prezenter ne upravlja dolaznim zahtjevima kao kontroler[6].

Na slici broj 5 vidimo da je korisnik u interakciji s pogledom, vidimo jedan prema jedan vezu pogleda i prezentera što znači da je svaki pogled vezan samo za jedan prezenter. Nadalje, pogled ima referencu na prezenter, ali nema referencu na model. Također, omogućena je dvosmjerna komunikacija između pogleda i prezentera[6].



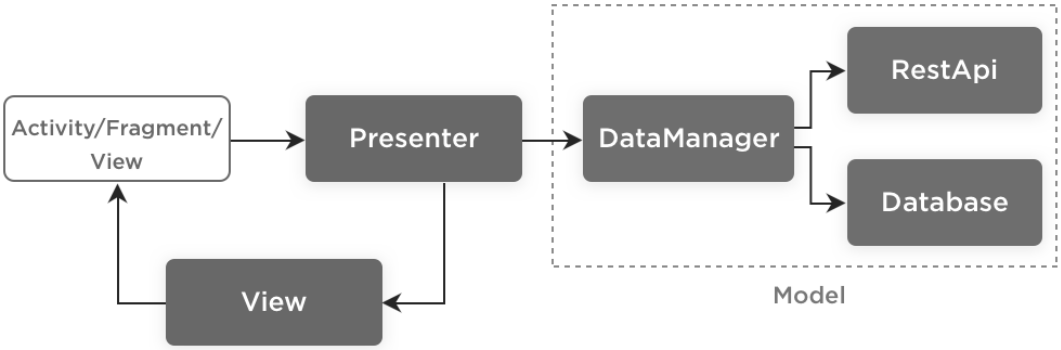
Slika 5: MVP arhitektura, [13]

Ponekad se ovaj uzorak implementira korištenjem četvrtog sloja koji se zove interactor te u tom slučaju taj sloj preuzima odgovornost poslovne logike, a prezenter se bavi samo već obrađenim podacima. Interactor je posrednik modela i prezentera[6].

Slika ispod prikazuje kako bi ovaj uzorak bio implementiran u Android razvoju. Sloj view bi bila kombinacija *Activity* i *Fragment* klasa zajedno s XML layout skriptama i implementacijom View sučelja. View sa slike poziva odgovarajuću metodu presenter-a na korisnički input, šalje mu podatke ili prima podatke koje je potrebno prikazati na ekranu. Sloj model bi bio DataManager koji sadrži reference na vanjske api-e i bazu podataka[6].

U slučaju implementacije MVP uzorka s interactor-om, podaci se šalju njemu na obradu ili se od njega dohvaćaju strukturirani podaci za spremanje u bazu, a u slučaju implementacije bez četvrtog sloja to se obavlja putem presenter-a. Presenter je ovdje zapravo posrednik između pogleda i modela, ne zna ništa o sloju pogleda te sadrži referencu na sloj modela. Glavna funkcija presentera bi bila dohvatiti podatke iz modela i vratiti ih u pogled te eventualno vršiti neku obradu istih[6].

Bitno je naglasiti neke finese kod razvoja mobilnih aplikacija. Recimo, sloj model-a služi za dohvat podataka, no često nam trebaju suženi ili strukturirani podaci te bi dohvaćanje viška izazvalo potrošnju memorije. Kako bi to izbjegli u sloju model-a odvija se mapiranje modela podataka u domenske modele koji zapravo predstavljaju oblik podatka koji nam treba tijekom određenog slučaja korištenja unutar aplikacije. Isto tako, jednom kada mapirani podaci stignu u presenter, koji je pretplaćen na podatke, ovdje se isto može odvijati mapiranje u svrhu prilagodbe podataka za sloj view[6].



Slika 6: MVP android arhitektura, [12]

MVP je široko korišten uzorak u razvoju Android aplikacija jer za razliku od MVC uzorka omogućuje testiranje. U nastavku slijedi programski primjer implementacije ove arhitekture.

4.1.2.1. Primjer *MVP* arhitekture u Kotlinu

Najprije je prikazan kod za sučelje klase *View* s dvije pripadne metode te xml fajlica sa poljima s formom za prijavu korisnika.

***View* paket: IPrijavaView.kt (*Interface*), MainActivity.xml (*Class*)**

```
interface IPrijavaView {
    fun OnPrijavaSuccess(message: String?)
    fun OnPrijavaError(message: String?)
}
```

Programski kod 64: IPrijavaView.kt klasa

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android
    "
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"">

        <androidx.appcompat.widget.AppCompatEditText
            android:id="@+id/inputEmail"
            android:layout_width="match_parent"
```

```
            android:layout_height="55dp"
            android:inputType="text"/>

        <androidx.appcompat.widget.AppCompatEditText
            android:id="@+id/inputPassword"
            android:layout_width="match_parent"
            android:layout_height="55dp"
            android:inputType="textPassword"/>

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="50dp">

            <com.google.android.material.button.MaterialButton
                android:id="@+id/buttonPrijava"
                android:layout_width="match_parent"
                android:layout_height="match_parent"/>
        </RelativeLayout>

    </LinearLayout>
</ScrollView>
```

Programski kod 65: MainActivity.xml fajl

Nadalje slijedi sučelje za podatkovnu klasu Korisnik kao i sama klasa Korisnik. Korisnik sadrži email i lozinku da implementira metode sučelja za dohvat emaila, lozinke te provjeru ispravnosti korisnika.

***Model* paket: IKorisnik.kt (*Interface*), Korisnik.kt (*Class*)**

```
interface IKorisnik {
    fun getEmail(): String?
    fun getPassword(): String?
    fun isValid(): Int
}
```

Programski kod 66: IKorisnik.kt sučelje

```
class Korisnik(
    private val email: String?,
    private val password: String?
) : IKorisnik {

    override fun getEmail(): String? {
        return email
```

```
    }

    override fun getPassword(): String? {
        return password
    }

    override fun isValid(): Int {
        if(TextUtils.isEmpty(getEmail()))
            return  0
        else if(!Patterns.EMAIL_ADDRESS.matcher(getEmail()).matches())
            return  1
        else if(TextUtils.isEmpty(getPassword()))
            return 2
        else if(getPassword()?.length!! <=7)
            return 3
        else
            return -1;
    }
}
```

Programski kod 67: Korisnik.kt klasa

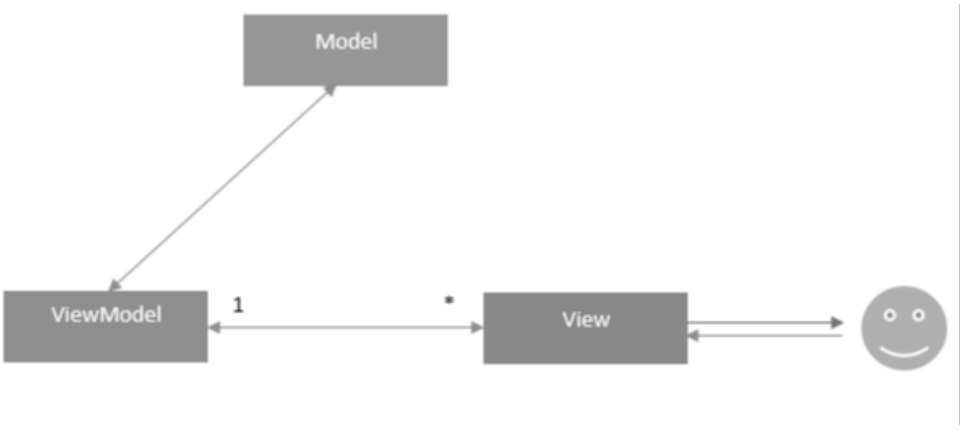
Nakon programsko primjera korištenja *MVP* uzorka arhitekture, slijedi dio rada gdje će na isti način biti prikazana *MVVM* arhitektura, odnosno samo teorijski jer je praktični primjer kreiran korištenjem *MVVM* uzorka.

4.1.3. *MVVM*

MVVM je kratica za Model-View-ViewModel, a osnovna značajka mu je to što podržava dvosmjerno povezivanje između view-a i view Model-a. To je jedna od novijih arhitektura koja je službeno podržana od strane Google-a te su u većini alata dostupne pomoćne klase i kosturi klasa za ovu arhitekturu[6].

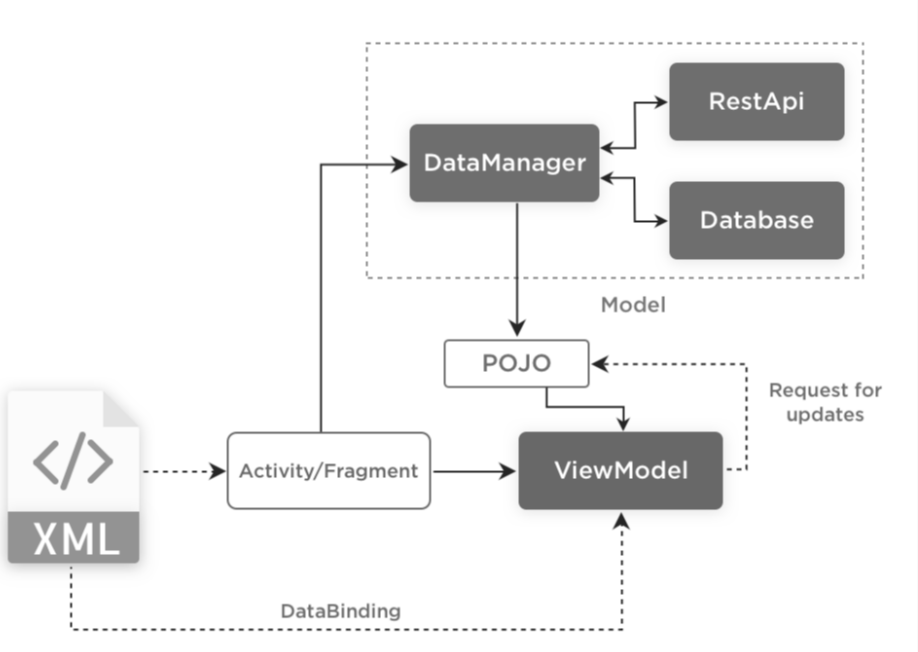
Model u ovom uzorku je jednak kao i kod ranije opisanih uzorka pa se neće dodatno objašnjavati. View također predstavlja isto što i u druga dva uzorka, no sličan je view sloju u MVC arhitekturi jer je također pretplaćen na podatke, no ne u sloju model već u sloju ViewModel. Sloj ViewModel je ovdje zadužen za poslovnu logiku i poslovna pravila[6].

Slika broj 7 predstavlja primjer prakticiranja MVVM uzorka u android razvoju. View sloj je kombinacija XML i Activity klasa. Activity klasa zatraži podatke te ih proslijedi ViewModel-u koji zatim ažurira korisničko sučelje. DateManager je sloj modela kao kod MVP uzorka. ViewModel je posrednik između sloja view i sloja modela, dobiva podatke iz sloja model i ažurira sloj view, a uz to manipulira sa stanjima modela. Ovdje vidimo prethodno objašnjenu razliku MVVM uzorka i MV uzorka, naime DateManager vraća odgovor Activity klasi, a ne presenter-u. To znači da je Activity klasa svjesna poslovne logike, a samim time i sloj view[6].



Slika 7: MVVM arhitektura, [13]

Kada je u pitanju Android razvoj, MVP i MVVM uzorak dizajna arhitekture nude bolju modularnu arhitekturu od MVC-a, ali isto tako mogu dodati veću složenost aplikaciji. MVC ne možemo u potpunosti odbaciti. U jednostavnijim aplikacijama koje uključuju dva ili više zaslona te koje u budućnosti neće zahtijevati nadogradnje, MVC može dobro funkcionirati. Radi li se o složenijim slučajevima, gdje je aplikacija veća te gdje se očekuju nadogradnje, preporučuje se MVVM ili MVP uzorak. Njima ćemo ostvariti modularnost, nadogradivost i izbjeći nagomilani i redundantni kod[6]. U nastavku slika 8 prikazuje MVVM arhitekturu u Android razvoju.



Slika 8: MVVM android arhitektura, [12]

U nastavku slijedi osvrt na prethodna tri uzorka arhitekture sa zaključkom koja arhitektura bi bila najbolja praksa u android razvoju.

4.1.4. Najbolja arhitekturalna praksa

Bitno je spomenuti i mogućnost agilnog razvoja koji danas vrlo čest na tržištu. Dakle, izmjene aplikacija su česte i uobičajna praksa tijekom samog razvoja. Cilj je razvijati aplikacije tako da imaju mogućnost izmjena odnosno da možemo biti agilni. Neki od parametara koji bi mogli pokazati koji uzorak arhitekture je agilan i u kojoj mjeri su: broj klasa koje se mogu modificirati, broj klasa koje mogu dodati novu funkcionalnost, te koliko linija koda moramo napisati kako bi dodali novu funkcionalnost. MVP je jedna od arhitektura koje je najviše moguće mijenjati i održavati, no kod MVVM uzorka najčešće trebamo pisati manje linija koda. MVC je najmanje agilan uzorak arhitekture[6].

Osim agilnog razvoja, vrlo bitna stavka koja bi se trebala sagledati kod odabira uzorka arhitekture jest mogućnost testiranja. Ranije je već spomenuto kako MVC značajno otežava testiranje aplikacije, te je pokrivenost metoda i linija koda testovima vrlo niska. Kod MVP arhitekture mogućnost testiranja je značajno bolja, no još uvijek nije moguća sto postotna pokrivenost testovima. MVVM uzorak je uzorak koji upravo to omogućava te ga to čini najboljim izborom po pitanju testiranja[6].

Kada bi odredili koji bi od opisanih uzoraka arhitekture bila najbolja praksa u razvoju mobilnih aplikacija morali bismo se složiti s Google-om. Da, najbolja opcija je MVVM. Naravno, na odabir značajno utječe tip, veličina i svrha aplikacije, a i MVP se pokazao kao kvalitetan odabir, no ipak teško je zamisliti razvoj profesionalnih aplikacija bez provođenja pedantnih testiranja[6].

4.2. Arhitektura na strani poslužitelja

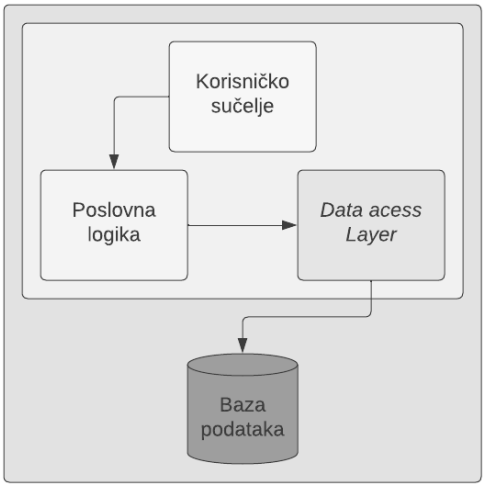
Ovo poglavlje će se pozabaviti arhitekturom mobilnih aplikacija na strani poslužitelja. Bit će opisana tri najpoznatija pristupa u arhitekturi te će se prikazati sveobuhvatni zaključak o tome što bi bila najbolja praksa[7]. Najprije slijedi potpoglavlje o monolitnoj arhitekturi, zatim o servisno orijentiranoj arhitekturi, a na kraju slijedi mikrosevisna arhitektura.

4.2.1. Monolitna arhitektura

Aplikacija koja sadrži monolitnu arhitekturu razvijena je na jednom mjestu odnosno sustav je jedna cjelina te je dostupna na jednom endpoint-u. Jednostavne su za postavljanje na server kao i za testiranje, ali bitno je naglasiti i jednu manu, a to je da se prilikom postavljanje ovakve aplikacije na server, uvijek treba aplikaciju ugasiti neovisno o veličini promjene[7].

Označava ih brz razvoj, a preporučuje se za prethodno opisane MVP projekte, ali i za projekte s malim prometom podataka. MVP projekti ovdje profitiraju jer se dohvaćanje podataka obavlja brzo i jeftino bez gubljenja vremena za komunikaciju među servisima. Bitno je naglasiti da ovakva arhitektura donosi brz razvoj isključivo ako se koristi onda kada treba, a to je za slučaj vrlo malog projekta koji koristi MVP uzorak arhitekture, U slučaju prakticiranja ove arhitekture na većim projektima dolazi do raznih poteškoća, a to su: prevelik razvojni tim, ne agilan razvoj, usporavanje razvojnog tima radi arhitekturalnih poteškoća, velika kompleksnost koda te recimo

otežano dovođenje novih programera te njihova prilagodba na projekt[7]. Na slici broj 9 je prikazan primjer monolitne arhitekture.



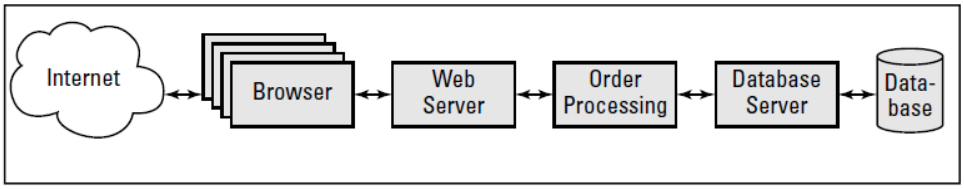
Slika 9: Arhitektura monolitne aplikacije, [Autorski rad]

Ovaj pristup se još uvijek koristi i postoje slučajevi kada bi odabir ove arhitekture bio najbolja opcija (Kod jako malih aplikacija, recimo aplikacija koja radi s vrlo malim skupom podataka i služi za internu upotrebu u IT tvrtki), no ipak obujam upotrebe ove arhitekture je u padu te se u poslovnom svijetu smatra zastarjelim i vrlo skupim za razvoj većih aplikacija. Kako bi se otklonili nedostaci ove arhitekture pojavili su se servisno orijentirani principi[7].

Nakon opisane monolitne arhitekture, u nastavku slijedi potpoglavlje koje će se baviti servisno orijentiranom arhitekturom.

4.2.2. Servisno orijentirana arhitektura

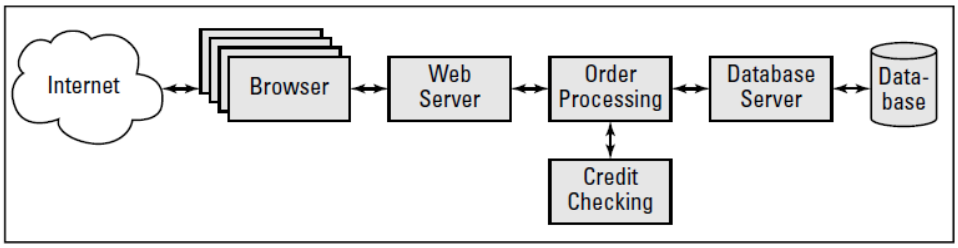
Servisno orijentirana arhitektura (*SOA*) nije nimalo mlad pojam, danas je zapravo standard u izgradnji aplikacija te je kroz godine postojanja razvijeno više koncepata temeljenih na (*SOA-i*) kao što je mikroservisna arhitektura. Pojam se prvi put pojavio 1998. godine i od tada mu je sve veća popularnost. Ova arhitektura uvodi koncept izgradnje aplikacija iz više komponenti[7].



Slika 10: *Jednostavna arhitektura, [7]*

Kako bi se arhitektura jednostavnije opisala gore je prikazana slika. Radi se o jednostavnom sustavu gdje informacije dolaze od preglednika do web poslužitelja, nakon toga se zahtjev obrađuje te se odlučuje što dalje učiniti. Ovisno o zahtjevu može se recimo zatražiti po-

datak iz baze podataka ili poslati neki podataka na arhiviranje u bazu. Nakon što se obavi što je traženo, određena informacija se šalje natrag na preglednik, dakle radi se o obostranoj komunikaciji. Sustavi u poslovnom svijetu, često su kompleksi te pokrivaju veliki broj slučaja korištenja. Zbog toga se pojavila potreba za drugačijom arhitekturom te se uvode takozvani servisno orijentirane komponente koje su omogućile da se određene usluge odnosno funkcionalnosti unutar aplikacije odvoje u samostalnu komponentu. Slika ispod prikazuje jedan takav primjer gdje je funkcionalnost plaćanja kreditnom karticom izdvojena u zasebnu komponentu[7].



Slika 11: *Servisno orijentirana arhitektura, [7]*

Takve odvojene komponente u ovoj arhitekturi nazivamo servisima, a oni imaju 4 osnovne odrednice: logički predstavljaju poslovnu aktivnost sa specifičnim rezultatom ili izlazom, za klijente su crna kutija, mogu sadržavati drugi servis te su samostalni.

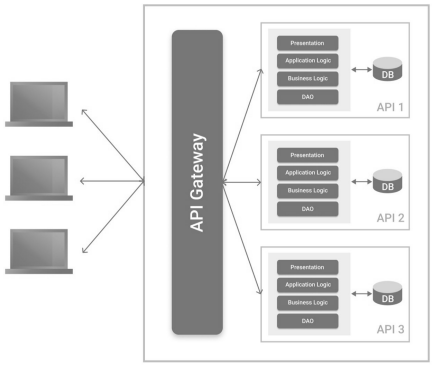
Mnogo je prednosti servisno orijentirane arhitekture, osobito u poslovanju temeljenom na web uslugama. Korištenjem ove arhitekture stvara se ponovno upotrebljivi kod pa tako i cjelokupne komponente što eliminira ponovno programiranje istih funkcionalnosti unutar neke organizacije. Ukoliko postoji potreba za korištenjem različitih programskih jezika, ovom arhitekturom je to također omogućeno jer pojedino komponente komuniciraju putem sučelja. Nadalje, omogućen je razvoj neovisnih sustava gdje se smanjuje interakcija *klijent-servis*, a i time se automatski ostvaruje veća skalabilnost. Uz to, razvoj ovakvih aplikacija smanjuje troškove samih sustava za organizaciju. Bitno je još naglasiti skrivenu kompleksnost, odnosno ako napravimo određene promjene na bazi podataka, to ne utječe na klijenta.

Naravno i praktični dio ovog rada će se baviti servisno orijentiranom arhitekturom, obzirom da se rad bavi najboljom praksom, ali radit će se o nešto specijaliziranijoj vrsti servisno orijentirane arhitekture za koju se kaže i da je evolucija SOA-e. Radi se o mikroservisnoj arhitekturi koja je objašnjena u sljedećem poglavlju.

Nakon opisane servisno orijentirane arhitekture, u nastavku slijedi potpoglavlje koje će se baviti mikroservisnom arhitekturom.

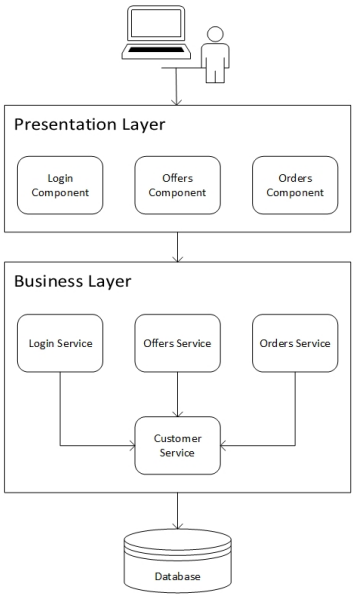
4.2.3. Mikroservisna arhitektura

Arhitektura mikroservisa postala je dobro istražena i omiljena arhitektura u izgradnji velikih složenih aplikacija, a često se naziva i mikroservisna arhitektura. Mikroservisi omogućuju razgradnju velike monolitne aplikacije na manje povezane entitete koji mogu komunicirati jedni s drugima putem API-a. Svaki pojedini servis je modeliran oko jedne poslovne domene, a oni zajedno čine jednu složenu aplikaciju.[8] Primjer takve arhitekture prikazan je slikom broj 12.



Slika 12: *Primjer mikroservisne arhitekture*, [12]

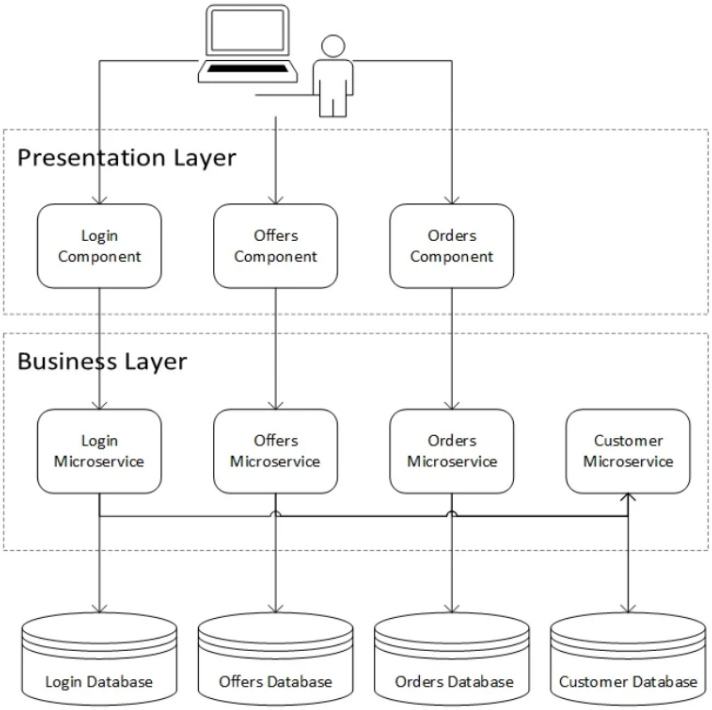
Na slici broj 13 vidimo klasičnu n-slojnu arhitekturu te primjenu SOA-e. Svaka komponenta na slici uključuje logiku za interakciju s klijentom u određenoj poslovnoj aktivnosti i da bi to učinila, koristi usluge koje pruža poslovni sloj. Svaka usluga predstavlja realizaciju neke poslovne aktivnosti.



Slika 13: *Primjer SOA arhitekturu u n-slojnu arhitekturi*, [7]

Dalje gledajući sliko broj 13, prijava u aplikaciju koristi servis za prijavu, provjera ponuda koristi servis za ponude, kreiranje narudžbe koristi servis za narudžbe itd. Te su usluge samostalne u poslovnom sloju i djeluju kao crna kutija za svoje klijente, odnosno nije im poznato kako su usluge implementirane. Dalje vidimo da sve usluge ovise o korisničkoj službi za dobivanje ili slanje podataka prema bazi podataka.

Na slici broj 14 sada vidimo primjer n-slojne arhitekture ali implementirane mikroservisima. Slojevi nisu međusobno povezani, a svaka mikrousluga potpuno je odvojena od ostalih usluga, posjeduju vlastite podatke i mogu se mijenjati bez utjecaja jedna na drugu.



Slika 14: *Mikroservisna arhitekturu u n-slojnoj arhitekturi*, [7]

Tu vidimo jednu jasnu razliku SOA i mikroservisa, a to je da je mikroservisna arhitektura bazirana na manjim servisima koji su puno više specijalizirani odnosno fokusirani na određenu svrhu te neovisni jedni o drugima, dok su SOA servisi puno veći te ovisni jedni o drugima. Ta samostalnost komponenti u mikroservisnoj arhitekturi ju čini puno otpornijom na greške, omogućuje lakše postavljanje novih verzija aplikacije na server (donosi DevOps kao standardnu praksu svih razvojnih timova) i naravno omogućuje lakšu i bržu skalabilnost. Još jedna bitna razlika su spremišta podataka jer u mikroservisnoj arhitekturi najčešće svaki mikroservis ima svoje vlastito spremište podataka (kao što je vidljivo na slici 10 i 13), dok servisi u SOA arhitekturi dijele isto spremište[8].

Dakle, infrastruktura za mikroservise je obično jednostavnija, jer nema toliko složenih poslužitelja za upravljanje konfiguracijom, nadzorom i kontrolom te nema ogromnih shema baza podataka. Time omogućuje razvijanje veče stručnosti tima tako što se pojedinci lakše mogu specijalizirati za određene poslovne domene i time organizacija može priučiti značajniju dodanu vrijednost svojim proizvodima ili uslugama. Osim toga, mikroservisna arhitektura omogućuje održavanje manjih timova[8].

Nadalje, SOA se temelji na takozvanom dijeljenju komponenti, a mikroservisi se temelje na konceptu 'ograničenog konteksta'. Ograničeni kontekst je spajanje komponenti i njezinih podataka bez ovisnosti čime se smanjuje potreba za dijeljenjem komponenti. Ova vrsta spajanja rezultira visokom kohezijom pa se sve točke kvara u određenoj usluzi brzo izoliraju i rješavaju prije nego što se ugrozi rad aplikacije[8].

Bitno je spomenuti i komunikaciju te protokole pristupa. Što se tiče komunikacije u SOA arhitekturi, često se koristi ESB (*engl. Enterprise service bus*) za komunikaciju među aplikacijama što je dosta zastarjelo te usporava procese, a s druge strane mikroservisna arhitektura obavlja sve komunikacije jednostavnim slanjem poruka putem API-a. Takva komunikacija je brža te nije ovisna o platformama odnosno jeziku u kojem je razvijen dio aplikacije ili neka druga aplikacija. Što se tiče udaljenog pristupa, SOA koristi neke od protokola kao što su SOAP (*engl. Simple Object Access Protocol*), AMQP (*engl. Advanced Messaging Queuing Protocol*) i MSMQ (*engl. Microsoft Messaging Queuing*), a mikroservisi najčešće koriste protokol REST (*engl. Representational State Transfers*) ili jednostavno slanje poruka poput JMS (*engl. Java Messaging Service*). Protokoli koje koriste mikroservisi su homogeniji, dok SOA koristi protokole koji su više za heterogenu interoperabilnost. Bitno je nadglasati da se u današnjem poslovnom svijetu prakticira korištenje REST protokola kao i JMS poruka i u SOA arhitekturi, no to nije specifično za sustave većih razmjera[8].

Zaključno, ako pogledamo SOA i zatim mikroservise, ono što možemo vidjeti je da su mikroservisi prirodna evolucija. Arhitektura mikroservisa koristi prednosti SoA-e, ali definira dodatne korake za izgradnju komponenti. Dakle, definitivno možemo reći sljedeće: Svi mikroservisi su SOA, ali svi SOA nisu mikroservisi." Gore u tekstu se kroz analizu razlike dvaju arhitektura već daju naslutiti neke karakteristike koje se mogu identificirati kao principi dizajna, no un nastavku skijedi opis svakog pojedinog principa.[8]

Prvi princip jest modeliranje bazirano na sposobnosti poslovanja. Dobro osmišljena mikroservisa arhitektura trebala bi biti oblikovana s obzirom na sposobnost poslovanja. Dizajniranje softvera ima komponentu apstrakcije te je standardna praksa nešto implementirati temeljem zahtjeva, ali moramo razmotriti kako će svi, uključujući i razvojni tim, razumjeti rješenje, sada i u budućnosti. Kada se mikroservisi trebaju ažurirati odnosno modificirati, moramo se vratiti natrag na apstraktnu razinu odnosno izvorni koncept koji ih je definirao. U tom smo procesu moguće je otkriti da stvarno stanje nije onakvo kakvo je izvorno shvaćeno, da ne postoji potreba za izvornom idejom ili da je ona već implementirana negdje drugdje, a i moguće je doći do zaključka da je potrebno pomaknuti granice domene. Prilikom tog procesa, u praksi je jako korisno surađivati sa stručnjacima specijaliziranim za poslovnu domenu kojom se proizvod bavi. Ovakav princip dizajna također približava koncept *Domain-Driven* dizajna[8].

Dalje, postoji princip slabe povezanosti. Naime, nijedna mikrousluga ne postoji sama za sebe, budući da svaki sustav treba komunicirati s drugima, ali to se ovdje implementira što je "labavije" moguće. Ovo je najbolje objasniti primjerom, pa tako možemo zamisliti da dizajniramo mikrouslugu koja vraća dostupne ponude nekog kupca, te prilikom prikazivanja tih ponuda, mikroservis ispisuje ima kupca čije su ponude. Dakle, komunicira s nekom drugom mikrosulugom koja se bavi kupcima. Recimo da se ta druga mikrosuluga modificira tako da se

promjeni oblik podatka koji se vraća na upite drugih mikroservisa. U tom slučaju bi svi ostali mikroservisi koji komuniciraju s mikrosevisom za kupce trebali ostati isti, odnosno dio koji prima odgovor s tog mikroservisa ne smije ovisiti o obliku samog podatka[8].

Treći princip jest princip jedinstvene odgovornosti. Ono govori da bi svaki mikroservis trebao biti odgovoran za jedan dio funkcionalnosti koju pruža aplikacija, a tu bi odgovornost mikroservis trebao u potpunosti obuhvatiti. Dizajn mikroservisa trebao bi biti usko usklađen s tom odgovornošću. Mogli bismo usvojiti definiciju Roberta C. Martina o principu primijenjenom na OOP koji kaže: "Klasa bi trebala imati samo jedan razlog za promjenu"; za ovo načelo možemo takođerreći: mikroservis treba imati samo jedan razlog za promjenu. Ako desi situacija da kada trebamo promijeniti poslovnu funkciju unutar naše aplikacije, zapravo modificiramo nekoliko mikroservisa ili da promjena kaskadno prelazi u nepovezane mikroservise, vrijeme je da preispitamo kako ih dizajniramo, jer u tom slučaju ovaj princip je prekršen[8].

Nadalje postoji princip skrivene implemnetacije koje govori da mikroservisi trebaju imati jasno i lako razumljivo sučelje koje mora sakriti detalje implementacije. Ne bi se trebali izlagati interni detalji, niti tehnička implementacija te poslovna pravila. Ovime se omogućuje sposobno modificiranja određenih komponenti bez da se utječe na cjelokupnu aplikaciju.

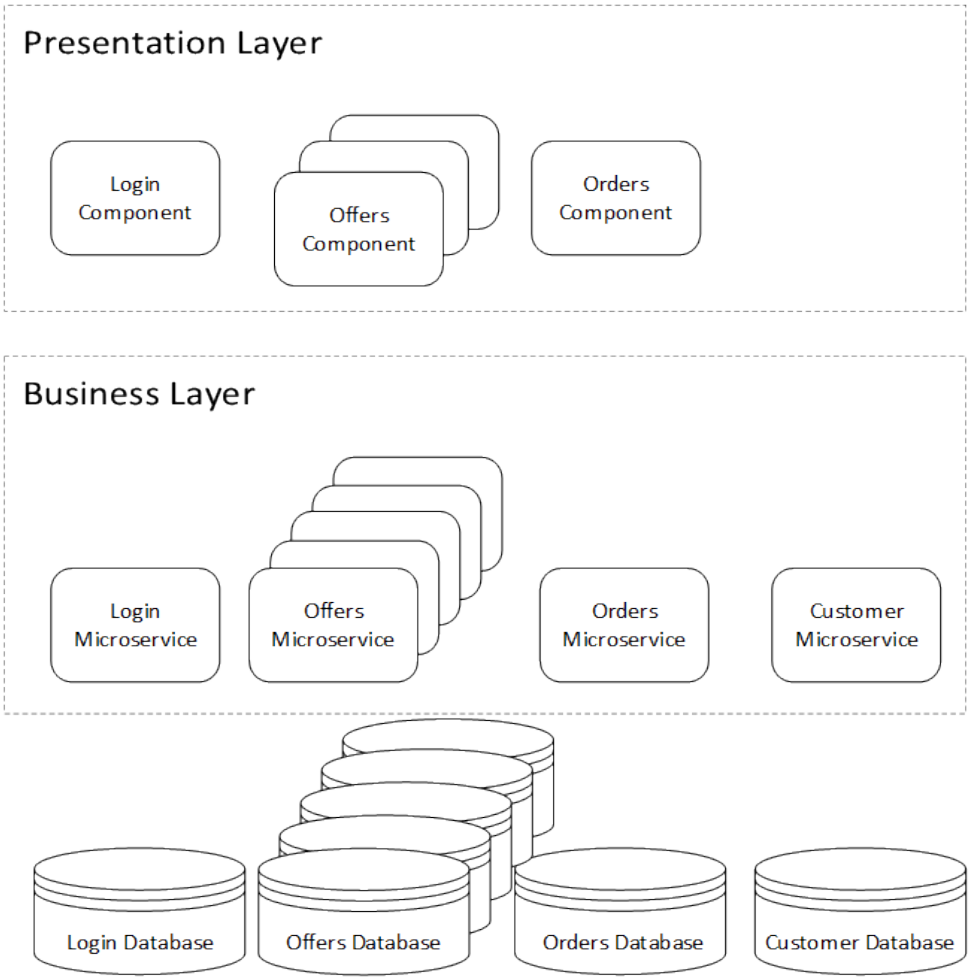
Princip izoliranosti govori da mikroservis treba biti fizički i/ili logički izoliran od infrastrukture koja koristi sustave o kojima ovisi. Ako koristimo bazu podataka, to mora biti naša baza podataka, ako radimo na poslužitelju, trebala bi biti na našem poslužitelju, i tako dalje. Time jamčimo da ništa vanjsko ne utječe na nas, a ni mi ne utječemo na bilo što vanjsko. Ovo će omogućiti izvedbu i praćenje performansi, ali i pružiti kontinuirani cjevovod isporuke. Lakše će biti izvoditi nadogradnje, skalirati, upravljati kvarovima i slično[8].

Mikrousluge bi se trebale moći samostalno implementirati odonso ovdje se radi o principu odvojene instalacije. Ukoliko to nije moguće, to vjerojatno znači da postoji neka vrsta sprege unutar arhitekture koju treba riješiti odnosno da pravila arhitekture nisu ispoštovana. Mogućnost stalne isporuke jedna je od prednosti ove arhitekture te bi trebala biti omogućena[8].

Mikroservisi bi trebali biti otporni na pogreške, granularna priroda ove arhitekture ju već u startu čini otpornijom za pogreške te lakšom za testiranje, no nije se dovoljno osloniti samo na to. Prilikom početnog planiranja dizajna potrebno je već identificirati osnovne potencijalne pogreške te razviti dizajn kojim bi se zaobišle. Nadalje, kako sustav raste, kontinuirano treba razmišljati, analizirati te isplanirati upravljanje svim rubnim scenarijima i situacijama koje mogu poći po zlu. Uz to, u razvoju je potrebno donesti odluku kako pratiti, kontrolirati te koje alate koristiti za upravljanje pogreškama u radu sustava[8].

Dalje, mikroservisi bi trebali biti dizajnirani da budu neovisno skalabilni. Ako trebamo povećati broj zahtjeva koje možemo obraditi ili koliko zapisa možemo držati, trebali bismo to učiniti izolirano. To znači da bismo trebali moći skalirati određene usluge bez da skaliramo sustav u cjelini. Možemo reći da u dobro osmišljenoj mikroservisnoj arhitekturi možemo ostvariti veći kapacitet s manje infrastrukture. Primjer takvog učinkovitog i izoliranog skaliranja prikazano je na slici ispod[8].

Prilikom samog dizajna mikroservisa trebamo voditi računa o automatizaciji nekih me-



Slika 15: Skaliranje u mikroservisnoj arhitekturi, [7]

hanizama recimo u području instalacije, izgradnje, testiranja te monitoringa. Najbolja praksa bi bila da su svi mehanizmi unutar aplikacije automatiziraju ako je to moguće. Ovime dobivamo veću agilnost sustava i daljnjeg razvoja[8].

Na kraju ovog poglavlja obrađene su arhitekture u mobilnom razvoju kroz opisivanje osnovnih obrazaca dizajna aplikacije do opisa najšire korištenih arhitektura na strani poslužitelja. Ovime je ostvaren jedan od ciljeva diplomskog rada, a to je bavljenje arhitekturama android aplikacije, ali isto tako donesen je zaključak o tome koji će se uzorak dizajna aplikacije koristiti pri razvoju te koja će se od servisno orijentiranih arhitektura koristiti. S obzirom na to da je mikroservisna arhitektura najnoviji koncept od svih opisanih time je najatraktivniji te odabran da bude dio praktičnog rada.

Nakon teme arhitekture aplikacija, slijedi predzadnji veći odlomak rada koji će se baviti radom s bazama podataka u Kotlin razvoju.

5. Rad s bazama podataka u Kotlin razvoju

Jedan od ciljeva ovog rada bilo je istražiti i analizirati najbolje prakse u radu s NoSQL te SQL bazama podataka u razvoju mobilnih aplikacija s Kotlinom. Ovdje nećemo ulaziti u već poznate teorijske koncepte kao što su definicija relacijskih baza i slično nego će se poglavlje direktno pozabaviti nekim od najšire korištenim tehnologijama za rad s bazama prilikom razvoja Android aplikacija u Kotlinu. Svaka od mogućih pristupa bit će popraćena praktičnim primjerima koji su izvedeni radi detaljnijeg razumijevanja, ali i jasnije odluke o tome koju tehnologiju koristiti u praktičnom djelu rada te ju time još dublje upoznati.

Budući da su podaci i pohrana podataka uobičajeni zahtjevi u većini mobilnih aplikacija, često se postavlja pitanje koja je baza podataka najbolja za korištenje. Baza podataka za Android je oblik trajne pohrane podataka namijenjen za korištenje u aplikacijama za Android uređaje, a često se sastoji i od lokalne pohrane na uređaju tako da je aplikacija i dalje dostupna čak i ako uređaj izgubi vezu. Iako sve baze podataka pružaju mogućnost pohranjivanja, postavljanja upita i rukovanja podacima, nisu sve baze podataka jednake te prije samog odabiramo moramo razumjeti zahtjeve sustava. Neka pitanja koja si trebamo postaviti prije odabira su:

Kakvu vrstu podataka je potrebno pohraniti? - ovdje treba znati hoće li to biti strukturirani ili nestrukturirani podaci ili pak velike datoteke, a osim toga moramo znati je li to tip podatka koji se može pohranjivati lokalno na uređaju ili će pak trebati biti široko dostupni.

Kako pristupiti svojim podacima? - ovdje vodimo računa o tome hoće li pristup biti jednostavan ili kompliciran te hoćemo li morati pisati komplicirane i dugačke upite.

Kako će korisnici komunicirati s podacima aplikacije? - moramo znati recimo, očekujemo li da naš sustav daje podatke u stvarnom vremenu te hoće li ti podaci trebati biti dostupni drugim korisnicima i uređajima.

Nakon što se prikupe odgovori na navedena pitanja potrebno je prije same tehnologije za bazu podataka odabrati tip ili vrstu baza, a u nastavku će biti navedena i kratko opisana tri tipa baza, a to su relacijske baze podataka, *NoSQL* baze podataka te objektno-orijentirane baze podataka.

Najprije relacijske baze podataka kao prvi tip koji će se spomenuti. Za koje se može reći i da su najšire korištene, a u Android razvoju najšire korišteni relacijski jezik jest *SQLite* koji je dostupan unutar Android SDK paketa. Za rad sa SQLite bazama podataka Android nudi *Room* tehnologiju koja će biti obrađena u idućem potpoglavlju. Nadalje, drugi tip baza podataka su *NoSQL* baze podataka koje se sve više koriste u mobilnom razvoju. Rad će se pozabaviti komparativnom usporedbom dvije najšire NoSQL platforme za razvoj mobilnih aplikacija, a to su *MongoDB* i *Firebase*. I na kraju treći tip baza podataka su baze koje mogu raditi sa složenim podatkovnim objektima kao što se to radi u objektno orijentiranim programskim jezicima te ima sposobnost brzih upita kod složenih podataka. Najpoznatija tehnologija za ORM baze podataka u Android razvoju je *Realm*.

5.1. Android *Room* u Kotlinu

Room baze podataka su dio komponenti Android osnovne arhitekture te pruža sloj apstrakcije preko *SQLite* baze podataka, dakle omogućuje robusniji pristup bazi podataka, ali pritom održava punu snagu *SQLite* baze podataka.[9]

Prednosti koje donosi Room tehnologija jesu: provjera SQL upita tijekom samog kompajliranja, omogućuje korištenje programskih anotacija čime smanjujemo nakupljanje standardnog koda te daje laku integraciju s drugim arhitekturalnim komponentama kao što su LiveData RxJava[9].

Postoje određene razlike Room baza podataka i *SQLite* baza podataka. Kod *SQLite* tehnologije ne postoji provjera upita kod kompajliranja, prilikom ažuriranja shema Room baza podataka se sama pobrine za prilagodbu postojećih upita novim provjerama dok u *SQLite* bazi to trebamo sami odraditi. Nadalje, postoji jako puno nepotrebnog koda koji moramo pisati kod konverzije podataka *SQL* upita u Java ili Kotlin objekte, dok *Room* ima mehanizam za mapiranje objekata, a isto tako *Room* je izgrađen kako bi mogao raditi s *LiveData* i *RxJava* mehanizmima[9].

U nastavku slijedi slika 16 koja prikazuje arhitekturu Android *Room* tehnologije. Dakle, postoje 3 glavne komponente, najprije Entiteti, odnosno prikaz tablice s pripadnim stupcima.

Slika 16: Android Room arhitektura, [14]

Klasa izgleda tako da se označi s *@Entity* anotacijom, kao naziv klase moramo uzeti ime tablice, a podatkovni atributi se nazivaju prema stupcima iz tablice, dakle *Entity* klase predstavljaju entitete u tablici[9]. U nastavku slijedi primjer takve klase.

```
@Entity(tableName = "user")
data class Users(
```

```
    @PrimaryKey(autoGenerate = true)
    var id: Int? = null,
    val firstName: String,
    var lastName: String,
    val email: String
)
```

Programski kod 68: Primjer *Entity* klase

Drugi element su takozvane Dao klase ili Data Access Object klase. To su zapravo sučelja gdje stavimo sve potrebne SQL upite, a osnovni upiti se ne moraju pisati odnosno ugrađeni su, te je potrebno samo staviti ime metode te ju anotirati s pripadnom oznakom, ovisno o SQL upitu[9]. Osnovne oznake anotacija su seljedeće: @Insert za upis jednog retka podatka, @Delete za brisanje jednog retka podataka, @Update za ažuriranje jednog retka podataka te @Query koji se koristi se za kreiranje specifičnih upita, recimo "(SELECT FROM*)"[9]. U nastavku slijedi primjer DAO klase.

```
@Dao
interface UserDao {
    @Insert
    fun insertUser(users: Users)

    @Query("Select * from user")
    fun gelAllUsers(): List<Users>

    @Update
    fun updateUser(users: Users)

    @Delete
    fun deleteUser(users: Users)
}
```

Programski kod 69: Primjer *DAO* klase

Treći dio arhitekture Room baze podataka je apstrakta klasa koja proširuje *RoomDatabase* klasu, a anotirana je s oznakom @Database. Ovdje se definiraju entiteti, verzija baze podataka, sadrži nositelje baze podataka te služi kao glavna pristupna točka za konekciju na bazu[9]. U nastavku slijedi primjer klase s anotacijom @Database.

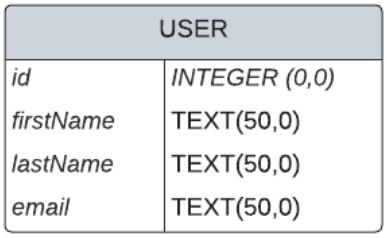
```
@Database(entities = [Users::class], version = 1, exportSchema = false)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao() : UserDao
}
```

Programski kod 70: Primjer *@Database* klase

Nakon opisa *Room* baze podataka te nekih osnovnih programskih primjera, u sljedećem potpoglavlju slijedi konkretni primjer korištenja ove baze nad entitetom korisnika.

5.1.1. Praktični primjer korištenja *Room* baze podataka

U ovom poglavlju slijedi praktični primjer korištenja *Room* baze podatak koji će se izvesti nad entitetom korisnika. Za potrebe primjera napravljena je baza s jednim entitetom, a slika broj 17 prikazuje ERA model korištene baze, odnosno u ovom slučaju model s jednom tablicom.



Slika 17: SQLite User tablica, [Autorski rad]

Najprije je potrebno dodati ovisnosti za *Room* bazu podataka u *build.Gradle* datoteku. Nakon toga je potrebno kreirani model klase za sve entitete koje ćemo koristiti te ih anotirati s *@Entity*. Ovdje je bitno voditi računa o tome da postavimo primarni ključ korištenjem upravo anotacije *@primaryKey* kod odgovarajućeg atributa data klase, koji predstavlja stupac primarnog ključa iz tablice. za postavljanje imena atributa koji predstavljaju imena kolumni koristimo anotaciju *@ColumnInfo(name = "column_name")*. Ako postoji više konstruktora koristimo anotaciju *@Ignore* kako bi dali room bazi podataka do znanja koji treba koristiti, a koji ne. Kod kreirane Data klase *User* za ovaj jednostavni primjer izgleda ovako:

```
@Entity(tableName = "user")
data class User(
      @PrimaryKey(autoGenerate = true)
      var id: Int? = null,
      val firstName: String,
      var lastName: String,
      val email: String
)
```

Programski kod 71: Primjer *Data* klase *User*

Nadalje, kreiramo *DAO* klasu gdje navodimo metode kojima pristupamo bazi podataka, u ovom primjeru bit će definirani samo osnovni SQL upiti što se vidi u sljedećem kodu.

```
@Dao
interface UserDao {
```

```
    @Insert
    fun insertUser(users: Users)

    @Query("Select * from user")
    fun gelAllUsers(): List<Users>

    @Update
    fun updateUser(users: Users)

    @Delete
    fun deleteUser(users: Users)

}
```

Programski kod 72: Primjer *DAO* klase za entitet *User*

Za potrebe ovog primjera kreirana je i klasa "*Converters*", koja se koristi kada deklariramo svojstva koje *Room* i *SQL* baze ne znaju serijalizirati. U nastavku kod kreirane klase serijalizira tip podatka *List<String>*.

```
class Converters {

    @TypeConverter
    fun fromString(value: String): List<String> {
        val listType = object : TypeToken<List<String>>() {

        }.type
        return Gson().fromJson(value, listType)
    }

    @TypeConverter
    fun fromArrayList(list: List<String>): String {
        val gson = Gson()
        return gson.toJson(list)
    }
}
```

Programski kod 73: Primjer klase *Converters*

Sljedeći korak jest kreiranje klase s anotacijom *@Database* koja proširuje klasu *RoomDatabase* iz Android paketa, a kod izgleda ovako:

```
@Database(entities = [Users::class], version = 1, exportSchema = false)
@TypeConverters(Converters::class)
```

```
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao() : UserDao

    companion object {
        private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase? {
            if (INSTANCE == null) {
                synchronized(AppDatabase::class) {
                    INSTANCE = Room.databaseBuilder(context
                    .applicationContext,
                        AppDatabase::class.java, "user.db")
                        .allowMainThreadQueries()
                        .build()
                }
            }
            return INSTANCE
        }
        fun destroyInstance() {
            INSTANCE = null
        }
    }
}
```

Programski kod 74: Primjer *@Database* klase za entitet *User*

Ono što je bitno ovdje uočiti jest da je deklarirana apstraktna funkcija za entitet *User*, odnosno potrebno je deklarirati takvu funkciju za svaki entitet ako ih ima više, a takva funkcija vraća *DAO* objekt. Vidljivo je i da postoji deklariran objekt putem kojeg dobivamo statički pristup metodi *getAppDatabse* koja nisam daje *Singleton* instancu baze podataka. Nadalje je potrebno definirati klasičnu *repository* klasu koja sadrži *CRUD* operacije, a kod za takvu klasu izgleda kao u primjeru programskog koda broj 75.

```
class UserRepository(context: Context) {

    var db: UserDao = AppDatabase.getInstance(context)?.userDao()!!

    fun getAllUsers(): List<Users> {
        return db.gelAllUsers()
    }

    fun insertUser(users: Users) {
        insertAsyncTask(db).execute(users)
    }
```

```
    fun updateUser(users: Users) {
        db.updateUser(users)
    }

    fun deleteUser(users: Users) {
        db.deleteUser(users)
    }

    private class insertAsyncTask internal constructor(private val
    usersDao: UserDao) :
        AsyncTask<Users, Void, Void>() {

        override fun doInBackground(vararg params: Users): Void? {
            usersDao.insertUser(params[0])
            return null
        }
    }
}
```

Programski kod 75: Primjer *Repository* klase za entitet *User*

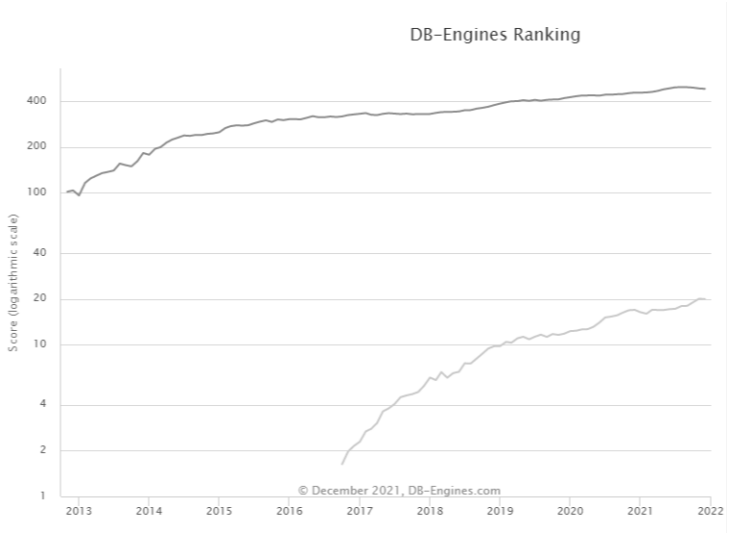
Ovdje je bitno primijetiti upotrebu *AsyncTask* klase. Naime ako se gornji kod pokuša izvršiti na glavnoj dretvi, aplikacija će se srušiti jer je u *Room* bazama podataka ugrađen mehanizam provjere koji detektira ako se izvođenje upita usmjerava na glavnu dretvu, te u tom slučaju ne dopušta izvršavanje upita. Razlog ovoga su smetnje u radu aplikacije, poput mogućnosti blokiranja korisničkog sučelja ako se blokira glavna dretva, Osim *AsyncTask*, možemo koristiti i *Hadler* ili *RyJava* s *IO* planerima ili pak neku sasvim drugu opciju koja akcije izvršavanja upita stavlja na sporedne dretve.

Zaključno je bitno sagledati sljedeće kod korištenja ove baze podataka. Osnovna namjera ove baze je lokalno spremanje podataka pa je prema tome ovo jako korisno za tipove aplikacija koje imaju potrebu za tim da određeni podaci budu dostupni korisniku u slučaju prekinutog pristupa mreži. Ova baza podataka je također dobrodošla u situacijama kod aplikacija koje obrađuju netrivijalne količine strukturiranih podataka gdje postoji korist u lokalnom spremanju istih.

Dakle, ovo je do sada već dobro poznata baza podataka s kojom smo se bavili i tijekom studija, a s obzirom na to da popratni praktični projekt nema neku veliku potrebu za lokalnim spremanjem podataka, ovaj rad će se po pitanju *SQL* baza podataka u razvoju Android aplikacija u Kotlinu zaustaviti ovdje, a u nastavku slijedi analiza *NoSQL* baza podataka koje su danas vrlo atraktivne tehnologije u svijetu mobilnog razvoja.

5.2. MongoDb vs Firebase

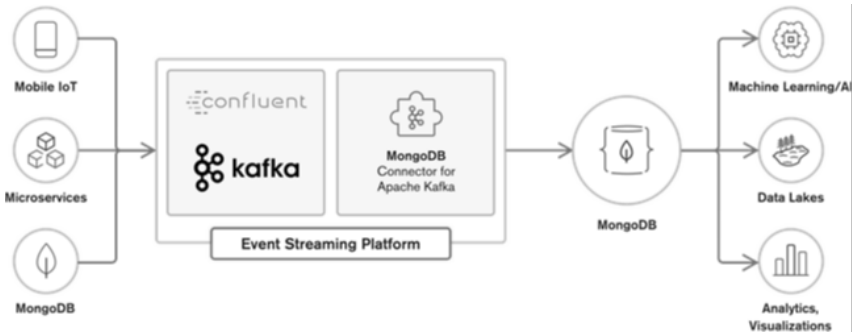
NoSQL baza podataka su sve učestalije u Android razvoju, a neki od najznačajnijih razloga su dostupnost podataka u realnom vremenu te mogućnost baratanja s velikim količinama istih. Dvije najpoznatije NoSQL tehnologije u mobilnom razvoju su MongoDB i Firebase, a sljedeći graf prikazuje porast njihovog korištenja.



Slika 18: Porast korištenja, [15]

5.2.1. MongoDb

MongoDB je NoSQL baza podataka koja je potpuno besplatna za korištenje, a kreirana je te ju održava tvrtka MongoDB Inc. To je dokumentna baza podataka koja pruža kompletni ekosistem usluga, a glavne osobine su joj osobine skalabilnost i fleksibilnost te je vrlo laka za kreiranje upita, kompleksnih upita te indeksiranje. Sljedeća slika prikazuje MongoDB arhitekturu gdje je vidljivo što sve pruža ovaj ekosustav[11].



Slika 19: MongoDD arhitektura, [11]

Dakle, neke od funkcionalnosti MongoDB tehnologije su analitika nad podacima, spremanje podataka u oblaku, jednostavna arhitektura, podatkovni izvještaji u realnom vremenu, a i tehnologija je dokumentno bazirana i prilagodljiva. Jezici koji podržavaju ovu tehnologiju s

puno brojniji od onih koji podržavaju Firebase, ima ih čak 27, a neki najpoznatiji su: C, C#, C++, Erlang, Go, Java, JavaScript, Matlab, Perl, PHP, Python, Ruby, Rust i Scala[11].

Prednosti korištenja ove tehnologije su moćne mogućnosti skaliranja, fleksibilnost, prikazivanje podataka u JSON-u i BSON-u, detaljna dokumentacija, besplatno korištenja, ali i *serverless* sustav ako se odabere usluga za plaćanje, visoko siguran. Neki nedostaci su slučajevi koruptiranih podataka, curenja podataka te gubljenja podataka, nije ACID kompatibilan, nisu podržane pohranjene procedure te indeksiranje[11].

MongoDB tehnologije je dobro koristiti kada želimo visoku skalabilnost i predmemoriranje, brzo logiranje i analitiku u realnom vremenu te kada trebamo opsežno enterprise upravljanje podacima. No, kada želimo izgraditi vrlo detaljno dizajniran sustava, sustav s puno transakcija te usklađenost s ACID pravilima, onda MongoDB nije dobar izbor[11].

5.2.2. Praktični primjer korištenja MongoDB baze podataka

U ovom poglavlju slijedi praktični primjer korištenja MongoDB baze podatak koji će se izvesti nad entitetom korisnika. Za potrebe primjera napravljena je baza s jednim entitetom, odnosno u ovom slučaju model s jednim dokumentom koji je zapravo JSON zapis, a prikazan je u nastavku.

```
{
  "id": ,
  "firstName": "",
  "lastName": "",
  "email": ""
}
```

Programski kod 76: JSON dokument *User*

Prije rada s bazom podataka potrebno je u application.properties datoteku dodati poslužitelja, port, naziv baze te naziv korisnika i njegovu lozinku, a to je prikazano u sljedećem primjeru.

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=diplomski
spring.data.mongodb.username=admin
spring.data.mongodb.password=admin
```

Programski kod 77: application.properties datoteka sa MongoDB postavkama

Za potrebe primjera korištena je SpringBoot biblioteka za rad s MongoDB bazom podataka, odnosno točnije sučelje *MongoRepository* koje sadrži sve osnovne CRUD metode. Najprije je kreirana takozvana POJO klasa za entitet korisnika, a kod je prikazan u sljedećem primjeru. Vidljivo je da se koristi anotacija *@Document* kako bi se naznačilo o kojem se dokumentu u bazi radi.

```
@Document("user")
data class User(

    @Id
    @JsonIgnore
    var id: String?,

    var firstName:String,
    var lastName:String,
    var email:String
    )
```

Programski kod 78: Primjer *@Document* klase za entitet *User*

Dalje je potrebno kreirati repozitorij sučelje, no kako se za potrebe primjera koristi SpringBoot biblioteka, ovo sučelje će biti prazno jer se preko sučelja koje proširuje može pristupiti svim osnovnim CRUD metodama. Nema metoda koje izlaze izvan okvira osnovnih CRUD operacija koje bi u tom slučaju bile pisane u ovoj klasi. U nastavku je prikazan kod sučelja.

```
interface UserRepository:MongoRepository<User,String>{}
```

Programski kod 79: Primjer *Repository* sučelja za entitet *User*

Nakon repozitorij sučelja potrebno je napraviti servisnu klasu koja će komunicirati s metodama iz repozitorija. Servisna klasa ima napisane metode za osnovne CRUD operacije, a to je vidljivo u sljedećem primjeru.

```
@Service
class UserService(private var repository: UserRepository) {

    fun save(user: User): User {
      user.id = UUID.randomUUID().toString()
      user.fistName = "Ana"
      user.lastName = "Sunjic"
      user.email = "anasunjic@gmail.com"
      try{
        return repository.save(user)
      } catch (e:Exception){
        throw UserServiceExceptions("Save error")
      }
    }

    fun getAll(): List<User>{
        try{
            return repository.findAll()
```

```
        } catch (e:Exception){
            throw UserServiceExceptions("Get all error")
        }
    }

    fun update(updatedUser: User, id: String): User {
            val existingUser  = repository.findById(id).orElseThrow {
                Exception("User wit this id does not exist.")
            }
            existingUser.firstName = updatedUser.firstName
            existingUser.lastName = updatedUser.lastName
            existingUser.email = updatedUser.email
            try{
                return repository.save(updatedUser)
            } catch (e:Exception){
                throw UserServiceExceptions("Update error")
            }
        }

        override fun deleteById(id: String) {
            return repository.deleteById(id)
        }

        override fun deleteAll(users: List<User>) {
            return repository.deleteAll(users)
        }
}
```

Programski kod 80: Primjer servisne klase za entitet *User*

Nakon kreirane servisne klase preostaje još kreirati pripadnu kontroler klasu kako bi se kreirane CRUD metode mogle testirati preko Postman alata koristeći endpoint-ove, ali isto tako kako bi bilo koja aplikacija mogla komunicirati s ovim malim servisom. U nastavku je prikazan kod te klase.

```
@RestController
@RequestMapping("/user")
class UserController(private val userService: UserService){

    @PostMapping
    fun save(@RequestBody @Valid user: User):ResponseEntity<User> {
        try {
            return ResponseEntity.ok(UserService.save(user))
        }catch (e: UserControllerExceptions){
```

```
            throw UserControllerExceptions("Error")
        }catch (e:Exception){
            throw UserControllerExceptions("Error")
        }
    }

    @GetMapping("/users")
    fun getAll():ResponseEntity<List<User>> {
        try{
            return ResponseEntity.ok(UserService.getAll())
        } catch (e: UserControllerExceptions){
            throw UserControllerExceptions("Error")
        }catch (e:Exception){
            throw UserControllerExceptions("Error")
        }
    }

    @PutMapping("/{id}")
    fun update(@RequestBody updatedUser : User,
            @PathVariable("id") id:String ):ResponseEntity<User> {
        try {
            return ResponseEntity.ok(userService.update(updatedUser,id))
        }catch (e: UserControllerExceptions){
            throw UserControllerExceptions("Error")
        }catch (e:Exception){
            throw UserControllerExceptions("Error")
        }
    }

    @DeleteMapping
    fun deleteAll(users:List<User>):ResponseEntity<Unit>
            =  ResponseEntity.ok(userService.deleteAll(users))

    @DeleteMapping("/{id}")
    fun delete(@PathVariable("id") id: String):ResponseEntity<Unit>
            = ResponseEntity.ok(userService.deleteById(id))

}
```

Programski kod 81: Primjer kontroler klase za entitet *User*

Na samom kraju još nedostaje *SpringApplication.kt* klasa. Prikazana je u primjeru broj 80 u nastavku.

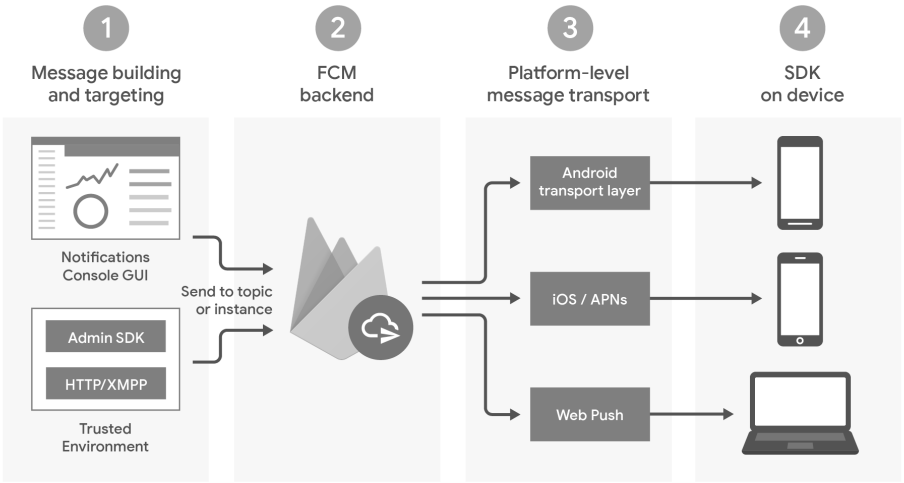
```
@SpringBootApplication
public class MongoDbExampleApp {

    public static void main(String... args) {
        SpringApplication.run(MongoDbExampleApp.class, args);
    }
}
```

Programski kod 82: Primjer *SpringApplication.kt* klase

Nakon praktičnog primjera korištenja MongoDB baze podataka za osnovne CRUD operacije slijedi potpoglavlje koje će teorijski opisati Firebase tehnologiju.

5.2.3. Firebase

Firebase je tehnologija koju je kreirao Google 2012. godine. Poznato je da se koristi kao baza podataka, no sadrži dodatne funkcionalnosti. Može se koristiti u web i mobilnom razvoju. Sadrži dvije osnovne funkcionalnosti, a to su *real-time* bazu podataka, koja pruža mogućnost korištenja podataka u realnom vremenu te servis *Cloud Firestone* koji je također baza podataka no funkcionira tako da vraća podatke kada se desi promjena nad istim. Uz to Firebase sadrži još nekoliko funkcionalnosti koje su prikazane na sljedećoj slici[10].



Slika 20: Firebase arhitektura, [10]

Firebase je podržan u 5 jezika, a to su: Java, JavaScript, Objective-C, C++, Swift. Neke prednosti Firebase tehnologije su: trenutno ažuriranje podataka bez osvježavanja, sinkronizacija na višestrukim uređajima, *push* notifikacije, hosting, plaćanje usluge po ulogu korištenja, stog događaja u oblaku, upravljanje ogromnim količinama podataka itd. Osim toga, neki nedostaci ove tehnologije bi bili otežana migracija podataka, nisu podržani relacijski upiti, kompleksni sigurnosni protokoli, postoji samo plaćena verzija za profesionalno korištenje i naravno serveri na kojima su hostani podaci su u vanjskom vlasništvu odnosno pripadaju tvrtki Google[10].

Zaključno ovo tehnologiju je najbolje koristiti kada trebamo *real-time* podatke, kada je potrebno razviti aplikaciju u kratkom vremenskom roku, kada se u budućnosti planira skalirati aplikacija, u slučajevima kada se razvijaju aplikacije iz domene socijalnih mreža, igrice ili aplikacije za izmjenu poruka. Također, krivulja učenja je vrlo mala te je *API* vrlo intuitivan, a i izuzetno je pogodna za situacije kada trebamo pravo-vremenu sinkronizaciju između verzije aplikacija za web preglednik i recimo mobilnu aplikaciju[10].

Postoje situacije u kojima nije preporučljivo koristiti ovu tehnologiju, a to su kod razvoja iOS aplikacija jer je nedavno podrška za taj operacijski sustav ukinuta, kada želimo jeftiniji opciju skladištenja te kada želimo imati vlasništvo nad podacima[10]. Nakon opisane Firebase tehnologije slijedi praktični primjer korištenja ove baze podataka.

5.2.4. Praktični primjer korištenja Firebase baze podataka

U ovom poglavlju slijedi praktični primjer korištenja Firebase baze podatak koji će se izvesti nad entitetom korisnika. Za potrebe primjera napravljena je baza s jednim entitetom, odnosno u ovom slučaju model s jednim dokumentom koji je zapravo JSON zapis jednak onome koji je korišten u MongoDB bazi podataka, a prikazan je u nastavku.

```
{
  "id": ,  {
      "firstName": "",
      "lastName": "",
      "email": ""
  }
}
```

Programski kod 83: JSON dokument *User* (Firebase)

Postavljanje Firebase konekcije je vrlo jednostavno ako se koristi Android Studio koji ima ugrađenu Firebase podršku. Nakon nekoliko klikova aplikacija se automatski poveže s postojećom odabranom bazom podataka u Firebase oblaku te time nije potrebno dodavati ručno nikakve postavke niti pisati kod za spajanje na bazu. Baza je ovime automatski dostupna kao i Firebase biblioteka. Odmah nakon spajanja aplikacije s Firebase oblakom mogu se kreirati potrebne Kotlin klase, a prva kreirana klasa je POJO klasa korisnika koja je jednaka kao i u primjeru za MongoDB, programski kod slijedi u nastavku.

```
class User : Serializable {

    var id: String? = ""
    var firstName: String? = ""
    var lastName: String? = ""

    constructor() {
```

```
    }
}
```

Programski kod 84: Primjer entitet klase *User* (Firebase)

Dalje je potrebno kreirati repozitorij klasu s osnovnim CRUD metodama, a u nastavku je prikazan programskog koda za taj repozitorij.

```
class UserRepository() {

    private lateinit var database: DatabaseReference
    database = Firebase.database.reference

    fun save(user: User) {
        val docRef = database.collection(users)
        docRef.add(user.toMap()).addOnSuccessListener {
            Log.d("Save success")
        }.addOnFailureListener {
            throw UserRepositoryExceptions("Error")
        }
    }

    fun update(user: User): User {
      val docRef = db.collection(users)
        docRef.document(user.id!!).update(user.toMap())
            .addOnSuccessListener {
            Log.d("Get all success")
        }.addOnFailureListener {
            throw UUserRepositoryExceptions("Error")
        }
    }

    fun deleteById(id: String) {
      val docRef = db.collection(products)
        docRef.document(id).delete().addOnSuccessListener {
            Log.d("Get all success")
        }.addOnFailureListener {
            throw UUserRepositoryExceptions("Error")
        }
    }
}
```

Programski kod 85: Primjer *Repository* klase za entitet *User* (Firebase)

Nakon kreirane *Repository* klase slijedi servisna klasa koja je prikazana u primjeru programskog koda 83 koji slijedi u nastavku.

```
class UserService(private var repository: UserRepository) {

    fun save(user: User): User {
      user.id = UUID.randomUUID().toString()
      user.fistName = "Ana"
      user.lastName = "Sunjic"
      user.email = "anasunjic@gmail.com"
      try{
        repository.save(user)
      } catch (e:Exception){
        throw UserServiceExceptions("Save error")
      }
    }

    fun update(updatedUser: User, id: String): User {
        try{
            repository.save(updatedUser)
        } catch (e:Exception){
            throw UserServiceExceptions("Update error")
        }
    }

    override fun deleteById(id: String) {
        repository.deleteById(id)
    }
}
```

Programski kod 86: Primjer *Service* klase za entitet *User* (Firebase)

Na samom kraju programski kod 85 prikazat će kontroler klasu za ovaj primjer rada s Firebase bazom podataka. Ovo će ujedno biti i posljednja prikazana klasa s obzirom na to da je ostatak jednak prethodnom primjeru za MongoDB bazu podataka.

```
@RestController
@RequestMapping("/user")
class UserController(private val userService: UserService){

    @PostMapping
    fun save(@RequestBody @Valid user: User):ResponseEntity<User> {
        try {
            return ResponseEntity.ok(UserService.save(user))
        }catch (e: UserControllerExceptions){
            throw UserControllerExceptions("Error")
        }catch (e:Exception){
```

```
            throw UserControllerExceptions("Error")
        }
    }

    @PutMapping("/{id}")
    fun update(@RequestBody updatedUser : User,
            @PathVariable("id") id:String ):ResponseEntity<User> {
        try {
            return ResponseEntity.ok(userService.update(updatedUser,id))
        }catch (e: UserControllerExceptions){
            throw UserControllerExceptions("Error")
        }catch (e:Exception){
            throw UserControllerExceptions("Error")
        }
    }

    @DeleteMapping("/{id}")
    fun delete(@PathVariable("id") id: String):ResponseEntity<Unit>
            = ResponseEntity.ok(userService.deleteById(id))

}
```

Programski kod 75: Primjer *Controller* klase za entitet *User* (Firebase)

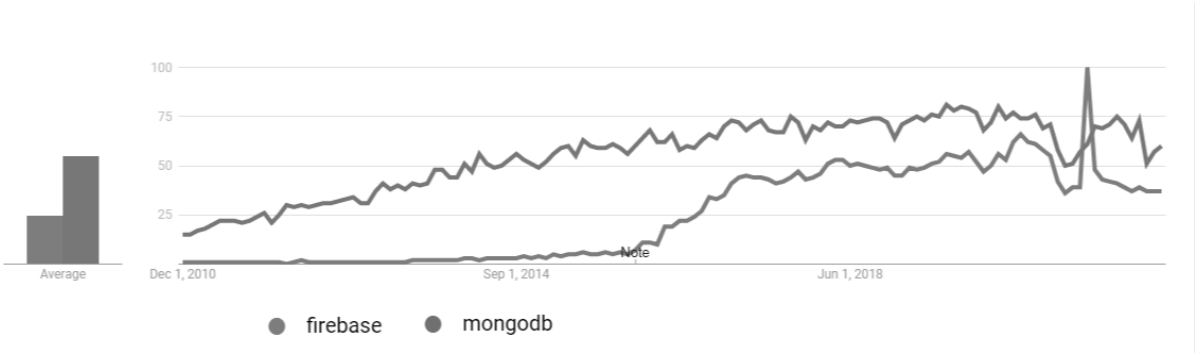
Na kraju ovog programskog primjera kojim je pokazan rad s Firebase bazom podataka slijedi još jedno potpoglavlje gdje će biti provedena komparativna usporedba Firebase i MongoDB baze podataka te će biti donesen zaključak o tome koja će se baza koristiti u praktičnom primjeru ovog rada.

5.2.5. Usporedba

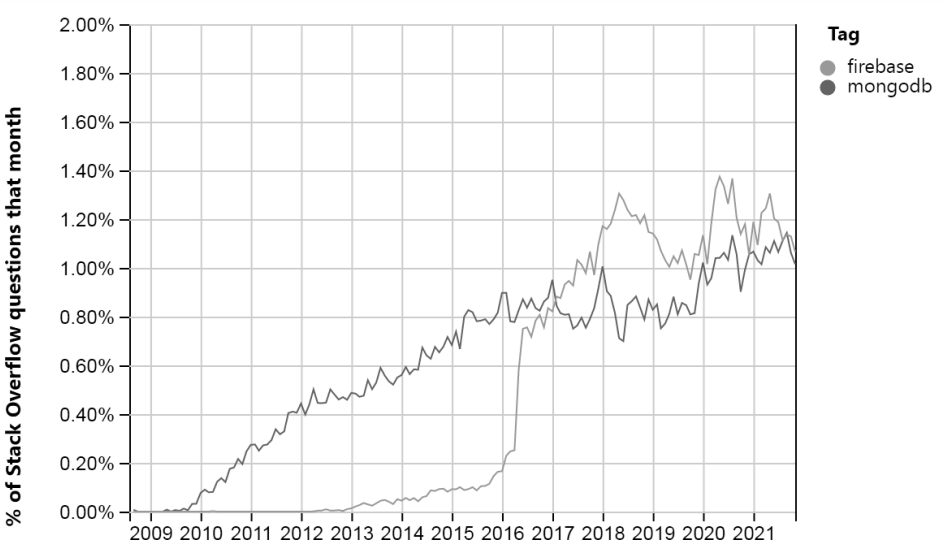
U nastavku će kroz popis osnovnih razlika biti uspoređene tehnologije Firebase i MongoDB uz neke popratne grafove koji će vizualno prikazati neke od sljedećih razlika[11]:

- Mogućnost rada s jako velikim prometom podataka koje nudi MongoDB dok je recimo Firebase tome vrlo inferioran.
- Podrška u oblaku koju Firebase nudi, a MongoDB ne.
- Broj jezika koji ih podržavaju koji je puno brojniji u slučaju MongoDB baza podataka.
- Kategorizacija: Firebase je platforma u oblaku za dijeljenje podataka u realnom vremenu te gdje podaci mogu biti djeljeniji na više uređaja. S druge strane MongoDB je baza podataka, dokumentno orijentirana, dinamična, JSON bazirana gdje mogu biti pohranjeni strukturirani i ne strukturirani podaci te koja nudi skalabilnost, replikaciju podataka te oporavak.

- Pohrana podataka: Firebase je platforma koja je jednostavna za uspostavu, najčešće je korištena za aplikacije namijenjene za kratko korištenje, kada se želi brzi i responzivni razvoj. S druge strane MongoDB je NoSQL baza podataka, ne strukturirana te najčešće korištena za aplikacije široke upotrebe
- Slučajevi primjene koji su u slučaju Firebase tehnologije aplikacije malih razmjera, a za MongoDB aplikacije većih razmjera koje zahtijevaju opsežnu bazu podataka.
- Cijena tehnologija: Firebase nudi *free trial* verziju, te nekoliko opcija s mjesečnom uplatom koje se kreču od 5 dollara mjesečno do 150 dollara mjesečno. S druge strane MongoDB je besplatan no nudi neke opcije koje se plaćaju, kao recimo *serverless* opcija.
- Performanse - MongoDB je stabilnija tehnologija, a u usporedbi s Firebase-om i fleksibilnija jer se podaci ne nalaze na tuđem serveru koji može biti odsječen. MongoDB također nudi veće prednosti po pitanju performanci, kao i stabilnije servere. popularnost MongoDB baza podataka, a drugi trenutnu ujednačenost obje tehnologije.
- Popularnost na tržištu predstavljena s dva grafa. prvi prikazuje popularnost putem *Google trends* statistike, a drugi putem *Stack Overflow Trends* statistike. Prvi graf prikazuje veću popularnost MongoDb-a, no drugi graf ukazuje na to da se vodi više rasprava, a samim time i da postoji veća zajednica na strani Firebase tehnologije.



Slika 21: *Google trends* - MongoDb vs Firebase, [Autorski rad]



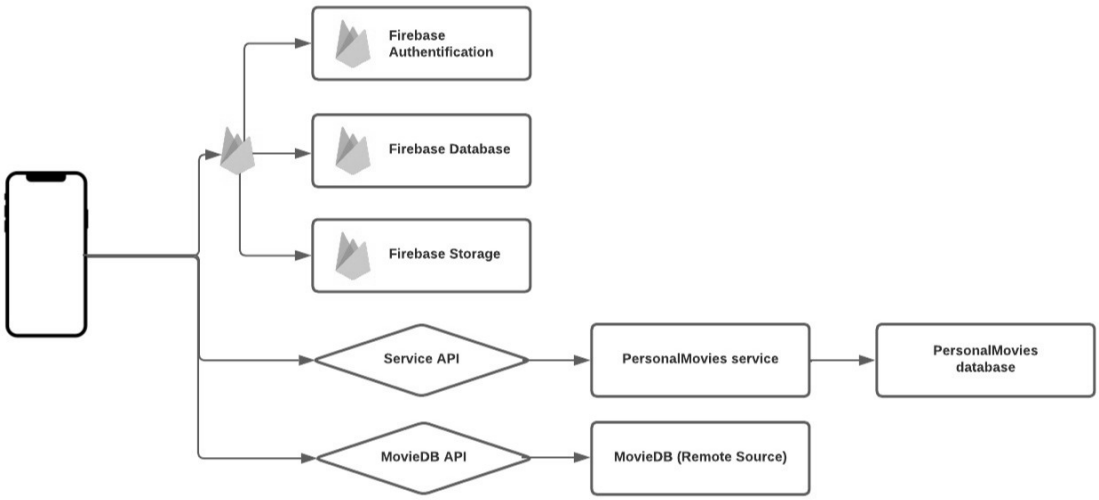
Slika 22: *Stack Overflow Trends* - MongoDb vs Firebase, [Autorski rad]

Zaključno iz gledišta ovog rada jasno je da je najbolja opcija za praktični primjer aplikacije Firebase tehnologija. Osim što je praktični dio zapravo mala aplikacija koja nije namijenjena za šire korištenje i ne barata velikom količinom podataka, također i opcija dostupnosti podataka u realnom vremenu te dodatne funkcionalnosti koje platforma nudi, kao što su autentifikacija bi također bili presudni faktor za odabir Firebase platforma prilikom izrade popratne aplikacije.

Osim zaključka oko odabira tehnologije za praktični dio rada, ovo poglavlje je rezultiralo i ispunjavanjem jednog od samih ciljeva rada a to je proučiti SQL i NoSQL tehnologije baza podataka za izradu Android aplikacija. Ovdje završava teorijski dio rada te ostaje još zadnje poglavlje u kojem će biti opisan praktični primjer koji je izrađen uz ovaj rad.

6. Praktična izrada apliakcije

U ovom poglavlju bit će opisane funkcionalnosti praktičnog djela diplomskog rada te će se prikazati Mockup zamišljene aplikacije, a prije samog popisa funkcionalnosti slijedi dijagram arhitekture aplikacije na slici broj 23. Vidljivo je da mobilna aplikacija komunicira s Firebase uslugom preko koje koristi usluge autentifikacije, baze u oblaku te pohranu podataka. Nadalje vidimo da aplikacija ima API za komunikaciju s dva vanjska servisa. Jedan od njih je Movie Database service koji je dostupan na internetu, a drugi servis je kreiran za svrhu ovoga rada.



Slika 23: Arhitektura aplikacije, [Autorski rad]

Nadalje slijedi popis funkcionalnosti aplikacije koje su podjeljenje u tri grupe 4 grupe. Prva grupa funkcionalnosti je vezana za autentifikaciju korisnika, druga grupa za korisnički profil odnosno njegovo upravljanje, treća grupa za funkcionalni vezane s vanjskim servisom dostupnim na internetu s kojim aplikacija komunicira, te četvrta grupa funkcionalnosti koja se odnosi na komunikaciju s kreiranim mikroservisom kao jednim djelom praktičnog rada.

- Autentifikacija - funkcionalnost koja je realizirana korištenjem Firebase autentifikacije te je omogućena putem maila te putem gmail servisa.
  - Prijava korisnika putem e-mail-a
  - Registracija korisnika e-mail-a
  - Upravljanje sesijama
- Korisnički profil - funkcionalnost koja omogućuje upravljanje profilom korisnika tako da je moguće vidjeti te ažurirati podatke koji se nalaze u Firebase bazi podataka.
  - Pregled podataka
  - Ažuriranje podataka

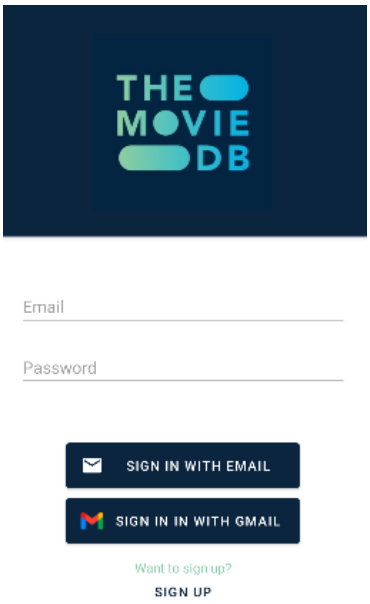
- Upravljanje filmovima - funkcionalnost koja upravlja dohvatom, prikazom te pretraživanjem filmova s udaljenog izvora odnosno putem API-a. Podatke uzima s MovieDB online web izvora filmova.
  - Pregled filmova
  - Pretraga filmova
  - Detalji filmova
- Personalizirano upravljanje filmovima - funkcionalnost koja omogućuje korisniku kreiranje šersonaliziranih lista filmova. Ova funkcionalnost kreirana je kao mikroservis s kojim komunicira mobilna aplikacija putem API-a.
  - Kreiranje listi filmova
  - Brisanje listi filmova
  - Pregled listi
  - Dodavanje filmova na listu
  - Brisanje filmova sa liste

Nakon opisanih funkcionalnosti slijedi potpoglavlje gdje će biti ukratko opisan proces kreiranja mockup-a za potrebe praktičnog rada te će prikazati neki od ekrana.

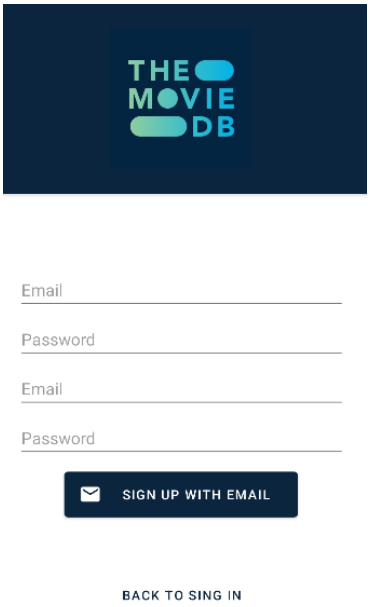
6.1. *Mockup*

Mockup za izradu aplikacije rađen je u alatu Figma u svrhu predodžbe potrebnih ekrana te entiteta unutar aplikacije. Aplikacija je najprije osmišljena na papiru nekakvom skicom ekrana te su nakon toga kreirani detaljniji prikazi ključnih ekrana u aplikaciji Figma. Nisu kreirani svi ekrani obzirom da je ovo bilo dovoljno da se odrede entiteti. Ovaj proces je rezultirao dizajnom ekrana na kojima se može identificirati svrha te mogućnosti i funkcionalnosti koje pružaju. Osim toga kreiranjem mockup bilo je moguće unaprijed sagledati razinu kompleksnosti izrade korisničkih sučelja. Na ekranima je prikazan višak elemenata u odnosu na implementaciju. U nastavku slijedi nekoliko slika tih kreiranih ekrana.

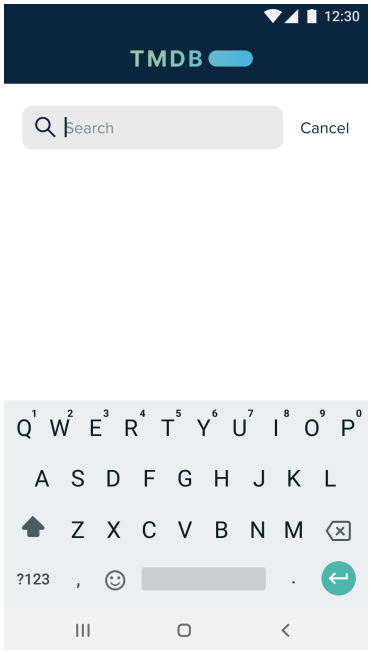
Slika 24: Početni ekran, [Autorski rad]



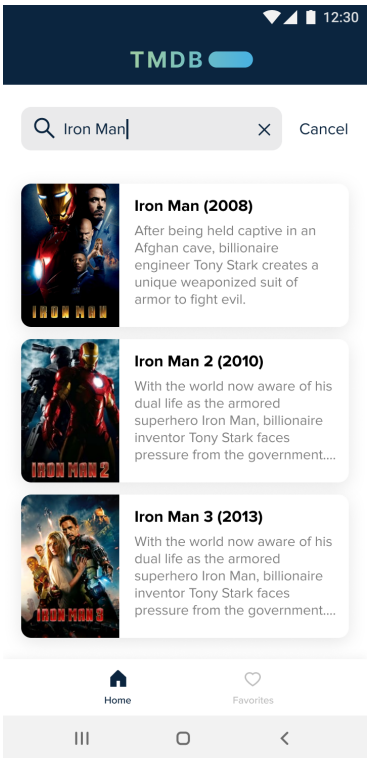
Slika 25: Ekran prijave, [Autorski rad]



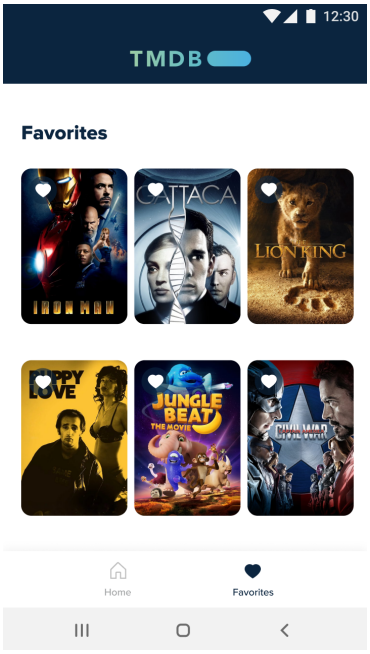
Slika 26: Ekran registracije, [Autorski rad]



Slika 27: Ekran pretrage, [Autorski rad]



Slika 28: Rezulati pretrage, [Autorski rad]

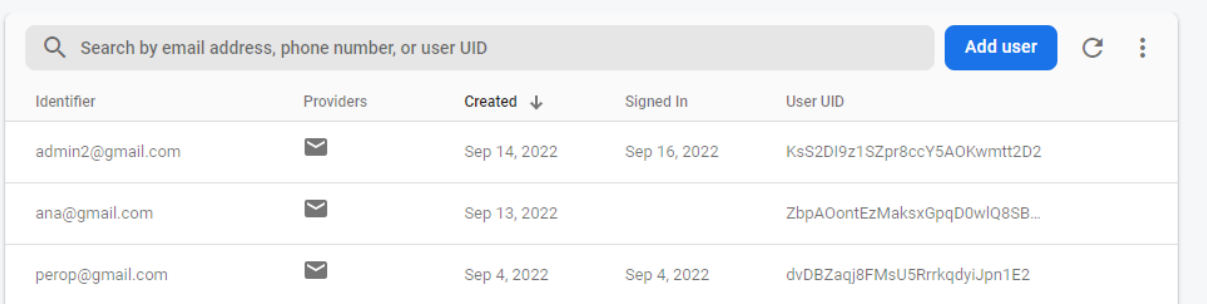


Slika 29: Prikaz liste filmova, [Autorski rad]

Nakon poglavlja u kojem je opisan Mockup slijedi poglavlje gdje će biti opisan slijed implementacije te ključni dijelovi koda.

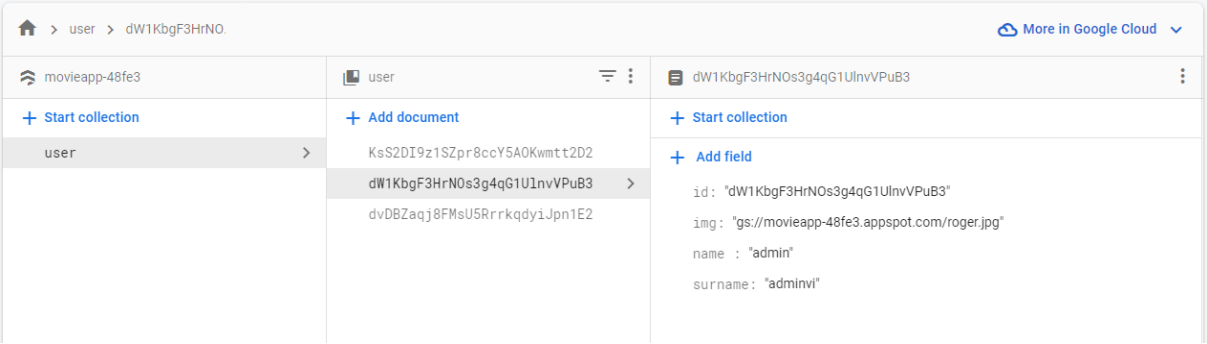
6.2. *Arhitektura rada s podacima u Firebase bazi*

U ovom poglavlju slijedi opis implementacije aplikacije, za svaku skupinu funkcionalnosti bit će opisano kako je implementirano te će biti priloženi neki od primjera. Najprije će biti opisana autentifikacija te izmjena podataka na profilu, a s obzirom na to da to radi putem Firebase alata, slijedi prikaz spremišta s autentifikacijama te sama baza.



Slika 30: Autentificirani korisnici, [Autorski rad]

Nakon slike s autentificiranim korisnicima vidljiva je slika s bazom. Može se vidjeti da sadrži samo entitet korisnika s pripadnim atributima.



Slika 31: Firebase baza podataka, [Autorski rad]

Prva skupina funkcionalnosti jest autentifikacija, te je to ujedno i prvo rađeno. Autentifikacija se vrši putem ugrađene podrške za autentifikaciju unutar Firebase alata. Kreiran je logiranje te registriranje korisnika. Prema ciljevima rada poštovana je MVVM arhitektura te je time kreirana pripadna *Activity* te *ViewModel* klasa te uz njih popratne klase za autentifikaciju odgovora te *Factory* za povezivanje viewmodel-a s *Activity* klasom. U nastavku slijedi prikaz *Factory* klase koji se koristi kontinuirano kroz aplikaciju, no kod će biti samo ovdje prikazan.

```
@Suppress("UNCHECKED_CAST")
class SignupViewModelFactory : ViewModelProvider.NewInstanceFactory() {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(SignupViewModel::class.java)) {
            return SignupViewModel(
                signupRepository = AuthRepository(
```

```
                    dataSource = FirebaseAuthDataSource()
                )
            ) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Programski kod 88: Primjer *Factory* klase

U primjeru 88 vidljivo je kako se prosljeđuju repozitorij klase i izvor podataka. Nadalje slijedi prikaz djela repozitorij klase preko koje se komunicira s izvorom podataka.

```
class AuthRepository( val dataSource: FirebaseAuthDataSource) {

     fun login(email: String, password: String): Completable {
        val result = dataSource.login(email, password)
        return result
    }

    fun register(email: String, password: String, name: String,
    surname: String): Completable {

        val result = dataSource.register(email, password, name, surname)
        return result
    }
}
```

Programski kod 89: Primjer *Repository* klase

Nakon prikaza repository klase gdje su priložene dvije metode, slijedi prikaz izvora podataka za iste metode u kodnom primjeru broj 90.

```
fun login(email: String, password: String) = Completable.create {
emitter ->
firebaseAuth.signInWithEmailAndPassword(email, password).addOnCompleteListener {
            if (!emitter.isDisposed) {
                if (it.isSuccessful)
                    emitter.onComplete()
                else
                    emitter.onError(it.exception!!)
            }
        }
    }
```

```
    fun register(email: String, password:
    String, name: String, surname: String) = Completable.create { emitter ->
        firebaseAuth.createUserWithEmailAndPassword
(email,password).addOnCompleteListener {
            if (!emitter.isDisposed) {
                if (it.isSuccessful){
                    val createdUserId = firebaseAuth.currentUser!!.uid
                    var user = User(createdUserId, name, surname, "")
                    firebaseDb.collection("user").document(createdUserId)
                        .set(user)
                        .addOnSuccessListener {
                            Log.d(TAG, "Dodatni podaci o useru
                            uspjesno spremljeni!")
                            emitter.onComplete()
                        }
                        .addOnFailureListener {
                                e -> Log.w(TAG, "Greška kod spemanja
                                dodatnih podataka o useru.", e)
                                emitter.onError(it.exception!!)
                        }
                } else
                    emitter.onError(it.exception!!)
            }
        }
    }
```

Programski kod 90: Primjer klase izvora podataka

Nakon autentifikacije kreirana je funkcionalnost uređivanja profila gdje je omogućeno izmijeniti informacije o imenu i prezimenu te korisnik ima uvid u svoje podatke. S obzirom na to da se ovi podaci nalaze isto u Firebase bazi podataka slijed klasa kao i sama implementacija prati prethodno opisanu metodologiju. Kako u prethodnim metodama nema dohvata podataka iz baze u svrhu prikaza na ekranu u nastavku slijedi primjer programskog koda 91 koji pokriva taj dio.

```
val createdUserId = firebaseAuth.currentUser!!.uid
        val docRef = firebaseDb.collection("user").document(createdUserId)

        docRef.get()
            .addOnSuccessListener { document ->
                if (document != null) {
                    namelab.text = document.data?.get("name").toString()
                    surnamelab.text = document.data?.get("surname").toString()
                } else {
```

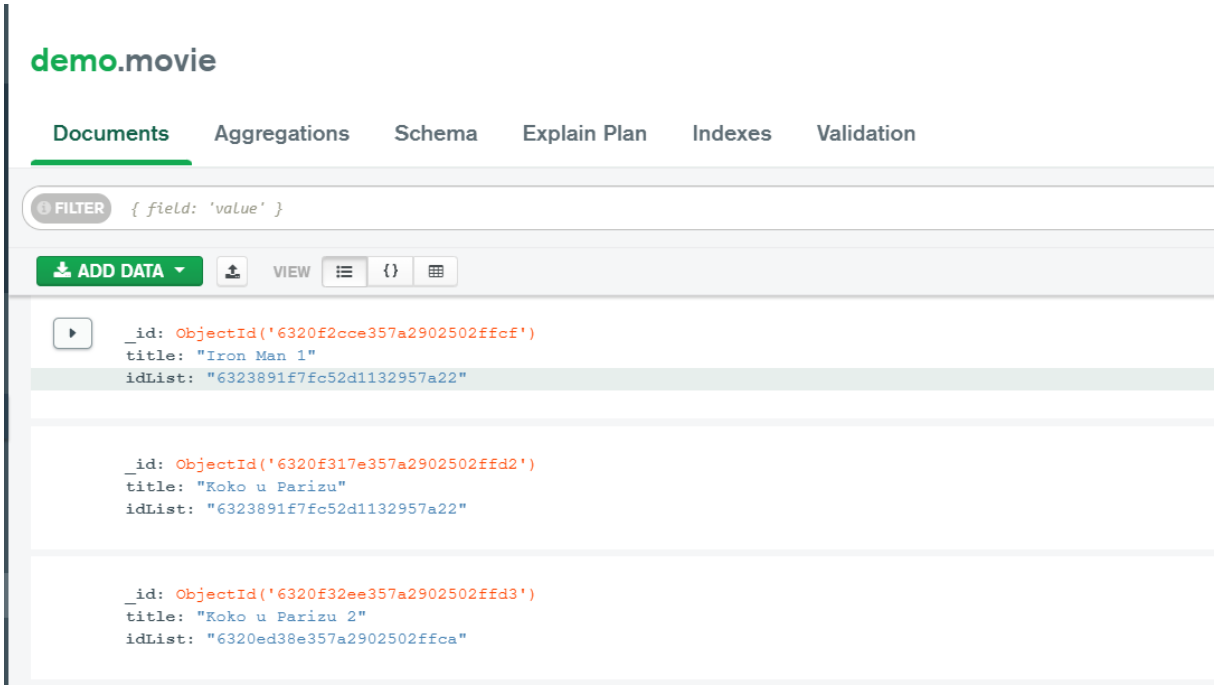
```
                Log.d(ContentValues.TAG, "No such document")
            }
        }
        .addOnFailureListener { exception ->
            Log.d(ContentValues.TAG, "get failed with ", exception)
        }
```

Programski kod 91: Primjer dohvaćanja podataka iz baze

Ovim dvjema funkcionalnostima ostvaren je cilj rada vezan za arhitekturu time što je prakticirana implementacija MVVM arhitekture kod autentifikacije, ali i korištena je tehnologija za baze podataka kojom se rad bavio a to je Firebase. U sljedećem poglavlju slijedi opis implementacije dijela aplikacije koji radi sa servisima.

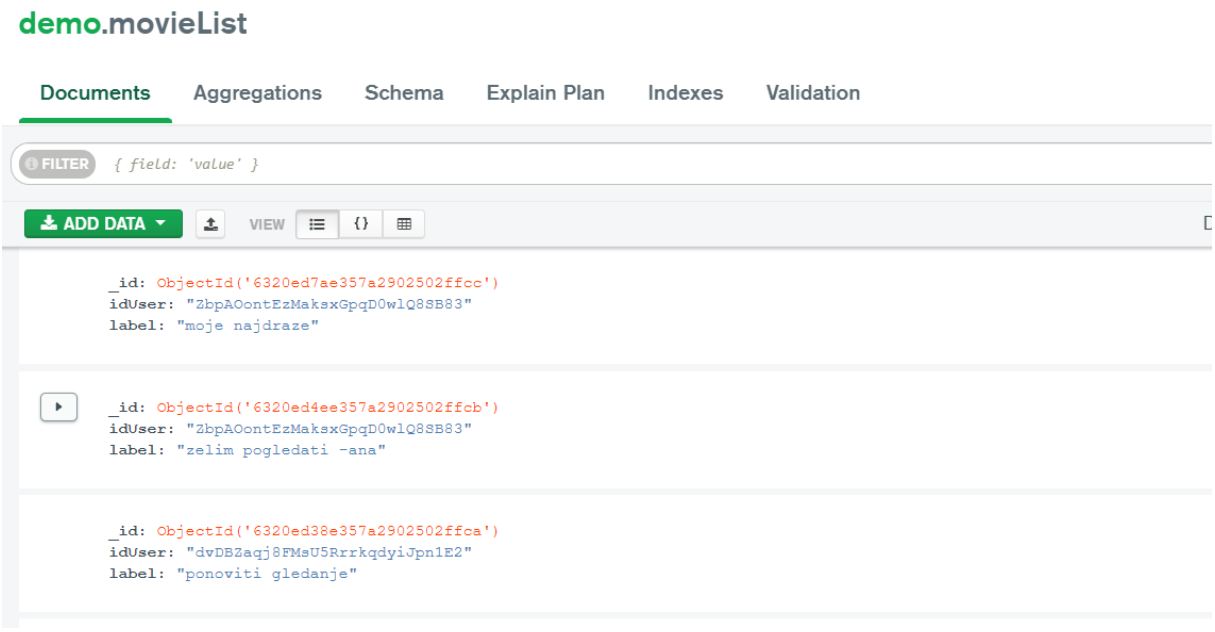
6.3. Arhitektura rada sa servisima

Ovo poglavlje bavit će se implementacijom dijelova aplikacije koji rade u komunikaciji s kreiranim mikroservisom te vanjskim servisom. Najprije je kreiran mikroservis u Kotlinu korištenjem SpringView biblioteke. Mikroservis komunicira sa svojom odvojenom bazom te vraća aplikaciji podatke vezane za funkcionalnost personaliziranih listi. Baza je kreirana korištenjem MongoDB tehnologije, a u nastavku slijedi prikaz baze. Prvo je u nastavku prikazan dokument u bazi za filmove gdje se vidi koje elemente ima.



Slika 32: MongoDBaza dokument filma, [Autorski rad]

Nakon toga je prikazana slika s dokumentom liste filmova gdje se isto može vidjeti koje ima entitet, slika slijedi u nastavku.



Slika 33: MongoDBaza dokumnet liste filmova, [Autorski rad]

Mikroservis se sastoji od data klasa koje oponašaju objekte za odgovor klijentu te objekte za spremanje u bazu, nastavku slijedi primjer jednog takvog objekta za film. Kreiranje su takve klase za objekt filma, te liste filma. Ovdje je implementiran prikaz listi i filmova, dodavanje istih te brisanje.

```
data class MovieListResponse (

        val id: String,

        val idUser: String,

        val label: String
        )
```

Programski kod 92: Primjer objekta za odgovor klijentu

Nakon toga su kreirane kontroler klase putem kojih vanjski klijent može kontaktirati mikroservis. Kreirana je kontroler klasa za dohvaćanje Filmova i Listi koje sadrže osnovne CRUD metode, a u nastavku je pokazan jedan kontroler za filmove.

```
@RestController
@RequestMapping("/movies")
class MovieController (
    private val movieRepository: MovieRepository
```

```
) {

    @GetMapping("/get")
    fun getAllMoviesListUser(@RequestParam("id") id: String):
ResponseEntity<List<Movie>> {
        val movies = movieRepository.findAll()
        var moviesSpecial : ArrayList<Movie> = arrayListOf()
        for (item in movies) {
            if (item.idList.equals(id)) {
                moviesSpecial.add(item)
            }
        }
        return ResponseEntity.ok(moviesSpecial)
    }

    @DeleteMapping("/delete")
    fun delete(@RequestParam("id") id: String):
ResponseEntity<String> {
        val response = movieRepository.deleteById(id)
        return ResponseEntity.ok(response.toString())
    }

    @PostMapping("/insert")
    fun insert(@RequestParam("title") title: String,
               @RequestParam("idList") idList: String):
ResponseEntity<String> {

        var objectId = ObjectId();
        val response = movieRepository.insert(Movie(objectId, idList, title))
        return ResponseEntity.ok(response.toString())

    }

}
```

Programski kod 93: Primjer kontroler klase

Nakon toga je za Android aplikaciju kreirana klasa izvora podataka koja kontaktira server, te pripadne *Activity*, *ViewModel* te *Adapter* klase kako bi se ispoštovala arhitektura, ali i dohvatili podatke. S obzirom na to da ovdje ima puno koda slijede ključni dijelovi koji su odraz ciljeva rada. Najprije Api pozivi koji su implementirani putem Retrofit biblioteke.

```
data class MovieListResponse (
interface RetroService {
```

```
    @GET("/lists")
    fun getAllLists(@Query("id") id : String): Call<List<ListResponse>>

    @GET("/lists/delete")
    fun deleteList(@Query("idUser") idUser :
String, @Query("label") label : String): Call<String>

    @GET("/lists/insert")
    fun insertList(@Query("idUser") idUser : String,
@Query("label") label : String): Call<String>

    // Movie methods
    @GET("/movies")
    fun getAllMovies(@Query("idUser") idUser :
String, @Query("idList") idList : String): Call<List<MovieResponse>>

    @DELETE("/movies/delete")
    fun deleteMovie(@Query("id") id : String): Call<String>

    @POST("/moviess/insert")
    fun insertMovie(title : String, idList : String): Call<String>
}
```

Programski kod 94: Primjer api metoda

Zatim primjer implementacije jedne od CRUD metoda gdje se vidi rad s *LiveData* mehanizmima što je spomenuto u radu.

```
fun deleteList(idUser : String, label : String) {
        val retroInstance = RetroInstance.getInstance()
.create(RetroService::class.java)
        val call = retroInstance.deleteList(idUser, label)

        call.enqueue(object : Callback<String> {
            override fun onResponse(
                call: Call<String>,
                response: Response<String>
            ) {
                if (response.isSuccessful) {
                    Log.i(ContentValues.TAG, "uspjesno obrisan"
            + (response.body()))
                }
            }
```

```
            override fun onFailure(call: Call<String>, t: Throwable) {
                Log.i(ContentValues.TAG, "neuspjesno obrisan");
            }
        })
    }
```

Programski kod 95: Primjer pozivanja CRUD metode te spremanja podataka u LIveData

Istim principom su implementirane ostale CRUD metode. Najprije se dohvate sa servera te postave u LiveData, a nakon toga pripadni element osluškivanjem preuzima podatke i prikazuje ih na fronti upotrebom adaptera. Dakle, prethodno opisana funkcionalnost se temeljem ciljeva rada dotakla nekih koncepata kao što su: POJO klase, konstruktori, lambda izrazi, LiveData. mikroservisna arhitektura te MVVM uzorak.

Dalje je kreirana i funkcionalnost za upravljanje filmovima. Ova funkcionalnost komunicira s vanjskim API-jem pod imenom https://www.tvmaze.com/, a s njega dohvaća listu filmova za prikaz filmova, zatim dohvaća detalje filmova za prikaz detalja filmova te dohvaća rezultate pretrage filmova.

Kreirana je API klasa jednako kao i za mikroservis te pripadne repozitorij klase koje povezuju s izvorom podataka. za dohvat podataka je isto korištena Retrofit biblioteka, a također je korišten mehanizam LiveData za emitiranje dohvaćenih podataka. Podaci se prikazuju na fronti uz pomoć adaptera. Kako prethodno nije prikazano dohvaćanje podataka iz LiveData bit će prikazano u nastavku. U primjeru dole prikazane su dvije metode jedna koja osluškuje rezultate dohvata, te druga koja podatke prosljeđuje adapteru.

```
    private fun initRecView() {
        viewModel = ViewModelProvider(this)
.get(SearchResultsViewModel::class.java)
        viewModel.getSearchResponseObservable()
.observe(this, Observer<List<MovieItem>> {
            if (it == null) {
Toast.makeText(this@SearchActivity,
"No result fount", Toast.LENGTH_LONG).show()
            }else{
                moviesAdapter.movies = it.toMutableList()
                moviesAdapter.notifyDataSetChanged()
            }
        })
    }

    private fun initViewModel() {
        val recyclerview = findViewById<RecyclerView>(R.id.recyclerView)
        recyclerview.apply {
layoutManager = LinearLayoutManager(
```

```
this@SearchActivity, LinearLayoutManager.HORIZONTAL, false)
            setHasFixedSize(true)
            //addItemDecoration(decorator)
            moviesAdapter = SearchAdapter(this@SearchActivity)
            adapter = moviesAdapter
        }
    }

    override fun onItemClick(movies: MovieItem) {
        globalMovieName = movies.name
    }
```

Programski kod 96: Primjer osluškivanja podataka

Kako također prethodno nije prikazan niti jedan adapter, bit će prikazano ovdje a i vidljiva je implementacija rada s Kotlin kortinama također temeljem ciljeva ovog rada.

```
class MoviesAdapter(val clickListener: MoviesAdapter
.onItemClickListener): RecyclerView.Adapter<MoviesAdapter.MyViewHolder>() {

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

        val moviesLabel : TextView = view.textView
        val movieImage : ImageView = view.imageView

        fun bind(data : MovieItem){
                moviesLabel.text = data.name
                movieImage.load(data.image.original) {
                    crossfade(true)
                    crossfade(1000)
                }

        }
    }

    private val diffCallback = object : DiffUtil.ItemCallback<MovieItem>() {
        override fun areItemsTheSame(oldItem:
MovieItem, newItem: MovieItem): Boolean {
            return oldItem.id == newItem.id
        }

        override fun areContentsTheSame(oldItem: MovieItem,
newItem: MovieItem): Boolean {
            return newItem == oldItem
```

```
        }
    }

    private val differ = AsyncListDiffer(this, diffCallback)
    var movies: List<MovieItem>
        get() = differ.currentList
        set(value) {
            differ.submitList(value)
        }

    override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): MyViewHolder {
        val inflater = LayoutInflater.from(parent.context).inflate(
R.layout.movie_elemnt_list, parent, false)
        return MoviesAdapter.MyViewHolder(inflater)
    }

    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        val currentMovie = movies[position]
        holder.bind(movies[position])

        holder.itemView.setOnClickListener{
            clickListener.onItemClick(movies[position])
        }

    }

    override fun getItemCount() = movies.size

    interface onItemClickListener {
        fun onItemClick(movies : MovieItem)
    }

}
```

Programski kod 97: Primjer adapter klase

Prikazani adapter radi s korutinama putem vanjske biblioteke. Takav način implementacije adaptera primijenjen je na prikazu za koji je potrebno dohvatiti velike količine podataka.

Ovime završava opis implementacije.Obzirom da programski primjer sadrži puno koda nije moguće prikazati sve te su prikaz svega bazirao na ciljevima rada. Dakle, prikazani su dijelovi koda koji su implementirani mehanizmima koji su opisani u radu te koji su koristili neke od opisane funkcionalnosti kotlina.

7. Zaključak

Ovim radom pokriveni su svi unaprijed deno vezani za ciljeve ovog rada. finirani ciljeoglavlje fokusiralo na to da prikaže djelove koda koji su direktvi rada. Najprije su opisane odabrane funkcionalnosti jezika Kotlin s fokusom na napredne koncepte. Nakon toga je uslijedio dio rada koji se bavio arhitekturama android aplikacija što teorijski tako i programskim primjerom, a taj dio je uključivao i arhitekture na strani poslužitelja. Nakon toga, zadnje teorijsko poglavlje pozabavilo se bazama podataka te je prema zadanim ciljevima pokrivena tema SQL i NoSQL baza podataka. Nakon provedenih istraživanja, usporedbi te proučavanja odabrane stručne literature i internet izvora doneseni zaključi o tome kako najbolji odabir arhitekture leži upravo u MVVM uzorku, ali isto tako da je i iz gledišta dugoročne održivosti i ponovne iskoristivosti modula najbolji izbor razvijati mikroservisne aplikacije. Naravno uz bitnu iznimku jako malih aplikacija gdje nema potrebe za takvom arhitekturom.

Praktičnim djelom rada izrađena je mala aplikacija za upravljanje filmovima kojom je pokušano pokriti obrađene koncepte Kotlin jezika, ali i prakticirana je MVVM arhitektura jer je identificirana kao najbolji izbor, te je izrađen jedan mikroservis u sklopu rada. Izrađeni mikroservis potpuno je autonoman i nadogradiv, te kao takav može poslužiti kao arhitekturalni dio i neke druge mobilne, a naravno i web aplikacije.

Na kraju ovog rada se može također zaključiti kako je Kotlin jezik izuzetno atraktivan i zanimljiv, a kako je ideja odabira ova tematike bila dublje upoznavanja s tim jezikom, može se reći i da je motivacija za ovom temom potpuno zadovoljena.

Popis literature

[1] “Kotlin and Android.” Android Developers, https://developer.android.com/kotlin. Dostupno 20.3.2022.

[2] Bruce Eckel, Svetlana Isakova, Atomic Kotlin: Learn the successor to Java today!, 1. izd, SAD: Mindview LLC, 2020.

[3] Nishant Srivastava, Filip Babić, Kotlin Coroutines by Tutorials, 3. izd, SAD: Mindview LLC, 2022.

[4] Alex Forrester, Eran Boudjnah, Alexandru Dumbravan, Jomar Tigcal, Dependency Injection with Dagger and Koin - How to Build Android Apps with Kotlin, 1. izd, Packt, 2021.

[5] “LiveData Overview.” Android Developers, https://developer.android.com/topic/libraries/architecture/livedata. Dostupno 25.4.2022.

[6] “Android Architecture.” GeeksforGeeks, 12 Sept. 2019, https://www.geeksforgeeks.org/android-architecture/. Dostupno 20.3.2022.

[7] Judith Hurwitz, Robin Bloor, Carol Baroudi, and Marcia Kaufman, Service Oriented Architecture for dummies, 3. izd, SAD: Wiley Publishing, Inc., 2010.

[8] Juan Antonio Medina Iglesias, Hands-On Microservices with Kotlin, 1. izd, Psckt, 2018.

[9] Room Database: Getting Started. www.raywenderlich.com, https://www.raywenderlich.com. Dostupno 30.6.2022.

[10] “Firebase Documentation.” Firebase, https://firebase.google.com/docs. Dostupno 30.6.2022.

[11] MongoDB Documentation. https://www.mongodb.com/docs/. Dostupno 18.7.2022.

[12] Shailendra Chauhan, Understanding MVC, MVP and MVVM Design Patters, 23.Aug.2022, https://www.dotnettricks.com/. Dostupno 30.6.2022.

[13] Hiren Dhaduk, MVC vs MVP vs MVVVM for Android Application Development, 29.Jan.2018, https://www.simform.com/. Dostupno 30.6.2022.

[14] Medium, Using Room Database, 2020, https://www.medium.com/. Dostupno 30.7.2022.

[15] Saurabh Barot, Firebase vs MongoDB, 2021, https://aglowiditsolutions.com. Dostupno 30.7.2022.

Popis slika

Popis programskih kodova