SREDNJA ELEKTROTEHNIČKA ŠKOLA
SARAJEVO

MATURSKI RAD

TEMA: ANDROID ARHITEKTURA U KOTLINU

Mentor:

Prof. mr. Amir Karačić dipl. el. ing.

Učenik:

Nejra Čorić

Sarajevo, maj 2023

Sadržaj

Uvod

Projekat Kotlin počeo je 2011. godine kada je kompanija JetBrains započela njegov razvoj. U to vrijeme nije bilo programskih jezika koji obuhvataju funkcionalnosti današnjih modernih jezika poput Swift-a. Jedini sličan programski jezik bio je Scala koja je imala problem dugog kompajliranja koda. 2012. godine projekat postaje otvorenog koda pod Apache 2 licencom. Prva verzija 1.0 izlazi 2016. godine, a najveća prekretnica dogodila se 2017. godine kada je projekat dobio podršku u razvoju i održavanju od strane Google-a.

Kotlin je statički napisan programski jezik koji se pokreće pomoću Java virtualne mašine zbog čega se može kompajlirati u Java byte kod. Osim Jave, može se kompajlirati i u JavaScript kod zbog čega se može koristiti i za razvoj više-platformskih aplikacija. Od Andorid Studio verzije 3.0, Kotlin je ugrađen kao jedna od primarnih funkcionalnosti gdje korisnik može birati između programiranja u Kotlin-u ili Jave, ali je moguće programirati i u oba jezika istovremeno sve dok je kod raspoređen u odgovarajuće klase.

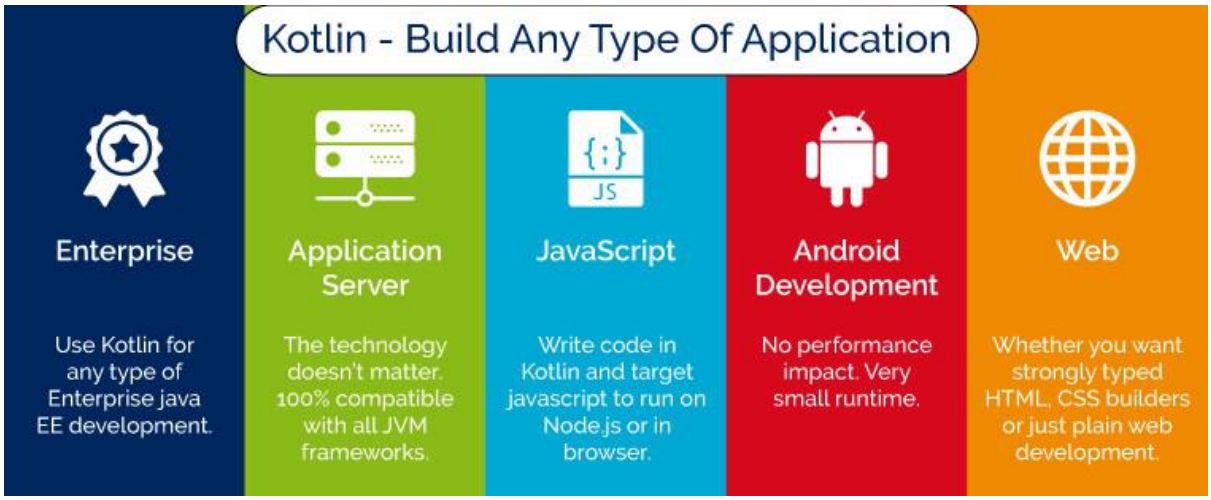
Način pisanja koda se razlikuje od dosadašnjeg programiranja u Javi. Iako je objektno orijentirani programski jezik, prvenstveno se piše stilom funkcionalnog jezika koji koristi lambda izraze. Lambda je anonimna funkcija, odnosno funkcija koja nema identifikator. Često se koristi kao argument koji treba proslijediti, a da se izbjegne pisanje definicije funkcije. Iz tog razloga koristi se samo jednom ili mali broj puta jer je sintaktički jednostavnije napisati od imenovanih funkcija.

Osim funkcijskog stila pisanja, postoje mnoge prednosti koje Kotlin pruža kako bi pojednostavnio i ubrzao pisanje koda. Takav način rada dobiven je objedinjavanjem jezika poput Scale, Swifta i JavaScripta tijekom svojih godina razvoja. U ovom radu će biti prikazane prednosti korištenja Kotlin-a u odnosu na Javu te kako se on može iskoristiti za izradu različitih arhitektura za Android aplikacije.

1. Kotlin

1.1. Platforme s kojima kotlin radi

Iako je Kotlin započeo svoj razvoj 2011. godine, trenutno je jedan od najmodernijih jezika. U posljednjih sedam godina svojeg razvoja, jezik se konstantno razvija i unaprjeđuje novim funkcionalnostima i svojstvima. Zahvaljujući podršci od strane Google-a, očekuje se i nastavak tog trenda te sve više programera prelazi s korištenja Jave na Kotlin. Kotlin je najviše pogodio tržište razvoja Android aplikacija, ali bitno je napomenuti da sve veći broj programera koji Koriste Javu ili JavaScript programski jezik za razvoj web stranica i pozadinskih servisa prelazi na potpuno korištenje Kotlin-a (Adit Microsys, 2016).



Slika 1. Kotlin za razvoj više vrsta aplikacija

Slika 1 pokazuje za šta se sve Kotlin može koristiti. Možemo vidjeti da ga možemo koristiti za izgradnju svih vrsta aplikacija osim onih za iOS operacijski sistem. Iako je to trenutno nemoguće, trenutno se radi na verziji koja se zove „Kotlin Native“ pomoću koje bi trebali moći napisati aplikaciju za android i iOS u isto vrijeme zbog podrške za razvoj više-platformskih aplikacija.

1.2. Nove funkcionalnosti u Kotlinu

1.2.1. Ekstenzijske funkcije

Ekstenzijske funkcije nam omogućavaju da dodamo ili promijenimo već postojeće funkcije neke klase bez da je moramo naslijediti ili implementirati. Kako bi dodali ekstenzijsku funkciju, potrebno je navesti klasu te ime funkcije nakon tačke, „fun“ ImeKlase.ime funkcije(parametri).

```
fun String.podjeliZnakovniNiz() {
    this.forEach {
        println(it)
    }
}
```

U datom primjeru definiramo funkciju *podjeliZnakovniNiz* nad već postojećom klasom „String“. Funkcija ne prima parametre nego koristi ključnu riječ this kako bi pristupila znakovnom nizu nad kojim je pozvana. Funkcija iterira po svakom znaku u nizu i ispisuje ga u 10 novi red. Ekstenzijske funkcije najčešće se postavljaju u gornji sloj aplikacije direktno ispod paketa kako bi bile dostupne svim klasama.

```
"Znakovni niz".podjeliZnakovniNiz()
```

Osim definiranja novih funkcija moguće je i izmijeniti postojeće. Ekstenzijske funkcije su vrlo moćne jer se mogu koristiti kao biblioteke nad klasama koje su česte u razvoju aplikacija. Iz tog razloga Google je napravio gotovu klasu s ekstenzijskim funkcijama koje se najčešće koriste kako bi bile dostupne svima. Da bi se pristupilo toj klasi u android projektu potrebno je dodati sljedeću liniju „apply plugin: 'kotlin-android-extensions'“. Najčešći primjer ekstenzijske funkcije je prikazivanje ili sakrivanje pojedinih elemenata na ekranu.

```
fun View.prikazi(show: Boolean = true) {
    visibility = if (show) VISIBLE else GONE
}
fun View.prikazi(hide: Boolean = true) {
    visibility = if (hide) GONE else VISIBLE
}
```

Korištenjem ekstenzijskih funkcija dovoljno je pozvati metodu nad nekim elementom ekrana, dok bi u protivnom svaki put bilo potrebno postavljati vidljivost elementa. Ekstenzije moguće je definirati i nad svojstvima, npr. ako želimo svojstvo koja vraća zadnji element u listi definirati ćemo svojstvo „zadnjiElement“.

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

1.2.2. Podatkovne klase

Prilikom izrade aplikacije, često nam trebaju klase koje služe samo kao modeli nekog objekta iz stvarnog svijeta. Takve klase često ne sadrže dodatne metode nego samo atribute realnih objekata. U Javi takve klase se nazivaju POJO (eng. Plain Old Java Object) obični stari java objekti. Kada koristimo takve klase, potrebne su nam neke ugrađene metode za kopiranje objekata, uspoređivanje ili pretvaranje u znakovni niz.

U Javi je takve metode potrebno napisati ručno zbog čega klase od samo 5 atributa znaju biti dugačke i preko 100 linija. Da bi se ubrzalo pisanje takvih klasa, Kotlin uvodi podatkovne klase, odnosno klase koje će u pozadini već imati implementirane metode koje su nam potrebne. Podatkovne klase su obične klase koje dodatno koriste ključnu riječ data.

```
data class Osoba(val ime: String,
                 val prezime: String,
                 val oib: Long,
                 val spol: String,
                 val dob: Int)
```

Ovako napisana klasa u pozadini ima implementirane metode toString(), equals() i copy(), hasCode(). Uočite da klasa ne sadrži vitičaste zagrade niti metode za postavljanje ili dohvaćanje podataka. Kotlin svojstva imaju implementirane te metode u pozadini i prilikom pisanja imena svojstva one će se pozivati bez da programer mora brinuti o tome.

1.2.3. Destrukturirajuće deklaracije

Kotlin je jedan od programskih jezika kod kojeg nije potrebno navoditi tip podataka varijable koju inicijaliziramo, osim ako to eksplicitno želimo. Ukoliko definiramo neku varijablu ili svojstvo bez zadavanja tipa podataka i inicijaliziramo je na neku vrijednost, ona će implicitno poprimiti tip podataka te vrijednosti.

```
val ime = "Nejra"
```

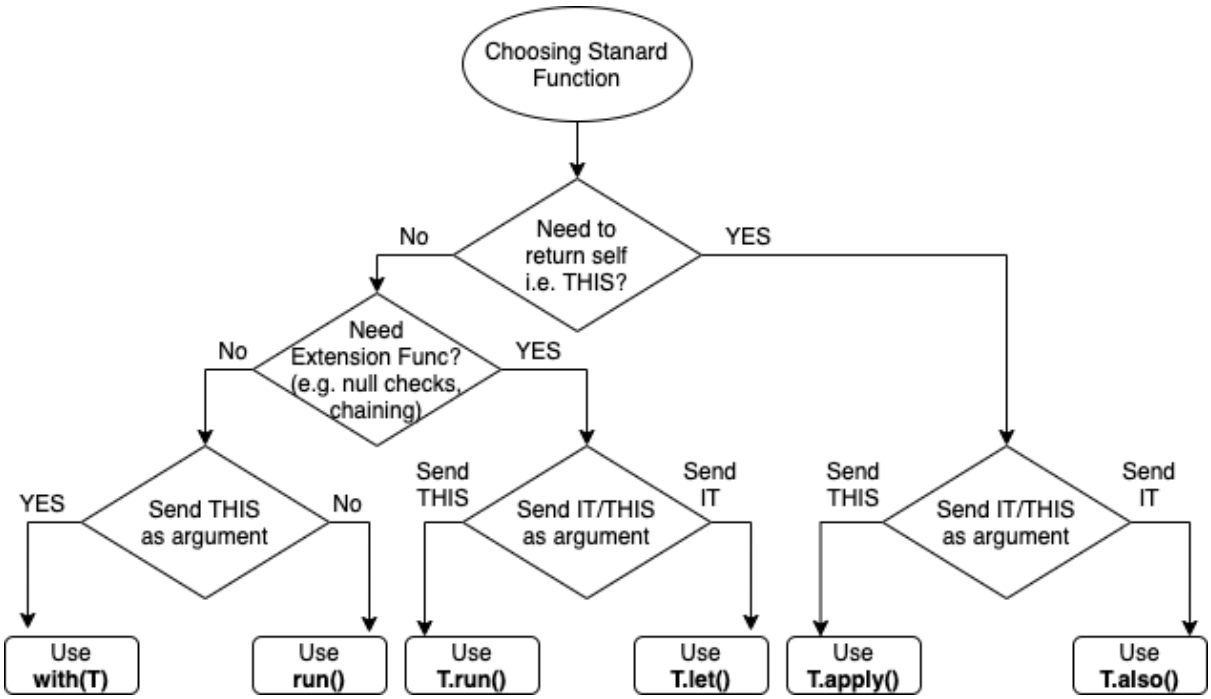
U ovom primjeru svojstvo „ime“ će implicitno poprimiti tip podataka „String“. Osim implicitnog postavljanja tipa podataka, Kotlin uvodi i destrukturirajuće deklaracije, odnosno ukoliko imamo neki tip podataka koji sadrži više svojstava, npr. „Pair“, možemo ga pohraniti u dekonstruirajuću deklaraciju tako da stavimo proizvoljne nazive svojstava između oblih zagrada.

```
val imePrezime = Pair<String, String>("N","Č")
val (ime, prezime) = imePrezime
```

Korištenjem dekonstruirajućih deklaracija dobivamo neimenovani objekat s imenovanim parametrima, za razliku od imenovanog objekta kod kojeg ne znamo imena parametara. Ukoliko bismo željeli pristupiti imenu ili prezimenu preko para „imePrezime“ morali bismo zvati atribute „first“ ili „second“ nad tim objektom. U tom slučaju nije nam vidljivo da li se radi o imenu ili prezimenu ako ne pogledamo dublje u kod. U ovom primjeru ako želimo pristupiti imenu ili prezimenu možemo to napraviti direktno jer smo ih imenovali.

1.2.4. Funkcije konteksta

Funkcije konteksta su funkcije koje koristimo nad bio kojim objektom kako bi pristupili tom objektu na lakši način pod drugačijim kontekstom. Postoji pet takvih funkcija, let(), apply(), run(), with() i also(). Sve funkcije rade na sličan način, ali postoji razlika u kojem slučaju se trebaju koristiti. Postoje dvije razlike između navedenih funkcija, unutar vitičastih zagrada funkcije objekt postaje it ili this, odnosno ukoliko želimo pristupiti objektu nad kojim zovemo funkciju ne moramo više pisati njegovo ime već mu pristupamo iz konteksta. Kada radimo s podacima nekad želimo kao povratnu vrijednost vratiti sam objekt nad kojim radimo dok u drugim slučajevima želimo vratiti neki novi objekt koji će biti rezultat akcija. To su neki od parametara koji odlučuju kada ćemo koristiti neku od navedenih funkcija umjesto druge.



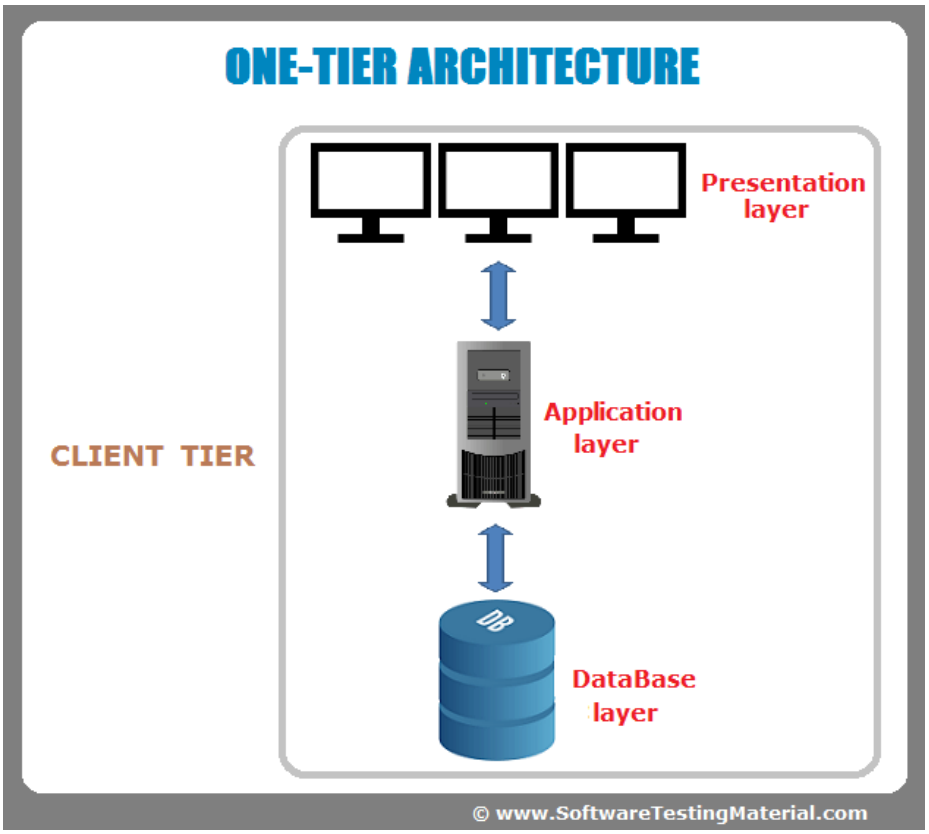
Slika 2. Dijagram korištenja funkcija konteksta

Na slici 2 možemo vidjeti dijagram koji objašnjava kada se koja funkcija treba koristiti.

2. Arhitektura Android aplikacija

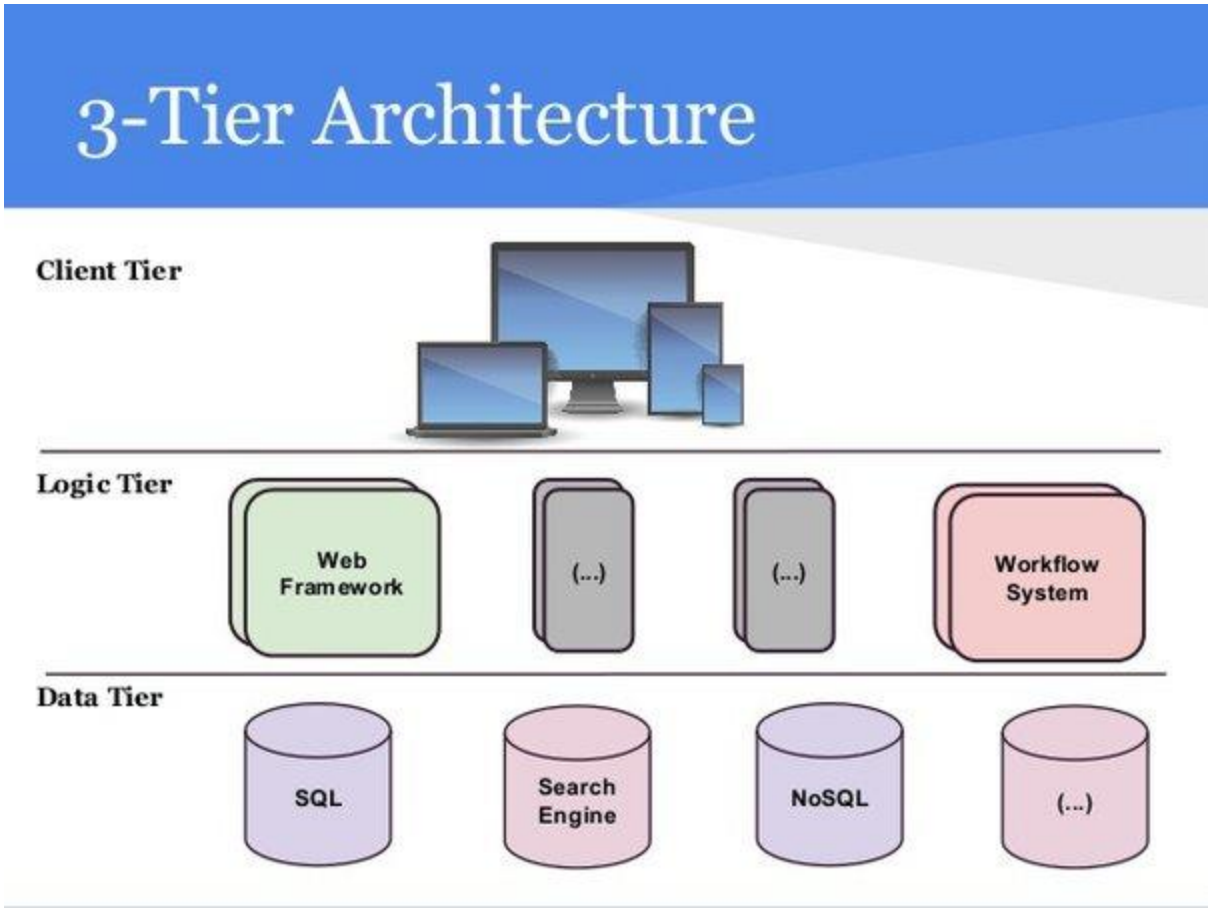
2.1. N-slojna arhitektura

Prilikom izrade android aplikacije često se radi s podacima iz različitih izvora. Prije spremanja podataka ili nakon dohvaćanja, podaci se preoblikuju kako bi odgovarali poslovnoj logici aplikacije. Osim poslovne logike, podaci se negdje moraju prikazati ili unijeti od strane korisnika tako da su vidljivi krajnjem korisniku. Ukoliko sve te korake pišemo na jednom mjestu, dolazi do poteškoća u daljnjem razvoju aplikacije. Prilikom promjene određenog djela aplikacije, programer često mora promijeniti više metoda jer je sve povezano zajedno. Osim potrebe za promjenom velike količine koda, kod je nečitljiv što prouzrokuje teško snalaženje novih programera. Takvu arhitekturu zovemo jednoslojna arhitektura. (Rajkumar, 2017).



Slika 3. Jednoslojna arhitektura

Ako govorimo o profesionalnoj aplikaciji, one sadrže testove za pojedine komponente aplikacije kako bi se dodatno provjerilo da podaci neće doći u neočekivano stanje ili ukoliko je to neizbježno da su takvi slučajevi riješeni od strane programera. Kako bi se izbjegli i riješili navedeni problemi, potrebno dizajnirati arhitekturu. Osnovna arhitektura bilo koje aplikacije sugerira da se aplikacija podijeli u tri sloja, sloj korisničkog sučelja, sloj poslovne logike i sloj podataka. Takvu arhitekturu zovemo troslojna arhitektura, a sve varijacije postojećih arhitektura u sebi sadrže ovakvu podjelu jer bez njih aplikacija ne može funkcionisati. (Ryan Jentzsch, 2016).



Slika 4. Troslojna arhitektura

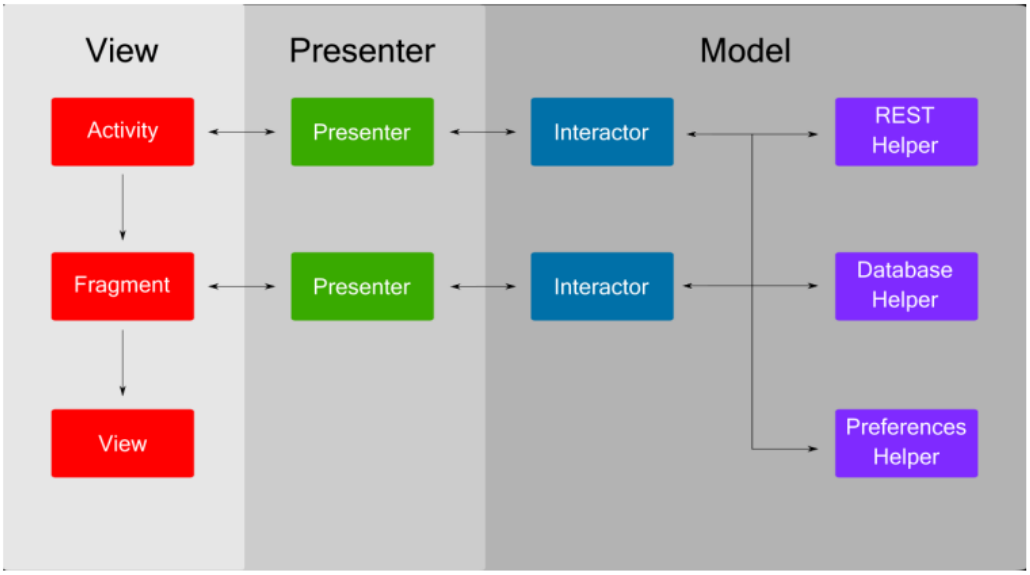
Slika 4 prikazuje protok podataka kroz troslojnu arhitekturu.

3. Značajke android arhitekture

Većina arhitektura je napravljeno da rade za bilo koju platformu iako postoje specifičnosti unutar svake. Iz tog razloga arhitektura sama po sebi sadrži osnovne komponente dok se specifične komponente nadodaju ovisno o platformi. Osnovne komponente se uglavnom baziraju na uzrocima dizajna kako bi se olakšalo programiranje i pojednostavilo testiranje.

3.1. MVP (Model – View – Presenter) arhitektura

Kada se uzmu principi troslojne arhitekture i nadograde s navedenim značajkama android arhitekture, dobiva se osnovni način rada svake moderne arhitekture koja se koristi za razvoj android aplikacija. Kreiranjem različitih veza između slojeva i dodavanjem dodatnih odgovornosti, može se izgraditi konkretna arhitektura iz one osnovne. Jedan primjer takve arhitekture je MVP koja se zasniva se na tri sloja gdje view odgovara sloju korisničkog sučelja, presenter odgovara sloju poslovne logike, a model odgovara sloju podataka. Ukoliko odemo korak dalje, često se koristi i četvrti sloj interactor koji je posrednik između presenter-a i modela. U tom slučaju interactor preuzima odgovornost za poslovnu logiku, a presenter prima gotove obrađene podatke. (Yong Heui Cho, 2016),



Slika 8. . MVP arhitektura

Na slici 8 možemo vidjeti princip rada MVP arhitekture. Sloj view sadrži aktivnosti, fragmente i ostale elemente koji se prikazuju na ekranu. Podaci iz tog sloja mogu biti poslani u presenter ukoliko korisnik napravi neku akciju ili mogu doći iz presenter-a ukoliko je potrebno prikazati nove podatke na ekranu.

Sloj presenter prima unesene podatke iz view-a i daje ih interactor-u na obradu ili prima obrađene podatke od interactor-a i predaje ih u view gdje će se oni prikazati.

Podaci unutar model sloja mogu doći iz interne baze podataka ili web servisa. Ti podaci se šalju u interactor na obradu kako bi odgovarali pojedinim slučajevima korištenja. Suprotno, 30 već obrađeni podaci mogu doći iz interactor sloja i trebaju biti spremljeni u bazu podataka ili poslani preko web servisa.

3.1.1. Model

Model sloj reprezentira sloj podataka i kao takav je zadužen za dohvat i slanje podataka iz bilo kojeg izvora. Izvor može biti web servis, interna baza podataka, lokalni JSON dokument, bluetooth servisi i sl.

Kada sloj primi podatke, oni znaju biti u različitim oblicima ili je moguće da dobivamo suvišne podatke koji nam ne trebaju. Da bi smanjili potrošnju memorije i vrijeme obrade podataka u sloju poslovne logike, potrebno je dobivene podatke mapirati iz modela podataka u takozvane domenske modele koji odgovaraju domeni na koju se odnose.

```
class TemplateClientImpl(private val templateService: TemplateService,
                         private val apiMapper: ApiMapper) : TemplateClient {

    override fun getNews(): Single<List<Article>> {
        return templateService.getNews()
                .map { it -> apiMapper.toArticles(it) }
    }
}
```

Na primjeru iz koda možemo vidjeti kako izgleda poziv web servisa za dohvaćanje podataka o vijestima. Koristi se objekt servisa koji dohvaća podatke i zatim ih mapira u listu članaka. U ovom trenutku imamo podatke spremne za obradu. Podaci se spremaju u repozitorij podataka i zatim se prosljeđuju u sloj poslovne logike interactor.

3.1.2. Presenter

Sloj presenter prima obrađene podatke članaka od interactor-a i predaje ih u sloj view gdje će se prikazati. Budući da postoji obostrana veza između slojeva, tok podataka može ići i u drugom smjeru, odnosno od korisnika prema obradi podataka.

```
addDisposable(getNewsUseCase.run()
            .subscribeOn(backgroundScheduler)
            .map { it.map { NewsViewModel(it.title, it.description, it.author,
it.imgUrl) } }
            .observeOn(mainThreadScheduler)
            .subscribe(this::onGetNewsSuccess, Throwable::printStackTrace))
}
```

U ovom ranije prikazanom primjeru, možemo vidjeti ulogu presenter-a. On je taj koji se pretplaćuje na podatke i zatim ih mapira u podatke koji su prilagođeni za sloj view.

3.1.3. View

View se odnosi na sam ekran, odnosno na ono što korisnik može vidjeti na ekranu. On je odgovoran za korisnikove akcije poput pritiska na dugme, pisanja teksta i sl. Sve podatke koji se prikazuju na ekranu dobiva od presenter-a ili od korisnika koji ih unese.

```
class HomeActivity : BaseActivity(), HomeContract.View {

    @Inject
    lateinit var presenter: HomeContract.Presenter

    @Inject
    lateinit var adapter: NewsListAdapter

    lateinit var recyclerView: RecyclerView

    override fun inject(activityComponent: ActivityComponent) {
        activityComponent.inject(this)
    }

    override fun getPresenter(): ScopedPresenter {
        return presenter
    }

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

        recyclerView = activity_home_recycler_view
        initRecyclerView()
        presenter.showNews()
```

```
    }

    private fun initRecyclerView() {
        recyclerView.layoutManager = createLinearLayoutManager()
        recyclerView.adapter = adapter
    }

    private fun createLinearLayoutManager(): LinearLayoutManager {
        return LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false)
    }

    override fun render(newsViewModels: List<NewsViewModel>) {
        adapter.submitList(newsViewModels)
    }
}
```

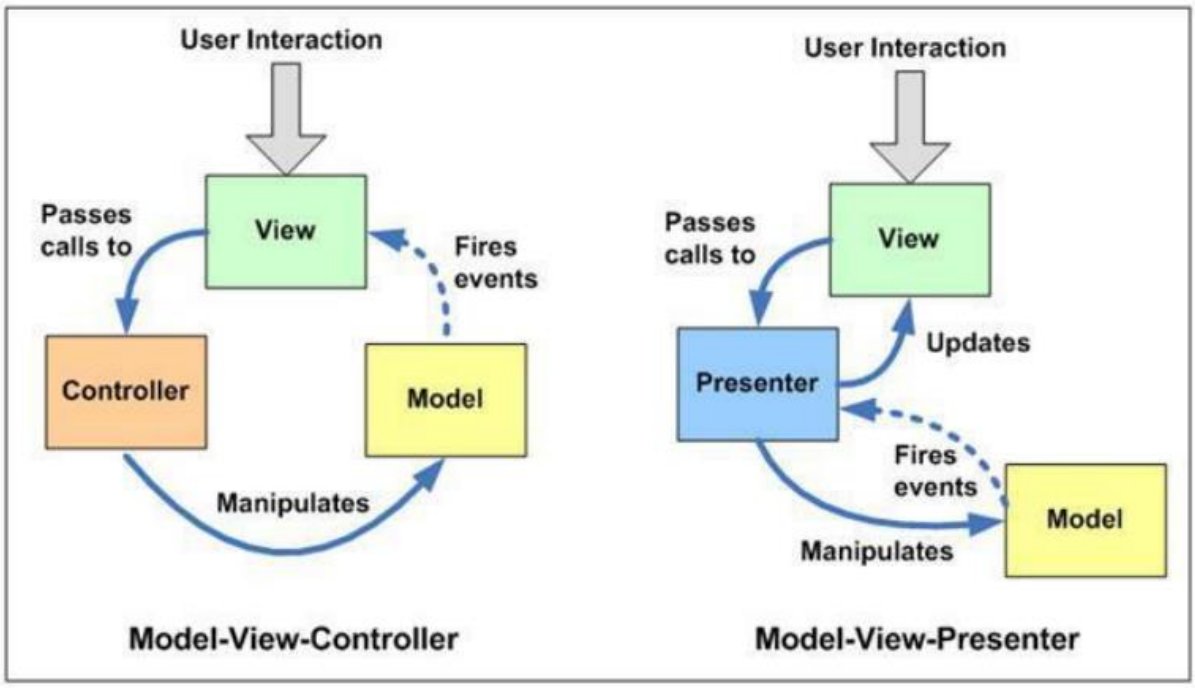
U danom primjeru može se vidjeti kako izgleda kod aktivnosti koja pripada sloju view. Većina koda odnosi se na postavljanje elemenata na ekranu poput „recyclerView-a“ i njegovih adaptera. Ono što je bitno za arhitekturu je da view sadrži instancu objekta presenter. Ukoliko korisnik napravi akciju na ekranu ili želimo postaviti neke početne podatke na ekran, potrebno je pozvati metodu nad tim objektom.

U primjeru možemo vidjeti kako se prilikom kreiranja ekrana poziva metoda „showNews()“ nad objektom „presenter“. Nakon što se vijesti dohvate, presenter poziva metodu „render“. View nadjačava metodu „render“ u kojoj primljene podatke o vijestima prikazuje na ekranu.

3.2. MVC(Model-View-Controller) arhitektura

Jedan od poznatijih uzoraka dizajna za arhitekturu je MVC. Najčešće se koristi kod izrade web i desktop aplikacija, dok se za izradu android aplikacija koristi samo u nekim dijelovima svijeta.

MVC arhitektura je vrlo slična prethodno objašnjenoj MVP arhitekturi, a sadrži i dva ista sloja, model i view dok je razlika u sloju poslovne logike controller. Osim imena, controller se razlikuje od presenter-a po tome da se u njemu nalazi logika koja odgovara na akcije korisnika. Dok je u MVP uzorku dizajna svaki view sloj imao svoj presenter sloj, u MVC uzorku dizajna više view slojeva mogu dijeliti zajednički controller sloj, a on će odlučivati koji view treba trenutno biti aktivan.



Slika 9. Razlike između MVC i MVP arhitektura

Na slici 9 možemo vidjeti glavne razlike između MVC i MVP arhitekture. U MVP arhitekturi, view i model nisu imali direktnu vezu komunikacije dok je u MVC arhitekturi stvar drugačija. Postoji posebni model prezentacijskoj sloja. View se pretplaćuje na promjene tog modela. Kada korisnik napravi akciju na ekranu, akcija se prosljeđuje s view-a na controller koji zatim radi promjenu nad modelom podataka. Budući da je view pretplaćen, on će se indirektnom vezom automatski promijeniti bez da podaci idu nazad preko controller-a.

3.2.1. Model

Kod model sloja ostaje isti kao što je ranije objašnjen u MVP arhitekturi. Međutim, njegova svrha više nije da vraća podatke u sloj poslovne logike već da ih predaje direktno na view.

3.2.2. Controller

Controller ima zadaću odgovoriti na akciju korisnika, po potrebi promijeniti view i zatražiti promjenu nad model-om. Za razliku od MVP arhitekture, Controller se ne pretplaćuje na promjene u model-u, niti ne predaje nikakve podatke nazad u view već samo manipulira trenutnim podacima.

```
override fun changeAuthor(author: String) {
    newsRepository.changeAuthor(author)
}

override fun removeArticle(articleId: Int) {
    newsRepository.removeArticle(articleId)
}


override fun addArticle(article: Article) {
    newsRepository.addArticle(article)
}
```

U danom primjeru možemo vidjeti uobičajene metode za promjenu trenutnih podataka. Controller će zatražiti pohranu podataka od model-a i izvršiti promjenu prema odgovarajućoj poslovnoj logici ukoliko je to potrebno.

3.2.3. View

View se pretplaćuje na podatke koje želi prikazati na ekranu. Na neku korisnikovu akciju, poziva se controller koji je odgovoran za nju. View ne smije znati što se događa u pozadini, već samo zna da očekuje nove podatke na koje je pretplaćen.

```
class HomeActivity : BaseActivity(), HomeContract.View {

    @Inject
    lateinit var controller: NewsContract.Controller

    @Inject
    lateinit var adapter: NewsListAdapter

    lateinit var recyclerView: RecyclerView

    override fun inject(activityComponent: ActivityComponent) {
        activityComponent.inject(this)
    }
```

```
    override fun getPresenter(): ScopedPresenter {
        return presenter
    }

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

        recyclerView = activity_home_recycler_view
        initRecyclerView()

         addDisposable(getNewsUseCase.run()
                    .subscribeOn(backgroundScheduler)
                    .map { it.map { NewsViewModel(it.title, it.description,
        it.author, it.imgUrl) } }
                    .observeOn(mainThreadScheduler)
                    .subscribe(this::render, Throwable::printStackTrace))
         }
    }

    @OnClick(addButton)
    public fun onAddClicked(){
        controller.addArticle(Article(author.text, article.text, description.text))
    }

    @OnClick(removeButton)
    public fun onRemoveClicked(view : View){
        controller.removeArticle(view.id))
    }

    override fun render(newsViewModels: List<NewsViewModel>) {
        adapter.submitList(newsViewModels)
    }
}
```

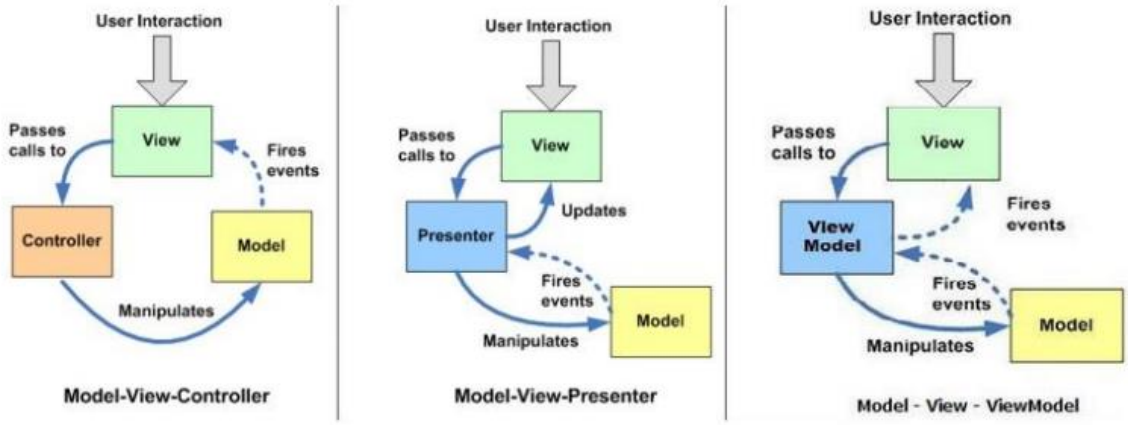
U danom primjeru možemo vidjeti kako izgleda aktivnost unutar arhitekture MVC. Aktivnost sadrži instancu objekta „controller“ koji je tipa „NewsContract.Controller“ što znači da će svi ekrani koji prikazuju vijesti u bilo kakvom obliku koristiti taj isti zajednički controller.

Unutar metode „onCreate“ umjesto zahtjeva za dohvaćanja podataka možemo vidjeti pretplatu na podatke koja se ranije nalazila u presentersloju. Ukoliko stignu novi podaci, poziva se metoda „render“ koja prikazuje te podatke u listi na ekranu.

Na dnu možemo vidjeti nadjačane metode koje se pozivaju kada korisnik klikne na dugme. U tom slučaju view traži od controller-a da napravi odgovarajuću akciju.

3.3. MVVM(Model-View-ViewModel) arhitektura

MVVM je novija arhitektura u svijetu android-a i odnedavno su od strane Google-a napravljene pomoćne klase za njenu lakšu implementaciju. Arhitektura je vrlo slična prethodnim dvjema, odnosno sadrži karakteristike od jedne i druge. Prema vezama između slojeva, MVVM je identičan MVP-u, ali razlikuje se po tome da je sloj gdje se pretplaćuje na podatke view, isto kao u MVC arhitekturi.



Slika 10. Razlike između MVC, MVP i MVVM arhitektura

3.3.1. Model

Kod model sloja ostaje isti kao što je ranije objašnjen u MVP arhitekturi zbog čega nema potrebe da se ponovo prikazuje.

3.3.2. ViewModel

ViewModel je sličan presenter sloju u varijanti MVP arhitekture bez interactora, odnosno odgovoran je za poslovnu logiku. Razlika je u tome da je presenter pretplaćen na izvor podataka i dobivene podatke predaje na view dok viewModel dohvaća podatke bez pretplate. Na taj način on postaje izvor na koji se treba pretplatiti.

```
class ProjectListViewModel(application: Application, newsRepository:
NewsRepository) : AndroidViewModel(application) {


override fun newsFlowable() : Flowable<List<News>>{

    return projectListObservable

}

 val projectListObservable: Flowable<List<News>>


 init {
     projectListObservable = Flowable.just(newsRepository.news())
 }
}
```

ViewModel prilikom inicijalizacije dohvaća podatke o vijestima iz repozitorija podataka i zatim ih omata kao reaktivni niz podataka kao što je to bilo spomenuto ranije. Nadjačana metoda koja je izložena view-u„newsFlowable“ vraća taj reaktivni niz kako bi se na njega moglo pretplatiti.

Cijela klasa nasljeđuje od klase „AndroidViewModel“, odnosno nasljeđuje pomoćnu klasu koju je Google napravio. Prilikom programiranja android aplikacija, potrebno je paziti na konfiguracijske promjene. To su promjene koje se događaju u posebnim uvjetima kao što je rotiranje ekrana. Prilikom konfiguracijske promjene, aktivnosti se ubijaju i ponovo kreiraju s novim konfiguracijama. Ukoliko dohvaćamo neke početne podatke prilikom pokretanja aktivnosti, svaki put kada se dogodi konfiguracijska promjena ti podaci će se ponovo dohvaćati. Ova pomoćna klasa služi tome da klase koje ju nasljeđuju ne budu uništene prilikom konfiguracijske promjene, odnosno u tom slučaju neće biti potrebno ponovo dohvatiti iste podatke. (Google Developers, 2018).

3.3.3. View

Kao i u MVC arhitekturi, glavna uloga view sloja je da se pretplati na podatke. Razlika je u tome što izvor podataka više nije u model već viewModel koji sadrži metodu izloženu za pretplatu.

3.4. Uporedba arhitektura

Kada bismo krenuli u izradu android aplikacije, morali bismo odabrati neku arhitekturu prema kojoj ćemo raditi. Prilikom odabira, važno je znati koje su prednosti i koji su nedostaci pojedine arhitekture.

4. Tesiranje

Kao što je već ranije napomenuto, testiranje aplikacije je bitno kako bismo izbjegli neugodne situacije u konačnoj verziji. Pisanje testova je brže nego ručno testiranje i jednom kada ih napišemo, ukoliko napravimo promjene, testovi će pokazati da li postoje pogreške prilikom implementacije promjena.

MVP arhitektura je orijentirana na lakoću testiranja. Presenter i model slojevi nemaju specifične metode vezane za android operacijski sistem i iz tog razloga se mogu testirati jediničnim testovima pomoću standardne biblioteke „jUnit“. Ako želimo testirati view, potrebno je pisati instrumentacijske testove koji će simulirati korisničke akcije na ekranu. Problem kod MVP arhitekture može nastati ukoliko ne koristimo četvrti sloj interactor kao pomoćni sloj između presenter-a i model-a. U tom slučaju je moguće da će se u većim aplikacijama nakupiti previše koda unutar presentera što znači ulaganje veće količine vremena na pisanje testnih metoda. (Kapil Sharma, 2017).

Kada želimo testirati MVC arhitekturu, nailazimo na problem da moramo pisati instrumentacijske testove i za view i za controller. Budući da je controller vezan uz korisničke akcije, to znači da je vezan uz klase koje su specifične za android operacijski sustav i kao takve ne mogu biti testirane jediničnim testovima.

MVVM, slično se može testirati slično kao i MVP. Razlika je u tome da MVVM nema presenter zbog čega ima ukupno manje klasa za testiranje. Problem kod MVVM arhitekture se nalazi u tome da je teže testirati view budući da xml datoteke zadužene za izradu izgleda ekrana sadrže podatke u sebi.

Kada se svi podaci uzmu u obzir, može se zaključiti kako je MVC arhitektura najteža za testiranje. MVVM arhitektura zahtjeva najmanje testnih slučajeva, ali ima problem u testiranju view sloja. Iz navedenih razloga, s obzirom na kriterije testiranja, preporučuje se MVP arhitektura.

5. Zaključak

Otkako je Google najavio da će pružati podršku u razvoju i održavanju jednom od novijih jezika Kotlin, njegova popularnost je naglo porasla. Kotlin je postao preporučeni jezik prilikom izrade android aplikacija i sve više novijih biblioteka podržava razvoj u Kotlin-u.

Svojim prednostima u programiranju naspram programskog jezika Jave, privlači sve više programera unutar drugih domena programiranja koji rade na drugim platformama. Zbog naglog rasta popularnosti, sve je veća potreba za predloškom po kojem bi programeri trebali raditi prilikom izrade aplikacija.

Predložak izrade aplikacija drugim riječima možemo nazvati i arhitektura jer se arhitektura aplikacije postavlja na početku, prije dodavanja novih funkcionalnosti. Arhitektura aplikacije određuje način na koji ćemo pisati aplikaciju, kako ćemo odvojiti slojeve u aplikaciji, koliko lako će biti testirati aplikaciju ili kako će se voditi njeno održavanje.

Prije nego se krene u izradu aplikacije, programski arhitekt zajedno s programerima mora diskutirati i odabrati arhitekturu koja zadovoljava svojstva i funkcionalnosti aplikacije koja će se izrađivati. Iz tog razloga potrebno je znati prednosti i nedostatke između najpopularnijih i najkorištenijih arhitektura.

U ovome radu uspoređivale su se tri arhitekture, MVP, MVC i MVVM, svaka sa svojim prednostima i nedostacima. Arhitektura MVC se pokazala najlošijom prema svim kriterijima, dok su arhitekture MVP i MVVM podjednake.

Ostaje problem kako se odlučiti između dvije podjednake arhitekture. Odgovor se nalazi u kombinaciji arhitektura, odnosno kako MVVM zadržava podatke prilikom promjene konfiguracije, rješenje je da umjesto podataka zadržava čitav presenter sloj koji će imati podatke pohranjene u sebi. Na taj način možemo zadržati najbolje karakteristike od obje arhitekture.

6. Mišljenje o radu

Potpis mentora:

_____

Prof. Amir Karačić

7. Literatura

[1] Kotlin Programming Language https://kotlinlang.org/

[2] Uvod u Kotlin – Koje su prednosti novog zvaničnog jezika za razvoj Android aplikacija https://startit.rs/uvod-u-kotlin-koje-su-prednosti-zvanicnog-jezika-za-razvoj-android-aplikacija/

[3] Kotlin Tutorial https://www.w3schools.com/KOTLIN/index.php

[4] Arhitektura Android aplikacija https://repozitorij.etfos.hr/islandora/object/etfos%3A2002/datastream/PDF/view

[5] Simple example of MVVM architecture in Kotlin https://dev.to/whatminjacodes/simple-example-of-mvvm-architecture-in-kotlin-4j5b

[6] Implementing MVC pattern in Android with Kotlin https://www.codementor.io/@dragneelfps/implementing-mvc-pattern-in-android-with-kotlin-i9hi2r06c

[7] MVP Android Kotlin Architecture https://www.manektech.com/blog/mvp-android-kotlin-architecture

[8] MVC vs MVP vs MVVM http://2.bp.blogspot.com/-nNwsxDAC3Og/VFOEMQDn4mI/AAAAAAAABDA/G7r3_tLAODM/s1600/image001-757983.png

[9] MVC vs MVP https://nirajrules.files.wordpress.com/2009/07/mvcmvp2.jpg

[10] Application Structure https://image.slidesharecdn.com/5-151008064036-lva1-app6892/95/android-message-3-638.jpg?cb=1477993052