Dizajn i implementacija troslojne arhitekture pri razvoju aplikacija za Android

Čulina, Kristijan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Kristijan Čulina

DIZAJN I IMPLEMENTACIJA TROSLOJNE ARHITEKTURE PRI RAZVOJU APLIKACIJA ZA ANDROID

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Kristijan Čulina

Matični broj: 0016140748

Studij: Informacijski sustavi

DIZAJN I IMPLEMENTACIJA TROSLOJNE ARHITEKTURE PRI RAZVOJU APLIKACIJA ZA ANDROID

ZAVRŠNI RAD

Mentor:

Izv. prof. dr. sc. Stapić Zlatko

Varaždin, rujan 2021

Kristijan Čulina

Izjava o izvornosti

Izjavljujem da je moj završni 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.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Najčešće korištena arhitektura pri razvoju programskih proizvoda je troslojna aplikacijska arhitektura. Međutim, kod razvoja mobilnih aplikacija postoje brojne specifičnosti zbog kojih je teško primijeniti klasične pristupe u primjeni troslojnih arhitektura. Stoga, cilj ovog završnog rada je analizirati moguće pristupe u dizajnu i implementaciji troslojne arhitekture pri razvoju mobilnih aplikacija, s posebnim naglaskom na arhitekturu predloženu u sklopu Android JetPack najboljih praksi.

Ključne riječi: razvoj mobilnih aplikacija, troslojna aplikacijska arhitektura, Android, JetPack

Sadržaj

1. Uvod

Tema završnog rada je dizajn i implementacija troslojne arhitekture pri izradi aplikacije za Android. Detaljno ću objasniti pojam arhitekture, odnosno što riječ arhitektura predstavlja, zašto je uopće bitna te ću pokazati kakve sve arhitekture postoje i koje su osnovne razlike između njih. Također, dotaknut ću se i Android Jetpack-a kojeg ću koristiti pri izradi naše aplikacije. Za izradu aplikacije odlučio sam pratiti arhitekturalni pristup predložen u sklopu Android Jetpacka. Bitno je napomenuti da kad se jednom odlučimo za jedan arhitekturalni pristup ne možemo ga promijeniti u polovici izrade projekta. Također, sama promjena arhitekturalnog pristupa je izrazito skup i dugotrajan proces koji iziskuje dosta vremena. Iz tog razloga je jako bitno na početku projekta imati jasnu predodžbu o funkcionalnim i nefunkcionalnim zahtjevima softvera. Sve to će biti objašnjeno detaljnije u nastavku ovog završnog rada.

Završni rad sastoji se od 11 poglavlja. U uvodnom poglavlju dajem kratki opis teme, motivaciju i cilj završnog rada. Također, u uvodnom dijelu dan je kratki sažetak rada, odnosno opis poglavlja i što ćemo vidjeti u njima.

Cilj završnog rada je objasniti pojam arhitekture te upoznati korisnika s važnošću korištenja određenog arhitekturalnog pristupa pri razvoju bilo koje aplikacije. U drugom poglavlju opisane su metode i tehnike rada, kao i istraživački dio. Drugi dio drugog poglavlja opisuje metodološki pristup koji sam koristio za izradu ovog cijelog završnog rada. Treći dio odnosi se na pojam arhitekture. Upravo u tom poglavlju dobiva se uvid u ono što je zapravo arhitektura, odnosno što upravo taj pojam predstavlja, za što služi i na kraju krajeva - zašto se uopće koristi. Opisani su neki koraci koji bi se trebali poštovati prilikom dizajniranja arhitekture. U četvrtom poglavlju navedene su tri arhitekture koje se koriste pri razvoju aplikacija za Android te je svaka detaljno objašnjena. Peto poglavlje opisuje Android Jetpack, što on predstavlja te detaljan uvid u predloženu arhitekturu koju trebamo pratiti prilikom razvoja Android aplikacija. Upravo tu arhitekturu ćemo i koristiti prilikom implementacije aplikacije u sedmom poglavlju. U šestom poglavlju uzet ćemo već objašnjene arhitekture iz poglavlja četiri, usporediti ih, dati mišljenje i objasniti koju preporučujem i zašto. U sedmom poglavlju, dizajnirati i implementirati ćemo arhitekturu te krećemo s izradom aplikacije. Aplikacija će pratiti već spomenuti arhitekturalni pristup predložen u sklopu Android Jetpacka. U završnom dijelu prikazan je zaključak u kojem je opisan kratak sažetak završnog rada te komentar i mišljenje autora.

Motivacija za odabir ove teme bila je želja da proširim svoje znanje iz programiranja, kao i želja za upoznavanjem Kotlin programskog jezika. Motiviran sam činjenicom da ću izraditi svoju prvu aplikaciju prateći neki arhitekturalni pristup. Do sad sam izradio par jednostavnih Android aplikacija, no, kao što sam i naveo, nikad nisam pratio određeni arhitekturalni pristup, što zna jako „zapapriti“ stvari prilikom programiranja. Najveća motivacija mi je to što volim izazove, a tema ovog završnog rada se čini jako izazovna!

2. Metode i tehnike rada

Glavni alat za izradu Android aplikacije je Android Studio (Developer Android, 2021a). Android Studio je službeno razvojno okruženje (IDE – integrated development environment) za Googleov operativni sustav Android, izgrađen na JetBrains' IntelliJ IDEA softveru i dizajniran posebno za razvoj Androida (Developer Android, 2021b). Sav programski kôd pisan je u Kotlin programskom jeziku (Kotlin, 2021a) unutar Android Studija. Kotlin kao jezik je postao jako popularan u zadnje vrijeme pri izradi Android aplikacija. Razlog tome je Googleova obavijest da je Kotlin programski jezik sad njegov preferirani jezik za razvoj Android aplikacija. Koliko je velika upotreba Kotlina u današnje vrijeme govori i podatak da preko 60% od top 1000 aplikacija na Play Storeu koristi Kotlin programski jezik (Kotlin, 2021b). Aplikaciju sam testirao direktno na svom Samsung Galaxy A7 Android mobitelu putem USB otklanjanje pogrešaka opcije razvoja, koji je jako zgodna značajka Android Studija. Android Studio također ima opciju da dodamo, odnosno napravimo, virtualni uređaj (AVD – Android Virtual Device), što sam iz početka i koristio, ali to je značajka koja zahtjeva minimalno 16 GB RAM-a i jako moderan procesor, pa sam ubrzo odustao od toga. Za pisanje dokumentacije, odnosno završnog rada koristio sam MS Word (Microsoft, 2021).

Kao što sam i ranije spomenuo, kad sam tek krenuo raditi na ovom završnom radu koristio sam Android virtualni uređaj u sklopu Android Studija. Korištenjem Android virtualnog uređaja naišao sam na niz problema kao što su pregrijavanje laptopa, zamrzavanje slike te nemogućnost kretanja kroz Android Studio. Iz početka nisam znao što se događa, no detaljnijom analizom sam došao do zaključka da Android virtualni uređaj treba dosta radne memorije za rad, kao i moderniji procesor. Morao sam smisliti drugo rješenje te sam analizom svih opcija koje sam imao na raspolaganju došao do zaključka da je najbolja opcija testirati aplikaciju direktno na vlastitom Android uređaju, u mom slučaju Samsung Galaxy A7 uređaju. Uređaj, kao i opcija testiranja se pokazala izvanrednom, brzom i zadovoljavajućom.

Za istraživački dio ovog završnog rada veliku zaslugu ima službena dokumentacija Kotlina (Kotlin, 2021c), kao i službena dokumentacija Android Studija (Developer Android, 2021b). Ostatak zasluga ima dobro staro pretraživanje interneta i razni video tutorijali na YouTubeu, koji su mi pomogli da lakše shvatim neke kompleksnije stvari.

Za pristup razvoju ovog završnog rada koristio sam vodopadni model, koji sam pratio prilikom izrade cijelog završnog rada. Prvo ću detaljno objasniti pojam arhitekture, onda krećem na opis arhitekturalnih pristupa koji se koriste u razvoju aplikacija za Android. Poslije

toga prelazim na Android Jetpack gdje detaljno opisujem i prikazujem arhitekturu koja je predložena u sklopu Android Jetpacka. Nakon što završim s Android Jetpackom prelazim na komparativnu analizu triju arhitekturalnih pristupa koji su ranije već opisani. Za završnicu ću implementirati aplikaciju korištenjem predložene arhitekture opisane u prethodnom poglavlju.

3. Arhitektura

U ovom dijelu rada ću objasniti pojam arhitekture, što on predstavlja, za što se koristi i zbog čega je bitan u razvoju softvera.

Prema Harrouku (Harrouk, 2021), arhitektura je proces i rezultat planiranja, dizajniranja i konstruiranja zgrada ili drugih struktura. Dobar dizajn može napraviti građevinu dugotrajnom i atraktivnom dugi niz godina dok loš dizajn građevine može napraviti da građevina uopće ne može normalno stajati na tlu. Posao arhitekta je da udruži snage umjetnosti i znanosti da svaki komadić građevine dođe točno na svoje mjesto. Slično arhitektima, inženjer za razvoj softvera isto treba kombinaciju umjetnosti i znanosti da dostavi zadovoljavajuće rješenje problema, no umjesto blokova, probleme rješava putem kôda.

Što je arhitektura softvera? Postoji puno definicija na ovu temu, no jedna od najpoznatijih dolazi od Ralpha Johnsona: „Arhitektura se odnosi na važne stvari. Što god to bilo.“ (Mino, 2020). U arhitekturi softvera fokus je više stavljen na strukturu nego na implementaciju detalja. Kod arhitekture softvera radi se o donošenju eksplicitnih ključnih odluka koje će omogućiti da izlazni softver ima visoku kvalitetu.

Prvo trebamo razmisliti o kontekstu, odnosno funkcionalnim zahtjevima. Postavlja se pitanje: „Koje sve funkcionalnosti treba naša aplikacija obavljati?“. Moramo znati što sve želimo da je aplikacija koju izrađujemo u stanju napraviti, odnosno, što sve krajnji korisnik može raditi. Uvijek se možemo podsjetiti dobrog starog primjera s bankomatom. Korisnik nakon što ubaci karticu u bankomat, ukuca svoj PIN, ima mogućnosti uplate, isplate, promjene PIN-a i uvida u stanje. Upravo sve ovo nabrojano su funkcionalnosti. Osim funkcionalnih zahtjeva, postoje i nefunkcionalni zahtjevi. Osim toga što bi naš softver trebao raditi, mora se staviti fokus i na onaj dio ponašanja softvera. Prema (Dabbagh, 2014) funkcionalni zahtjevi opisuju funkcionalno ponašanje sustava, dok nefunkcionalni zahtjevi opisuju koliko bi dobro sustav trebao funkcionirati. Opće je poznato da je atribut kvalitete, poput pouzdanosti, promjenjivosti, performansi ili upotrebljivosti, nefunkcionalni zahtjev softverskog sustava. Iako se funkcionalni i nefunkcionalni zahtjevi jako razlikuju, oni imaju ozbiljan utjecaj jedno na drugo. Navodi da je postizanje nefunkcionalnih zahtjeva zajedno s funkcionalnim zahtjevima ključno za uspjeh softverskog sustava. Na primjer, recimo da želimo da naš softver bude održiv. Održivost sustava je njegova sposobnost da pretrpi popravke i evoluciju (Barbacci et al., 1995). Također

želimo da nam softver bude pouzdan. Pouzdanost sustava mjera je sposobnosti sustava da nastavi s radom tijekom vremena (Barbacci et al., 1995).

Osim funkcionalnih i nefunkcionalnih zahtjeva postoje i restrikcije, odnosno ograničenja, koja mogu ograničiti opcije koje imamo prilikom dizajna arhitekture. Recimo, u našoj aplikaciji se moramo ponašati u skladu s GDPRom, što ćemo morati uzeti u obzir prilikom dizajna arhitekture. To će nam možda smanjiti broj opcija funkcionalnosti pa i broj opcija na koji se način može implementirati neka funkcionalnost.

Drugo, dolazimo do prioriteta. Nakon što dobijemo kontekst i znamo što sve naš softver treba obavljati, kako se ponašati i koja ograničenja postoje, moramo posložiti prioritete. Recimo, ako smo u stisci s vremenom, morat ćemo odbaciti neke funkcionalnosti aplikacije jer nemamo dovoljno vremena za sve. Tada trebamo odlučiti koje funkcionalnosti su prioriteti a koje ne. Osim prioriteta funkcionalnosti, postoje i prioriteti nefunkcionalnih zahtjeva. Danas postoje razni operacijski sustavi, no tema ovog završnog rada je Android aplikacija. Prema tome, ne moramo se uopće brinuti o portabilnosti jer se izrađuje aplikacija isključivo za Android. Iz tog razloga možemo se više fokusirati na učinkovitost, održivost i skalabilnost. Nakon što smo prioritizirali stvari, donijeli neke važne odluke u dizajnu softvera, možemo krenuti s dizajnom arhitekture.

Kod dizajna arhitekture najbitnija stavka je fokusirat se na jednu stvar od jednom. Ako odmah krenemo razmišljati o svim mogućim scenarijima u budućnosti, možemo završiti s „overengineered solutionom“ (Kanat, 2008). Prema (Kanat, 2008) „overengineering“ se sastoji od dvije riječi, „over“ što znači preko (u ovom slučaju previše) i „engineer“ što znači dizajnirati i izgraditi. Iz ovoga možemo zaključiti da to znači da smo previše dizajnirali ili previše izgradili. Dolazimo do pitanja, kako znamo da smo „overengineerali“ nešto? Odgovor je lagan, ako dizajn zapravo otežava shvaćanje stvari više nego ih olakšava, došlo je do „overengineeringa“. Najbolja zaštita od „overengineeringa“ je ta da ne dizajniramo previše u budućnost, kao što je i prethodno napisano. Ako nismo sigurni da će nam neke stvari trebati ili nisu prioriteti, možda nije dobra ideja raditi na njima odmah. Kada budemo imali bolji kontekst moći ćemo donijeti bolju odluku. Naravno, važno je napomenuti da nije baš tako jednostavno i lagano završiti s „overengineered solutionom“, odnosno pretjerati. Postoji jasna razlika između predviđanja programskih konstrukta koji će biti potrebni u budućnosti i pretjerivanja u tom procesu. Danas je više „under“ nego „over“ „engineered“ sustava, pa iz tog razloga nije loše staviti malo više fokusa prilikom dizajna arhitektura kako bi se izbjeglo „underengineered“ rješenje. Postoji mnogo arhitekturalnih pristupa koji bi mogli odgovarati tvom softveru. U knjizi Mark Richardsa, Software Architecture Patterns (Richards, 2015) objašnjeno je par različitih pristupa u dizajnu

arhitekture uključujući njihove prednosti i mane. Mark u svojoj knjizi (Richards, 2015) objašnjava važnost praćenja arhitekturalnog pristupa prilikom dizajniranja softvera. Govori o tome kako danas često programeri kreću s programiranjem aplikacije bez formalnog arhitekturalnog plana, što na kraju rezultira skupom neorganiziranih izvornih modula kojima nedostaju jasne uloge, odnosno odnosi između njih i odgovornosti. Problem aplikacija koje ne prate neki formalni arhitekturalni pristup je što je takve aplikacije teško izmijeniti, odnosno promijeniti neke funkcionalnosti.

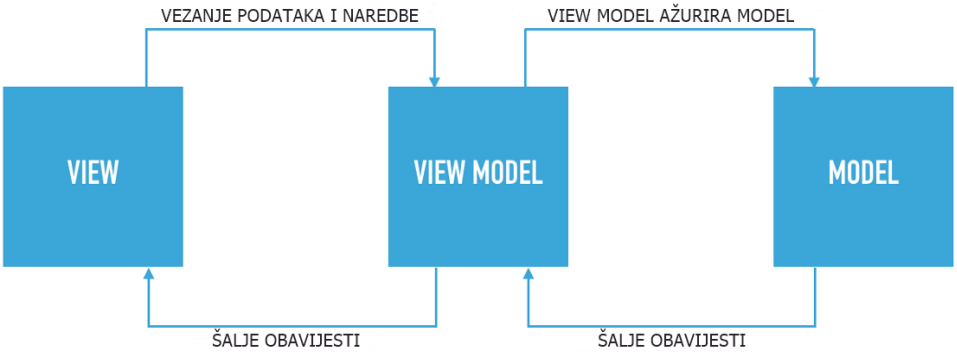
Arhitekturalni pristup pomaže u definiranju osnovnih karakteristika i ponašanja aplikacije. Na primjer, neke se arhitekture prirodno prepuštaju visoko skalabilnim aplikacijama, dok se druge prepuštaju aplikacijama koje su vrlo prilagodljive (eng. agile). Potrebno je poznavati karakteristike, slabosti i mane svakog arhitekturalnog pristupa kako bi odabrali onaj koji zadovoljava vaše specifične poslovne potrebe i ciljeve. Kao arhitekt uvijek morate opravdati svoje arhitekturalne odluke, posebno kada je u pitanju odabir određenog arhitekturalnog pristupa (Richards, 2015).

Arhitektura sustava opisuje njegove glavne komponente, njihove odnose i način međusobne interakcije. Arhitektura softvera i dizajn uključuju nekoliko čimbenika koji doprinose kao što su poslovna strategija (eng. Business strategy), atributi kvalitete, ljudska dinamika, dizajn i IT okruženje (Richards, 2015).

Ukratko, arhitektura sustava je jednostavno organizacija sustava. Ona uključuje sve komponente, način na koji će komponente međusobno komunicirati, okruženje u kojem djeluju

te principe koji se koriste za dizajniranje softvera. Arhitektura se fokusira na relacije između komponenata i komunikaciju između njih.

Na Slici 1 prikazan je MVVM arhitekturalni pristup. Sastoji se od Modela, ViewModela i Modela. Prikazane se veze između pojedinih komponenti, koje će dalje u nastavka biti detaljnije objašnjene.



Slika 1: Prikaz MVVM arhitekturalnog pristupa (Perera, 2019)

Sad kad je dobiven detaljniji uvod u samu arhitekturu i značenje koje ta riječ predstavlja, možemo krenuti na opis arhitekturalnih pristupa koji se koristi pri implementaciji aplikacija za Android. Prvo će biti uvodno poglavlje u kojem će se dobiti uvid u postojeće arhitekturalne pristupe koji se koriste pri razvoju aplikacija za Android. Nakon toga detaljno se opisuje svaki od spomenutih arhitekturalnih pristupa.

4. Arhitekturalni pristupi

U radu (Akhtar & Ghafoor, 2021) postavlja se pitanje: „Zašto je arhitektura softvera potrebna za razvoj softvera?“. Arhitektura softvera je model za strukturu softvera i definira kako će softver raditi. To omogućuje ponovnu uporabu modula za razvoj drugih softvera. Arhitektura se koristi za donošenje važnih strukturnih odluka koje je vrlo skupo mijenjati nakon implementacije. Uočeno je da aplikaciju koja je razvijena bez korištenja arhitekturalnog dizajna vuče niz problema za sobom. Jako ju je teško, odnosno gotovo nemoguće održavati, kao i dalje razvijati. Također, može drastično povećati troškove razvoja. Pravilna arhitektura donosi jednostavnost, ponovnu upotrebu i učinkovito testiranje u našim aplikacijama. Ne postoji fiksna arhitektura za razvoj Android aplikacije. Arhitekturalni dizajni pomažu programerima da strukturiraju svoju aplikaciju kako bi smanjili složenost, što zauzvrat čini testiranje i izmjene lakšima za upravljanje. Programeri koriste različite arhitekturalne pristupe za razvoj Android aplikacija, uključujući MVC (Model-View-Controller), MVP (Model-View-Presenter) i MVVM (Model-View-ViewModel). Studija je pokazala da je MVC najčešće korištena arhitektura za razvoj Android aplikacija, MVP rijetko korištena te MVVM jako rijetko korištena.

U poglavlju koje slijedi upoznat ćemo se s različitim arhitekturalnim pristupima koji se koriste pri razvoju aplikacija za Android.

4.1. MVC

MCV je jedan od osnovnih uvida u rani razvoj grafičkih korisničkih sučelja. MVC je postao jedan od prvih pristupa opisivanju i implementaciji softverskih konstrukcija u smislu njihovih odgovornosti. Trygave Reenskaug predstavio je MVC u Smalltalk-79 tijekom posjeta istraživačkom centru Xerox Palo Alto 1970-ih. Osamdesetih su Jim Alhoff i drugi implementirali verziju MVC-a za bilbioteku Smalltalka. Tek kasnije članak iz 1988 „The Journal of Object Technology (JOT) predstavio je MVC kao opći koncept (Model-View-Controller, 2021).

Prema (Daoudi et al., 2019) MVC arhitekturalni pristup je predložen 1979. godine s ciljem odvajanja poslovne logike od prezentacijskog sloja. Definira tri glavna sloja. Prvi, Model koji predstavlja skup entiteta koji predstavljaju znanje, informacije i skup pravila koja rukuju ažuriranjima i pristupom tim informacijama. Kako bi se osiguralo labavo povezivanje s drugim slojevima, Model ne bi trebao znati za ostale komponente, ali im može slati obavijesti o stanju svojih objekata. Drugi sloj je View, koji je zadužen za vizualni prikaz Modela. Prikazuje podatke koje preuzima iz Modela slanjem upita. Treći, Controller, koji je glavna ulazna točka aplikacije,

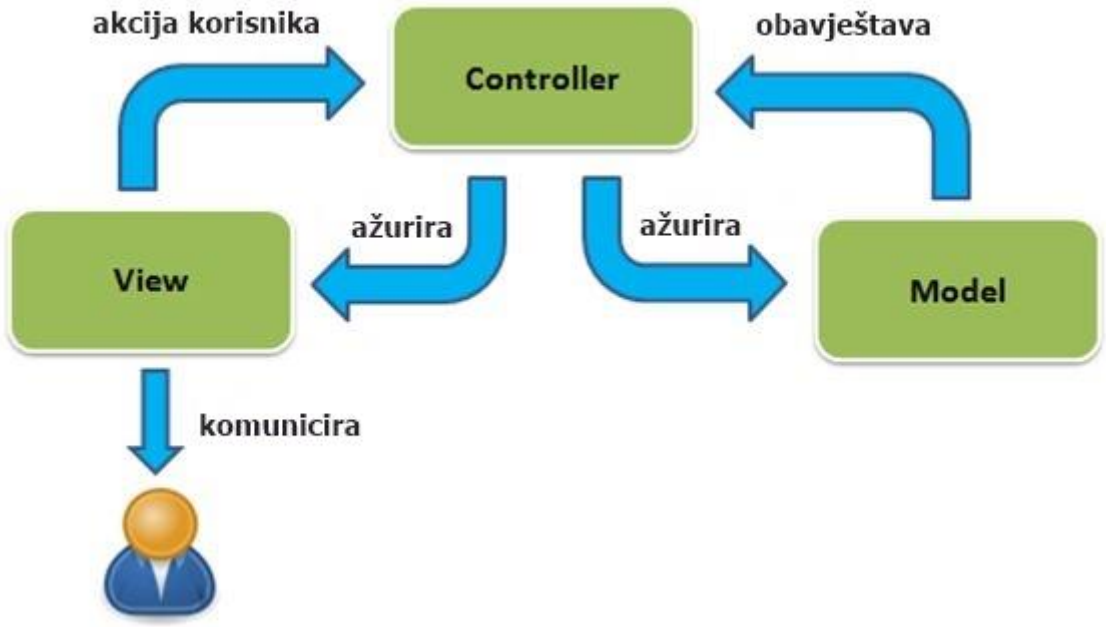
manipulira Viewom i prevodi interakciju korisnika Modelu. Controller komunicira izravno s Modelom i mora biti u mogućnosti promijeniti njegovo stanje.

MVC arhitekturalni pristup sugerira razdvajanje kôda u tri glavne komponente, što znači da je prilikom stvaranja nove klase ili dokumenta posao programera da je smjesti u jednu od tri kategorije:

- Model – Model predstavlja podatke koji se prikazuju na ekranu. Općenito, Model je model domene koji sadrži poslovnu logiku, podatke za manipulaciju i objekte za pristup podacima. (Sokolova & Lemercier, 2013)
- View – Prema (Mishra, 2020) to je sloj grafičkog sučelja (eng. user interface), odnosno sloj koji sadrži sve ono što krajnji korisnik vidi na zaslonu. Također, pruža vizualizaciju podataka pohranjenih u Modelu i nudi interakciju s korisnikom.
- Controller – Ovo je povezna komponenta između Viewa i Modela, odnosno prema (Mishra, 2020) Controller je komponenta koja uspostavlja odnos između Viewa i Modela. Također sadrži logiku aplikacije i dobiva informacije o ponašanju korisnika od Viewa te ažurira Model prema potrebi.

Prema (Maxwell, 2017) Model je mozak aplikacije. View je reprezentacija Modela te je odgovoran za prikaz korisničkog sučelja i komunikaciju s Controllerom kada korisnik stupi u interakciju s aplikacijom. Controller je ljepilo koje povezuje aplikaciju. To je glavni kontroler za ono što se događa u aplikaciji. Kada View kaže Controlleru da je korisnik kliknuo gumb, Controller odlučuje kako će na odgovarajući način stupiti u interakciju s modelom. Na temelju promjena podataka u Modelu, Controller može prema potrebi ažurirati stanje View komponente.

Na Slici 2 ispod možemo vidjeti kako MVC arhitekturalni pristup zapravo izgleda. Krajnji korisnik komunicira s View komponentom, koja kao što je već prethodno napisano predstavlja grafičko sučelje aplikacije. Kada krajnji korisnik klikne na neki gumb View komponenta poziva odgovarajući Controller, koji zatim ažurira Model prema potrebi. Nakon što je Model ažuriran, obavještava Controller komponentu koja zatim ažurira View komponentu za krajnjeg korisnika.



Slika 2: Prikaz MVC arhitekturalnog pristupa (The DotNet Guide, bez dat.)

Sada kad smo dobili detaljniji uvid u Model-View-Controller arhitekturalni pristup, gdje smo se upoznali s osnovnim komponentama ovog arhitekturalnog pristupa, možemo krenuti na prikaz prednosti i mana MVC arhitekturalnog pristupa. Prednosti i mane MVC arhitekturalnog pristupa bit će prikazane u dvije odvojene tablice. Nakon uvida u prednosti i mane ovog arhitekturalnog pristupa bit će prikazan dio kôda, odnosno vidjet ćemo kako MVC arhitekturalni pristup izgleda u praksi.

Ispod u tablici 1 imamo uvid u prednosti MVC arhitekturalnog pristupa.

Tablica 1: Prednosti MVC arhitekturalnog pristupa

Brži razvojni proces – MVC podržava brzi i paralelni pristup razvoju aplikacije. Moguće je da jedan programer radi na View dijelu, dok drugi može raditi na Controller dijelu. Na ovaj način aplikacija se može dovršiti tri puta brže nego koristeći druge arhitekturalne pristupe.

Koristeći MVC arhitekturalni pristup povećavamo testabilnost aplikacije i omogućuje lakšu implementaciju novih funkcionalnosti aplikacije.
Podrška za asinkronu tehniku – MVC aplikacije mogu raditi i s PDF datotekama, specifičnim preglednicima te desktop widgetima. MVC također podržava asinkronu tehniku koja pomaže programerima da razviju aplikacije koje se učitavaju jako brzo.
Modifikacija ne utječe na cijeli model – korisničko sučelje se jako često mijenja, no njegova promjena ne utječe na ostatak arhitekture.
Moguće je jedinično testiranje Modela i Controllera jer ne nasljeđuju niti koriste Android klase

Izvor: (Mishra, 2020; Mburu, 2018)

Nakon što smo vidjeli prednosti MVC arhitekturalnog pristupa, moramo vidjeti i onu drugu stranu, odnosno same mane MVC arhitekturalnog pristupa koje su prikazane u nastavku u tablici 2.

Tablica 2: Mane MVC arhitekturalnog pristupa

Slojevi kôda ovise jedni o drugima čak i ako se MVC pravilno primjeni.
Nema parametara za rukovanje logikom grafičkog sučelja, npr. kako da prikaže podatke
Povećana kompleksnost

Izvor: (Mishra, 2020; Mburu, 2018)

Nakon što smo dobili uvid u prednosti i mane MVC arhitekturalnog pristupa možemo donijeti zaključak da postoji mnogo više prednosti nego mana kod korištenja MVC arhitekturalnog pristupa. Neke od prednosti uključuju brži razvojni proces, zbog mogućnosti paralelnog pristupa razvoju aplikacije. Naravno jedna od prednosti je i povećana testabilnost aplikacije, što je i glavna prednost praćenja određenog arhitekturalnog pristupa. Jedna od ne velikog broja je povećana kompleksnost kod praćenja MVC arhitekturalnog pristupa, no to je i očekivano i normalna stvar.

U nastavku ćemo vidjeti dio programskog kôda aplikacije koja prati MVC arhitekturalni pristup. Prikazano je na primjeru TicTacToe aplikacije. Imamo tri komponente, Model, View i Controller.

Evo kako izgledaju komponente TicTacToe aplikacija sa svojim klasama od kojih svaka ima svoju ulogu.

Slika 3: TicTacToe - Model-View-Controller (Maxwell, 2017)

Nadalje, progarmski kôd u nastavku prikazuje programski kôd Controllera.

```
public class TicTacToeActivity extends AppCompatActivity {

    private static String TAG = TicTacToeActivity.class.getName();

    private Board model;

    private ViewGroup buttonGrid;
    private View winnerPlayerViewGroup;
    private TextView winnerPlayerLabel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tictactoe);
        winnerPlayerLabel = (TextView)
findViewById(R.id.winnerPlayerLabel);
        winnerPlayerViewGroup = findViewById(R.id.winnerPlayerViewGroup);
```

```
        buttonGrid = (ViewGroup) findViewById(R.id.buttonGrid);

        model = new Board();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.menu_tictactoe, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action_reset:
                reset();
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

    public void onCellClicked(View v) {

        Button button = (Button) v;

        String tag = button.getTag().toString();
        int row = Integer.valueOf(tag.substring(0,1));
        int col = Integer.valueOf(tag.substring(1,2));
        Log.i(TAG, "Click Row: [" + row + "," + col + "]");

        Player playerThatMoved = model.mark(row, col);

        if(playerThatMoved != null) {
            button.setText(playerThatMoved.toString());
            if (model.getWinner() != null) {
                winnerPlayerLabel.setText(playerThatMoved.toString());
                winnerPlayerViewGroup.setVisibility(View.VISIBLE);
            }
        }

    }

    private void reset() {
        winnerPlayerViewGroup.setVisibility(View.GONE);
        winnerPlayerLabel.setText("");

        model.restart();

        for( int i = 0; i < buttonGrid.getChildCount(); i++ ) {
            ((Button) buttonGrid.getChildAt(i)).setText("");
        }
    }

}
```

Isječak kôda 1: Prikaz Controllera (Maxwell, 2017)

U isječku kôda vidimo da su buttonGrid, winnerPlayerViewGroup i winnerPlayerLabel View komponente referencirane od strane Controllera. Unutar onCreate metode tražimo i zadržavamo reference na View komponente. Kada nam View kaže da je ćelija kliknuta, metoda onCellClicked će se pozvati. Na osnovu toga ažuriramo Model, a zatim ispitujemo njegovo stanje kako bismo odlučili što dalje. Ako je X igrač ili O igrač pobijedio ovim potezom, ažurira se View za prikaz pobjednika, ako nije, označava se ćelija koja je kliknuta. Prilikom resetiranja brišemo labelu koja prikazuje pobjednika i skrivamo je. Također govorimo Modelu da resetira svoje stanje.

Prema (Maxwell, 2017) MVC odlično odvaja Model od View komponente. Model se može lako testirati jer nije vezan ni za što, kod Viewa se nema što posebno testirati, no Controller ima nekoliko problema. Controller je toliko čvrsto vezan za Android API-je da je teško izvesti jedinično testiranje. Controller je čvrsto povezan s Viewom, ako promijenimo View moramo se vratiti i promijeniti Controller. Zadnji problem je održavanje. Vremenom sve se više kôda počinje prenositi u Controller, čineći ga ogromnim.

Upravo zbog ovih problema došlo je do pojave MVP arhitekturalnog pristupa koji slijedi u nastavku.

4.2. MVP

Softverski dizajn Model-View-Presenter nastao je početkom 1990-ih u Taligentu, u zajedničkom pothvatu Applea, IBMa i Hewlett-Packarda. MVP je osnovni programski model za razvoj aplikacija u Taligentovom CommonPoint okruženju zasnovanom na C++. Dizajn je kasnije Taligent preselio na Javu te popularizirao u radu Taligent CTO Mike Potela (Model-View-Presenter, 2021).

Nakon što je Taligent prestao s radom 1998., Andy Bower i Blair McGlashan iz tvrtke Dolphin Smalltalk prilagodili su uzorak MVPa kako bi bio osnova njihovog Smalltalk korisničkog sučelja. Microsoft je 2006. godine počeo uključivati MVP u svoju dokumentaciju i primjere programiranja korisničkog sučelja u .NET Frameworku (Model-View-Presenter, 2021).

Kada se krenu razvijati kompleksnije aplikacije način strukturiranja kôda postaje sve važniji i važniji. Mnogi tutorijali ignoriraju strukturiranje kôda, što rezultira prevelikom količinom kôda u Aktivnostima ili Fragmentima. MVP je nastao kao nedostatak MVCa. Naime, problem koji se javljao kod MVC arhitekturalnog pristupa bilo je pretrpavanje Controller komponente odgovornošću, kao što su rukovanje poslovnom logikom, razni mrežni zahtjevi te

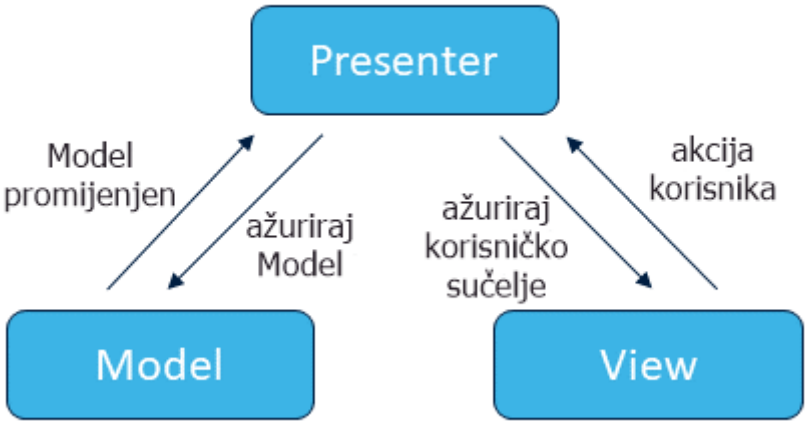
komuniciranje s bazom podataka. Prema (Mishra, 2020), MVP arhitekturalni pristup dolazi kao alternativa tradicionalnom arhitekturalnom pristupu MVC (Model-View-Controller). Jedan od problema s kojim su se programeri susretali prilikom korištenja MVC arhitekturalnog pristupa je taj da je se većina poslovne logike nalazi u Controlleru. Također, tijekom životnog vijeka aplikacije, ova datoteka raste i postaje jako teško održavati kôd. Uz to, View sloj i Controller sloj spadaju u istu Aktivnost ili Fragment što uzrokuje probleme prilikom mijenjanja aplikacije.

Problemi sa kojima su se programeri susretali prilikom korištenja MVC pristupa u implementaciji aplikacije, kod MVP pristupa ne postoje. Pruža modularnost, pojedino testiranje i čistiju bazu kôda. Sastoji se od tri komponente koje uključuju:

- Model - sloj za pohranu podataka. Odgovoran je za rukovanje logikom domene i komunikaciju s bazom podataka i mrežnim slojevima (Mishra, 2020).
- View - prema (Mishra, 2020) predstavlja sloj korisničkog sučelja (user interface). Omogućuje vizualizaciju i prikaz podataka te prati radnje korisnika kako bi obavijestio Presentera.
- Presenter - dohvaća podatke iz Modela i primjenjuje logiku korisničkog sučelja da odluči što će prikazati (Mishra, 2020).

Prema (Maxwell, 2017) MVP razdvaja Controller komponentu tako da može doći do povezivanja pogleda/aktivnosti bez vezanja za ostale odgovornosti Controllera. Model je ostao isti u odnosu na MVC arhitekturalni pristup. Kod Viewa, jedina promjena je što se sada aktivnost/fragment smatra dijelom Viewa. Prestajemo se boriti protiv prirodne tendencije da idu ruku pod ruku. Dobra je praksa da aktivnost implementira sučelje prikaza. Presenter komponenta je zapravo Controller komponenta iz MVCa, osim što uopće nije vezan za View, već samo za sučelje. Ovo rješava probleme vezane za testabilnost kao i zabrinutost oko modularnosti/fleksibilnosti koje smo imali s MVCom.

U nastavku je prikazan izgled Model-View-Presenter arhitekturalnog pristupa kao i veze između pojedinog sloja arhitekture.



Slika 4: Prikaz MVP arhitekturalnog pristupa (TouchGFX, bez dat.)

Iz slike vidimo da akokorisnik izvrši neku akciju, View komponenta šalje tu akciju Presenter komponenti. Na osnovu akcije Presenter komponenta obavlja poslovnu logiku vezanu za tu akciju te ažurira Model po potrebi. Model također obavještava Presentera da je došlo do promjene u Modelu. Presenter na osnovu akcije korisnika i poslovne logike ažurira korisničko sučelje.

Ovakav pristup rukuje interakcijama između View i Presenter sloja na drugačiji način nego kod MVC pristupa. Programeri koriste različite varijacije MVP pristupa, no najpoznatiji je definiranje Contract interfacea između Presentera i Viewa. Prilikom akcije korisnika, View poziva metode smještene u Presenter interfaceu. Isto tako Presenter poziva View da mu kaže koje podatke da prikaže iz Modela. Važno je napomenuti da jedna Presenter klasa odgovara isključivo jednoj View klasi, odnosno veza između Presentera i Viewa je jedan naprema jedan. Kao i kod MVCa, Model i View ne znaju za postojanje jedno drugog.

Sada kad smo dobili detaljniji uvid u Model-View-Presenter arhitekturalni pristup, gdje smo se upoznali s osnovnim komponentama ovog arhitekturalnog pristupa, možemo krenuti na prikaz prednosti i mana MVP arhitekturalnog pristupa. Prednosti i mane MVP arhitekturalnog pristupa bit će prikazane u dvije odvojene tablice. Nakon uvida u prednosti i mane ovog arhitekturalnog pristupa bit će prikazan dio kôda, odnosno vidjet ćemo kako MVP arhitekturalni pristup izgleda u praksi.

Ispod u tablici 3 vidimo koje su prednosti MVP arhitekturalnog pristupa.

Tablica 3: Prednosti MVP arhitekturalnog pristupa

Poslovna logika i logika korisničkog sučelja su odvojene jedna od druge.
Nema konceptualnog odnosa u Android komponentama.
Jednostavno održavanje kôda i testiranje – zbog toga što su Model, View i Presentation slojevi međuosobno odvojeni jedan od drugog
Mogućnost ponovne upotrebe kôda – zbog odvajanja odgovornosti.
Izolirana implementacija omogućuje testiranje svake komponente zasebno, pogotovo je to slučaj kod MVP arhitekturalnog pristupa koji koristi sučelje (eng. interface) za prikaz.

Izvor: (Mishra, 2020; Svirca, 2020)

Nakon što smo vidjeli prednosti MVP arhitekturalnog pristupa, moramo vidjeti i onu drugu stranu, odnosno same mane MVP arhitekturalnog pristupa koje su prikazane u nastavku u tablici 4.

Tablica 4: Nedostatci MVP arhitekturalnog pristupa

Ako programer ne slijedi načelo odgovornosti pojedinih slojeva, Presenter sloj se često pretvori u ogromnu klasu u kojoj je sve smješteno.
Povećana kompleksnost.
Nije idealno riješenje za jednostavne i male aplikacije.

Izvor: (Mishra, 2020; Svirca, 2020)

Nakon što smo dobili uvid u prednosti i mane MVP arhitekturalnog pristupa možemo donijeti zaključak da MVP pristup ima svojih prednosti ali i mana. Neke od prednosti uključuju to što su poslovna logika i logika korisničkog sučelja odvojene jedna od druge kao i mogućnost ponovne upotrebe kôda. Mana je da s MVP arhitekturalnim pristupom dolazi i povećana kompleksnost te također nije idealan izbor za jednostavne i male aplikacije.

Isto kao u prethodnom potpoglavlju, u nastavku ćemo vidjeti dio programskog kôda aplikacije koja prati MVP arhitekturalni pristup. Prikazano je na primjeru TicTacToe aplikacije.

Pogledajmo kako to izgleda u aplikaciji.

Slika 5: TicTacToe - Model-View-Presenter (Maxwell, 2017)

Pogledajmo dio kôda koji se odnosi na Presenter komponentu. Primjećujemo koliko je jednostavnija i jasnija namjera svake radnje. Umjesto da kaže Viewu kako nešto prikazati, samo mu govori što treba prikazati.

```
public class TicTacToePresenter implements Presenter {

    private TicTacToeView view;
    private Board model;

    public TicTacToePresenter(TicTacToeView view) {
        this.view = view;
        this.model = new Board();
    }

    @Override
    public void onCreate() {
        model = new Board();
```

```
    }

    @Override
    public void onPause() {

    }

    @Override
    public void onResume() {

    }

    @Override
    public void onDestroy() {

    }

    public void onButtonSelected(int row, int col) {
        Player playerThatMoved = model.mark(row, col);

        if(playerThatMoved != null) {
            view.setButtonText(row, col, playerThatMoved.toString());

            if (model.getWinner() != null) {
                view.showWinner(playerThatMoved.toString());
            }
        }
    }

    public void onResetSelected() {
        view.clearWinnerDisplay();
        view.clearButtons();
        model.restart();
    }


}
```

Isječak kôda 2: Prikaz Presentera (Maxwell, 2017)

U isječku kôda 2 možemo vidjeti da su metode onCreate(), onPause, onResume(), onDestroy() definirane u Presenter sučelju koje implementiramo. Kada korisnik odabere ćeliju, naš Presenter samo „čuje“ ono što je pritisnuto (red/stupac) te je na Viewu da utvrdi što je točno pritisnuto. Kada trebamo resetirati aplikaciju, samo diktiramo što treba učiniti.

Ovo je mnogo čišće i jasnije rješenje. Prema (Maxwell, 2017) lakše je jedinično testirati logiku Presentera jer nije vezana za nikakve Android API-je, a to nam također omogućuje rad s bilo kojim drugim Viewom sve dok View implementira TicTacToeView sučelje. Naravno, kao i kod MVC, tako i kod MVP arhitekturalnog pristupa postoje određeni problemi. Održavanje –

Presenter, baš kao i Controller, s vremenom su skloni prikupljanju dodatne poslovne logike. MVVM arhitekturalni pristup, opisan u nastavku, može pomoći u rješavanju ovog problema.

4.3. MVVM

Prema (Model-View-ViewModel, 2021) MVVM (skraćeno od Model-View-ViewModel) je softverski arhitekturalni dizajn koji olakšava odvajanje razvoja grafičkog korisničkog sučelja (View) od razvoja poslovne logike i od back-end logike, tako da View ne ovisi o ničemu. ViewModel je odgovoran za pretvaranje podatkovnih objekata iz Modela na način da se tim objektima lako upravlja i prezentira ih.

Izumili su ga Microsoftovi arhitekti Ken Cooper i Ted Peters posebno kako bi pojednostavili programiranje korisničkih sučelja na temelju događaja. John Gossman, jedan od Microsoftovih arhitekata, najavio je MVVM na svom blogu 2005. godine (Model-View-ViewModel, 2021).

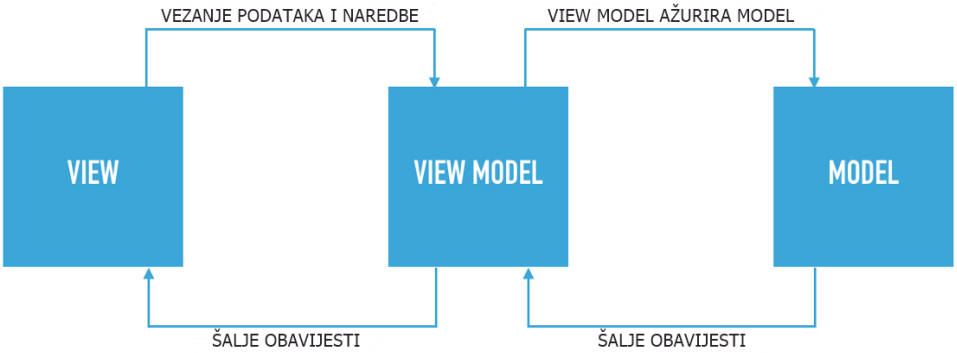
Pojavom MVVM arhitekturalnog pristupa uklonjene su čvrste veze između svake komponente, kako je to bio slučaj kod MVC i MVP arhitekturalnog pristupa. Kod MVVM arhitekturalnog pristupa djeca nemaju izravnu referencu prema roditelju, već imaju samo referencu prema roditelju putem „observables“. MVVM predlaže odvajanje prezentacijske logike od poslovne logike aplikacije.

MVVM arhitekturalni pristup sastoji se od sljedećih komponenti:

- Model – unutar njega je smještena poslovna logika, lokalna i/ili vanjska baza podataka i spremište (eng. Repository). Zadaća spremišta je da komunicira s lokalnom ili vanjskom bazom podataka prema zahtjevu ViewModela.
- View – ne sadrži poslovnu logiku, nego samo interakciju s korisnikom. Njegova svrha postojanja je obavijestiti ViewModel o radnji korisnika.
- ViewModel – služi kao poveznica između Modela i Viewa. Nema direktnu poveznicu prema Viewu, iz toga razloga ne zna koji ga View treba koristit.

Ima prednosti poput lakšeg testiranja i modularnosti na odnosu na gores pomenute arhitekturalne pristupe, odnosno MVC i MVP arhitekturalni pristup. Istovremeno smanjuje količinu kôda koja je potrebna da bi povezali View i Model. Prema (Maxwell, 2017), Model se ne mijenja u odnosu na MVC i MVP arhitekturalne pristupe. Pogled se veže za „observable“ varijable i radnje izložene od strane ViewModela. ViewModel je odgovoran za pripremu „observable“ podataka potrebnih za View, te je bitno napomenuti da nije vezan za View.

Detaljnije veze između pojedinih komponenti mogu se vidjeti na slici koja se nalazi u nastavku.



Slika 6: Prikaz MVVM arhitekturalnog pristupa (Perera, 2019)

Iako je svaki arhitekturalni pristup, uključujući MVC, MVP i MVVM bolji od popularnog „spaghetti code“ koji predstavlja ne strukturirani kôda, MVVM je ipak najbolji izbor kad je u pitanju razvoj Android aplikacija.

Sada kad smo dobili detaljniji uvid u Model-View-ViewModel arhitekturalni pristup, gdje smo se upoznali s osnovnim komponentama ovog arhitekturalnog pristupa, možemo krenuti na prikaz prednosti i mana MVVM arhitekturalnog pristupa. Prednosti i mane MVVM arhitekturalnog pristupa bit će prikazane u dvije odvojene tablice. Nakon uvida u prednosti i mane ovog arhitekturalnog pristupa bit će prikazan dio kôda, odnosno vidjet ćemo kako MVVM arhitekturalni pristup izgleda u praksi.

Ispod u tablici 5 vidimo koje su prednosti MVVM arhitekturalnog pristupa.

Tablica 5: Prednosti MVVM arhitekturalnog pristupa

Nema čvrste veze između View i ViewModel komponenti.
Nema sučelja između View komponente i Model komponente.

Jednostavno jedinično testiranje i kôd je „event-driven“, odnosno vođen događajima.
Kôd je moguće iznova koristiti više puta.
Omogućuje lakše održavanje projektnih datoteka i jednostavne izmjene.

Izvor: (Mishra, 2021).

Nakon što smo vidjeli prednosti MVVM arhitekturalnog pristupa, moramo vidjeti i onu drugu stranu, odnosno same mane MVVM arhitekturalnog pristupa koje su prikazane u nastavku u tablici 6.

Tablica 6: Nedostatci MVVM arhitekturalnog pristupa

Nije idealno riješenje za male projekte.
Ako je logika povezivanja podataka previše složena, otklanjanje pogrešaka (eng. debugging) u aplikaciji će biti teže.
Veličina kôda je poprilično velika.
Potrebno je kreirati „observables“ za svaku komponentu korisničkog sučelja.

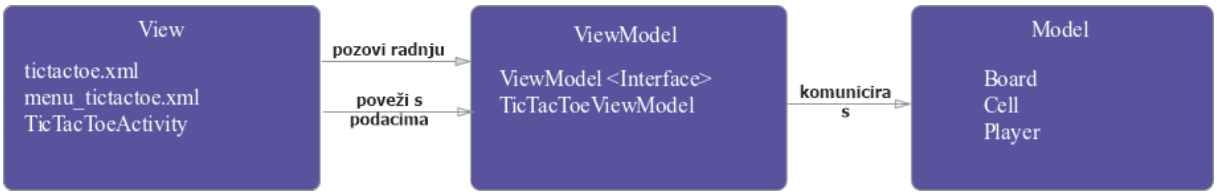
Izvor: (Mishra, 2021).

Nakon što smo dobili uvid u prednosti i mane MVVM arhitekturalnog pristupa možemo donijeti zaključak da MVVM arhitekturalni pristup ima svojih prednosti ali i mana. Neke od prednosti uključuju to što nema čvrste veze između View i ViewModel komponente kao i jednostavno jedinično testiranje. Također, omogućuje lakše održavanje projektnih datoteka i jednostavne

izmjene. S druge strane, mana je ta što nije idealan za male projekte te je veličina kôda poprilično velika.

Isto kao u prethodna dva potpoglavlja, u nastavku ćemo vidjeti dio programskog kôda aplikacije koja prati MVP arhitekturalni pristup. Prikazano je na primjeru TicTacToe aplikacije.

U nastavku su prikazane komponente aplikacije.



Slika 7: TicTacToe - Model-View-ViewModel (Maxwell, 2017)

Pogledajmo kako izgleda ViewModel u ovom slučaju.

```
public class TicTacToeViewModel implements ViewModel {

    private Board model;

    public final ObservableArrayMap<String, String> cells = new
ObservableArrayMap<>();
    public final ObservableField<String> winner = new ObservableField<>();

    public TicTacToeViewModel() {
        model = new Board();
    }

    @Override
    public void onCreate() {

    }

    @Override
    public void onPause() {

    }

    @Override
    public void onResume() {

    }

    @Override
    public void onDestroy() {
```

```
    }

    public void onResetSelected() {
        model.restart();
        winner.set(null);
        cells.clear();
    }

    public void onClickedCellAt(int row, int col) {
        Player playerThatMoved = model.mark(row, col);
        cells.put("" + row + col, playerThatMoved == null ? null :
playerThatMoved.toString());
        winner.set(model.getWinner() == null ? null :
model.getWinner().toString());
    }

}
```

Isječak kôda 3: Prikaz ViewModela (Maxwell, 2017)

U isječku kôda 3 susrećemo se sa „observable“ varijablama. Predstavljaju varijable koje će ViewModel ažurirati prema potrebi. View komponente su vezane izravno za te objekte i odmah reagiraju na promjene, bez da im ViewModel to mora reći. View po potrebi poziva metodu onClickedCellAt(). Ova metoda prenosi poruku Modelu o ćeliju koja je kliknuta, a zatim ažurira polja trenutnim stanjem Modela. Metoda onResetSelected() prenosi poruku Modelu za ponovno pokretanje.

Iz gore navedenog možemo zaključiti da postoji nekoliko prednosti korištenja MVVM arhitekturalnog pristupa. Prema (Maxwell, 2017) jedinično testiranje je sada još lakše, jer stvarno ne ovisite o Viewu. Prilikom testiranja bitno je samo provjeriti jesu li „observable“ varijable pravilno postavljene kada se Model promjeni. Naravno postoje i neki problemi kod korištenja MVVM arhitekturalnog pristupa. Održavanje – Budući da se View može povezati i s varijablama i izrazima, postoji mogućnost uvlačenja kôda u naš XML. Da bi se to izbjeglo bitno je uvijek dohvaćati vrijednosti direktno iz ViewModela.

Ovim završavamo poglavlje koje se odnosi na tri arhitekturalna pristupa koja smo opisali u ovom završnom radu. Iz analize je vidljivo da su i MVP i MVVM bolji izbor od MVCa po pitanju modularnosti, no bitno je napomenuti da s njima dolazi i veća složenost naše aplikacije.

5. Usporedba arhitekturalnih pristupa

U ovom poglavlju prikazana je usporedba tri glavna arhitekturalna pristupa kada su u pitanju troslojna arhitektura i razvoj Android aplikacija. To su MVC, MVP i MVVM arhitekturalni pristupi. Sva tri arhitekturalna pristupa olakšavaju programerima razvoj aplikacija koje su jednostavne za testiranje i održavanje. MVP i MVVM su nastali iz MVC arhitekturalnog pristupa. Razlika između MVCa i arhitekturalnih pristupa nastalih od njega je ovisnost svakog sloja o drugim slojeva, kao i koliko su međusobno čvrsto povezani.

Kako bi znali ključne razlike između svakog od arhitekturalnih pristupa, u nastavku u tablici 7 prikazane su osnovne i ključne razlike između svakog od arhitekturalnih pristupa koje smo ranije detaljno obradili u ovom završnom radu.

Tablica 7: Usporedba arhitekturalnih pristupa

MVC – Model View Controller	**MVP – Model View Presenter**	**MVVM – Model View ViewModel**
Jedna od najstarijih softverskih arhitektura.	Razvijena kao druga iteracija softverske arhitekture koja je naprednija od MVCa.	Industrijski priznati arhitekturalni pristup za razvoj Android aplikacija.
View i Model komponente su čvrsto povezane.	Rješava problem ovisnosti View komponente korištenjem Presenter komponente kao komunikacijskog kanala između Modela i Viewa.	Ovaj arhitekturalni pristup više je vođen događajima (eng. event-driven) jer koristi vezivanje podataka (eng. data binding) te na taj način olakšava odvajanje osnovne poslovne logike od Viewa.

Controller i View komponente imaju odnos jedan naprema više. Jedan Controller može odabrati drugačiji prikaz na temelju potrebne operacije.	Odnos između Presenter i View komponente je jedan naprema jedan. Jedna klasa Presentera upravlja jednim Viewom.	Više Viewa se može mapirati s jednim ViewModelom i stoga postoji odnos jedan naprema više između Viewa i ViewModela.
View nema saznanja o Controlleru.	View ima referencu na Presenter.	View ima referencu na ViewModel.
Teško je mijenjati značajke aplikacije jer su slojevi kôda čvrsto povezani.	Slojevi su labavo povezani i stoga je lako izvršiti promjene u aplikacijskom kôdu.	Lako je napraviti izmjene u aplikaciji. Međutim, ukoliko je logika povezivanja podataka previše složena, bit će malo teže otkloniti pogreške u aplikaciji.
Controller rukuje ulazima korisnika.	View je ulazna točka aplikacije.	View uzima korisnički unos i djeluje kao ulazna točka aplikacije.
Idealan samo za male projekte.	Idealan za jednostavne i za složene aplikacije.	Nije idealan za male projekte.
Ograničena podrška za jedinično testiranje (eng. Unit testing).	Jednostavno jedinično testiranje, ali čvrsta veza između Viewa i Presentera može ga otežati.	Testabilnost jedinica najveća je kod ove arhitekture.
Ima veliku ovisnost o Android API-ima.	Ima nisku ovisnost o Android API-ima.	Ima mali ili gotovo nikavu ovisnost o Android API-ima.

Ne slijedi načelo modularne i jedinstvene odgovornosti.	Slijedi modularno načelo i načelo jedinstvene odgovornosti.	Slijedi modularno načelo i načelo jedinstvene odgovornosti.

Izvor: (Mishra, 2020).

MVC arhitekturalni pristup je zastarjeli pristup za izradu Android aplikacija. Danas se jako rijetko koristi. Iz gore prikazane tablice može se vidjeti niz mana koje postoje kod MVCa. Bitno je napomenuti da su MVP i MVVM arhitekturalni pristupi nastali kao nedostatak MVC arhitekturalnog pristupa. MVC pristup ima čvrsto povezane slojeve što jako otežava mijenjanje značajki aplikacije. Danas se u razvoju Android aplikacija najviše koristi MVVM arhitekturalni pristup te je također preporučen u sklopu Android Jetpacka. Za izradu aplikacije izabran je pristup MVVM arhitekturalni pristup, čiji će postupak biti prikazan u zadnjem poglavlju ovog završnog rada.

6. Android Jetpack

U ovom poglavlju upoznat ćemo se s Android Jetpackom. Upoznat ćemo se s osnovnim Android Jetpack komponentama. Također imat ćemo uvid u arhitekturalni pristup predložen u sklopu Android Jetpacka. Krenimo s definicijom Android Jetpacka.

Jetpack je zbirka biblioteka koja pomaže programerima slijediti najbolje prakse, smanjiti ponavljajući kôd i napisati kôd koji dosljedno radi na svim Android verzijama i uređajima, tako da se programeri mogu usredotočiti na kôd do kojeg im je stalo (Android Developers, 2019).

Prema (Guarav, 2019) Android Jetpack je zbirka Android komponenti koja nam pomaže u izgradnji izvrsnih Android aplikacija. Ove komponente pomažu na način da slijede najbolje prakse, smanjuju ponavaljući kôd te čine složene stvari vrlo jednostavnima. Prije pojavljivanja Android Jetpacka bilo je dosta izazova kao što su upravljanje životnim ciklusima aktivnosti, preživljavanje promjena konfiguracije te sprječavanje curenja memorije. Svi ovi izazovi nestali su pojavom Android Jetpacka. Još jedna važna stvar stvar kod Jetpacka je da se ažurira češće od Android platforme, tako da uvijek dobivamo najnoviju verziju. Jetpack sadrži biblioteke paketa androidx.*, odvojene od API-ja platforme. Android Jetpack komponente su zbirka biblioteka koje se pojedinačno mogu usvojiti i izrađene su za zajednički rad uz iskorištavanje jezičnih značajki Kotlina koje nas čine produktivnijima. Komponente su raspoređene u četiri kategorije:

- Temeljne komponente (eng. Foundation components)
- Arhitekturalne komponente (eng. Architecture components)
- Komponente ponašanja (eng. Behavior components)
- Komponente korisničkog sučelja (eng. UI components)

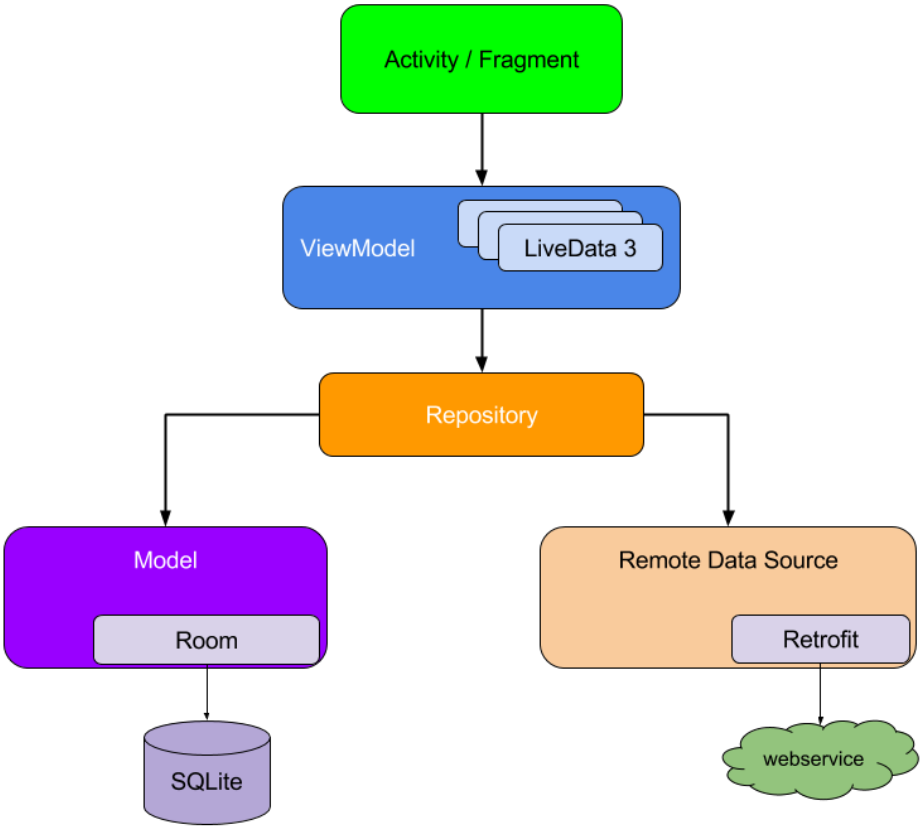
Pogledajmo što koja kategorija predstavlja. Prema (Gurav, 2019) temeljne komponente pružaju kompatibilnost unatrag (eng. Backward compatibility), testiranje, podršku za Kotlin programski jezik. U temljne komponente spadaju App Compat, Android KTX koji predstavlja skup Kotlinovih proširenja za pisanje sažetijeg Kotlinovog kôda, Multidex i Test koji predstavlja okvir za testiranje korisničkog sučelja u Androidu. Arhitekturalne komponente nam pomažu u izgradnji robusnih aplikacija, aplikacija koje se mogu testirati i aplikacija koje se mogu održavati. Arhitekturalne komponente čine: Data Binding koja nam omogućuje „vezanje“ elementa korisničkog sučelja za izvore podataka naše aplikacije. Lifecycles – upravlja životnim ciklusima aktivnosti i fragmenata. LiveData – obavještava prikaze o svim promjenama

baze podataka. Navigacija – rukuje svime što je potrebno za navigaciju unutar aplikacije. Straničenje – postupno učitava informacije na zahtjev izvora podatka. Room – tečan pristup SQL bazi podataka. Korišten je i u sedmom poglavlju prilikom implementacije aplikacije. ViewModel – upravlja podatcima povezanim s korisničkim sučeljem na način da je svjestan životnog ciklusa. WorkManager – mogućnost upravljanja pozadinskim procesima u Androidu prema okolnostima koje odaberemo. Komponente ponašanja nam pomažu u integraciji sa standardnim Android uslugama poput obavjesti, dopuštenja, dijeljenja. U njih spadaju upravitelj preuzimanja – koji služi za upravljanje velikim preuzimanjima u pozadini. Mediji i reprodukcija – API-ji za reprodukciju i usmjeravanje medija. Obavijesti – pruža unatrag kompatibilan API za obavijesti s podrškom za Wear i Auto. Dopuštenja – API-ji za provjeru i traženje dopuštenja u aplikaciji. Komponente korisničkog sučelja pružaju dodatke i pomagače koji vašu aplikaciju čine ne samo jednostavnom, već i ugodnom za korištenje. Komponente korisničkog sučelja su sljedeće: Animacije i prijelazi – premještanje dodataka i prijelaz između zaslona. Auto – komponente za razvoj aplikacija za Android Auto. Emoji – omogućavanje ažuriranog fonta emojia na starijim Android platformama. TV – komponente za razvoj Android TV aplikacija. Wear – komponente za razvoj Wear aplikacija.

S ovim dijelom završili smo s Android Jetpack komponentama. Prije nego što se upoznamo s arhitekturalnim pristupom preporučenim u sklopu Android Jetpacka, pogledajmo kakva arhitekturalna načela se preporučuju.

Prema (Developer Android, 2021c) najvažnije načelo koje treba slijediti je razdvajanje zaduženja. Najčešća greška je pisanje cijelog kôda unutar aktivnosti (eng. Activity) ili fragmenta (eng. Fragment). To su klase za prikaz korisničkog sučelja, prema tome trebale bi sadržavati samo logiku koja je vezana za interakcije s korisničkim sučeljem, odnosno korisnikom. Drugi važan princip koji treba poštovati je taj da korisničko sučelje treba biti pokrenuto iz modela, po mogućnosti trajnog modela. Modeli predstavljaju komponente koje su odgovorne za rukovanje podatcima za aplikaciju. Oni su neovisni o View objektima i komponentama aplikacije pa na njih ne utječu životni ciklus aplikacije. Postavlja se pitanje zašto je trajnost bitna. Trajnost, odnosno postojanost, bitna je jer korisnici vaše aplikacije ne gube podatke ako Android odluči ubiti aplikaciju kako bi oslobodio resurse. Također, temeljeći aplikaciju na klasama modela s jasno definiranom odgovornošću upravljanja podatcima, aplikacija je stabilnija i dosljednija.

Već kroz rad imali smo priliku čuti o preporučenom arhitekturalnom pristupu u sklopu Android Jetpacka. Pogledajmo kako izgleda i koje su komponente arhitekturalnog pristupa.



Slika 8: Preporučeni arhitekturalni pristup (Developer Android, 2021d).

Iz slike je vidljivo da svaka komponenta ovisi samo o komponenti koja je na razini ispod nje. Na primjer, Aktivnosti i Fragmenti ovise samo o ViewModelu. U nastavku ovog završnog rada, odnosno u poglavlju 7 upravo ovaj arhitekturalni pristup će biti korišten za implementaciju aplikacije.

Kad se spomene programiranje zna se da je to kreativno područje i da je „samo nebo granica“. Postoji mnogo načina za rješavanje pojedinog problema, kao i preporuke predložene u sklopu Android Jetpack najboljih praksi. One naravno nisu obavezne, ali se preporučuje pratiti prakse jer čine implementaciju kôda robusnijom, provjerljivijom i dugoročno održivom (Developer Android, 2021c).

- Izbjegavajte označavati ulazne točke svoje aplikacije, kao što su aktivnosti, usluge i prijemnici za emitiranje kao izvore podataka – umjesto toga oni bi samo trebali koordinirati s drugim komponentama kako bi dohvatili podskup podataka koji su relevantni za tu ulaznu točku (Developer Android, 2021c).

- Stvorite dobro definirane granice odgovornosti između različitih modula vaše aplikacije – na primjer, nemojte dio kôda koji učitava podatke s mreže dijeliti na više klasa.
- Izlazite što je moguće manje iz svakog modula (Developer Android, 2021c).
- Razmislite kako svaki modul učiniti zasebnim za testiranje – na primjer, dobro definiran API za dohvaćanje podataka s mreže olakšava testiranje modula koji te podatke zadržava u bazi podataka (Developer Android, 2021c).
- Usredotočite se na jedinstvenu jezgru aplikacije kako bi se izdvojila od drugih aplikacija – nemojte ponovno izmišljati kotač tako da uvijek iznova pišemo isti kôd. Umjesto toga, usredotočite svoje vrijeme i energiju na ono što vašu aplikaciju čini jedinstvenom i dopustite da komponentne Android Arhitekture i druge preporučene biblioteke upravljaju ponavljajućim kôdom (Developer Android, 2021c).
- Inzistirajte na što je više moguće relevantnih i svježih podataka – tako korisnici mogu uživati u funkcionalnostima vaše aplikacije čak i kad nisu spojeni na mrežu (Developer Android, 2021c).

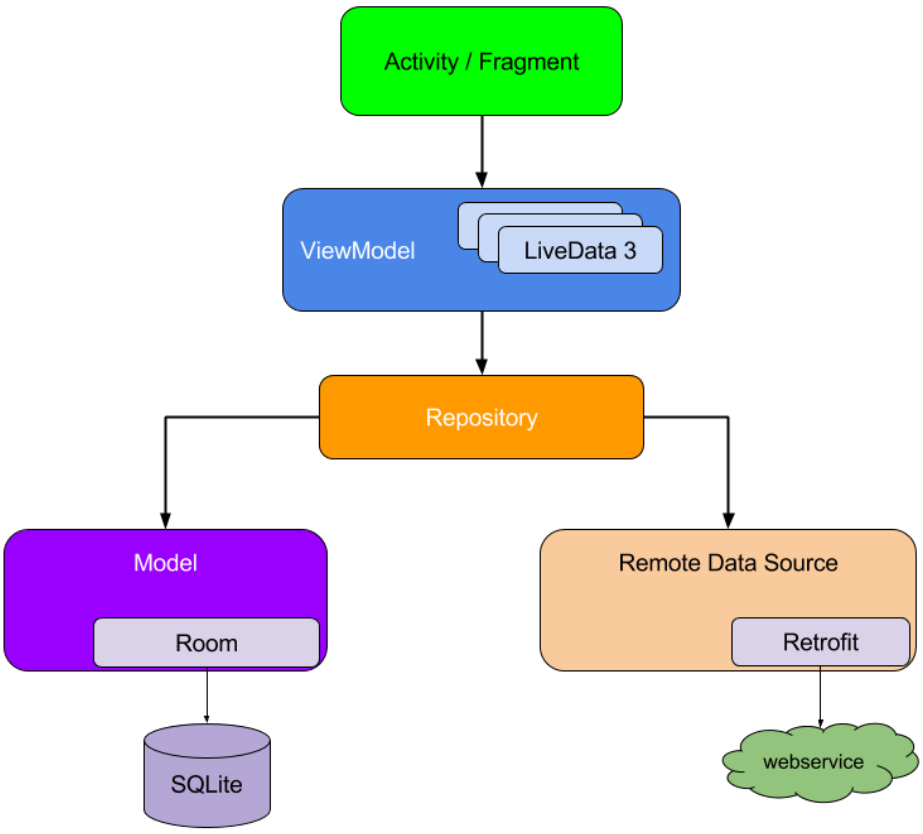
7. Implementacija aplikacije

U završnom dijelu ovog završnog rada implementirat će se aplikacija prateći arhitekturalni pristup predložen u sklopu Android Jetpacka, koji smo u prethodnom poglavlju imali priliku upoznati. Za implementaciju aplikacije korišten je Kotlin programski jezik. Sav kôd napisan je u Android Studiju. Android Studio je službeno razvojno okruženje (IDE – integrated development environment) za Googleov operativni sustav Android, izgrađen na JetBrains' InteliJ IDEA softveru i dizajniran posebno za razvoj Androida [2]. Važno je napomenuti da je tema ovog završnog rada Dizajn i implementacija troslojne arhitekture pri implementacij aplikacija za Android.

Prvo ćemo reći par riječi o dizajnu arhitekture, s kojim smo se već upoznali u trećem poglavlju ovog završnog rada, samo će se staviti fokus na praktični dio. Odnosno, dizajn arhitekture za aplikaciju koja čija će implementacija biti prikazana u nastavku ovog završnog rada. Prva stvar kod dizajna arhitekture je razmisliti o kontekstu aplikacije, odnosno odgovoriti na pitanje „Koje sve funkcionalnosti treba naša aplikacija obavljati?“ Da pokušamo pojednostavniti, trebamo razmisliti o tome što će sve krajnji korisnik moći raditi u aplikaciji? S obzirom na kompleksnost samog Model-View-ViewModel arhitekturalnog pristupa, nije bilo potrebno dodatno komplicirati stvari implementacijom kompleksnije aplikacije. Naša aplikacija ima par funkcionalnosti. Aplikacija služi za evidenciju studenata. Korisnik ima mogućnost pregleda studenata te dodavanja novog studenta na popis studenata. Dodatan plus naše aplikacije je taj što je korištena trajna baza podataka, odnosno podatci se spremaju u SQLite bazu podataka korištenjem Room Android Jetpack komponente, čija je implementacija prikazana u nastavku. To omogućuje korisniku da izađe iz aplikacije kad god poželi, bez brige o tome hoće li se zapisi obrisati. Prilikom dodavanja novog studenta na popis studenata, popis se automatski osvježava te korisnik odmah vidi dodanog studenata na postojećem popisu studenata. Prilikom dizajna sučelja stavljen je naglasak na jednostavnost i preglednost. Podatci o pojedinom studentu su čitki, vidljivi i zasebni, odnosno svaki zapis je odvojen jedan kako bi korisnik imao jasan uvid u podatke o pojedinom studentu. Što se tiče nefunkcionalnih zahtjeva aplikacije, možemo spomenuti portabilnost. Kako je tema ovog završnog rada Dizajn i implementacija troslojne arhitekture pri implementaciji aplikacije za Android, nismo se previše trebali baviti portabilnosti, s obzirom na to da je naša aplikacija izrađena isključivo za Android operativni sustav te će se na njemu pokretati. Ono što se tiče portabilnosti je da smo prilikom kreiranja Android Studio projekta odabrali minimalnu verziju Androida koja će biti potrebna da se pokrene aplikacija. U ovom slučaju odabrana je verzija Android 5.0 (Lolipop), opširnije o

ovome ćemo vidjeti u opisu slike 8 koja prikazuje konfiguraciju novog projekta unutar Android Studija. Također prilikom dizajna arhitekture vodili smo brigu o performansama aplikacije, odnosno vodili smo brigu da naša aplikacija brzo upisuje podatke u bazu podataka kao i iste ažurira na ekranu. Aplikacija je također lagana, pa nema problema s ogromnim korištenjem Android resursa koji su ograničeni. Prilikom dizajna arhitekture također je vođeno računa o upotrebljivosti aplikacije. Aplikacija je jako jednostavna za korištenje, s toga nije potrebna dodatna edukacija. Što se tiče ograničenja naše aplikacije, postoji jedno ograničenje, da slijedimo Model-View-ViewModel arhitekturalni pristup. Upravo ovo ograničenje povećalo je samu kompleksnost izrade ove aplikacije. S obzirom na to da smo imali dovoljno vremena za izradu cjelokupnog završnog rada, nije bilo potrebe birati prioritete, odnosno mogli smo sve zamišljeno implementirati.

Naglasak je stavljen na arhitekturu, ne na aplikaciju, iz tog razloga neće se praviti neka jako kompleksna aplikacija nego aplikacija koja će slijediti arhitekturalni pristup prikazan u nastavku:

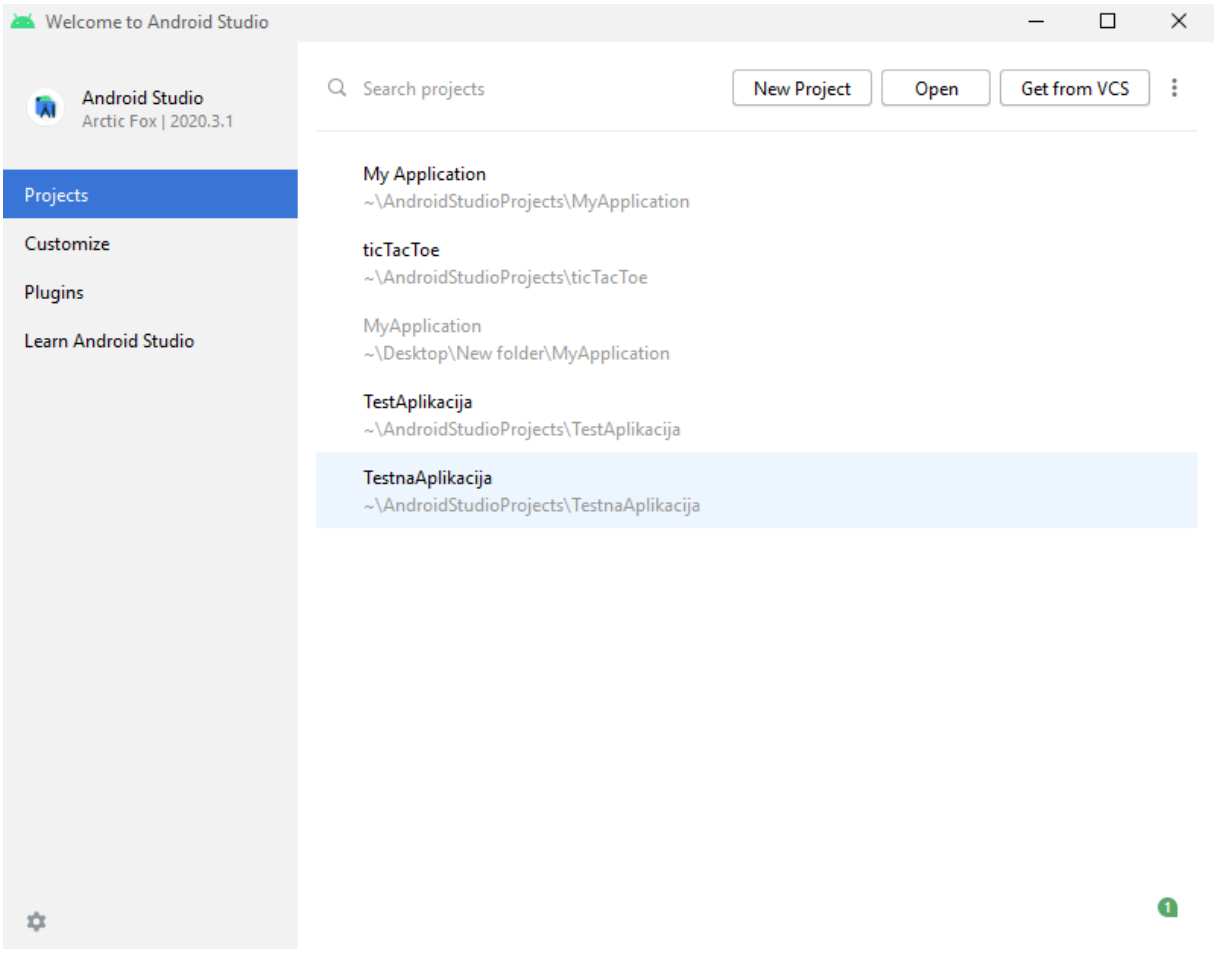


Slika 9: Arhitekturalni pristup za implementaciju aplikacije (Developer Android, 2021d.).

Iz slike vidimo da je ovo isti arhitekturalni pristup kao i MVVM arhitekturalni pristup. Kako se u izradi aplikacije ne koristiti neki API (Application programming interface), Remote Data Source neće trebati te ćemo ga izostaviti u implementaciji naše aplikacije. Ostatak komponenti će ostati isti.

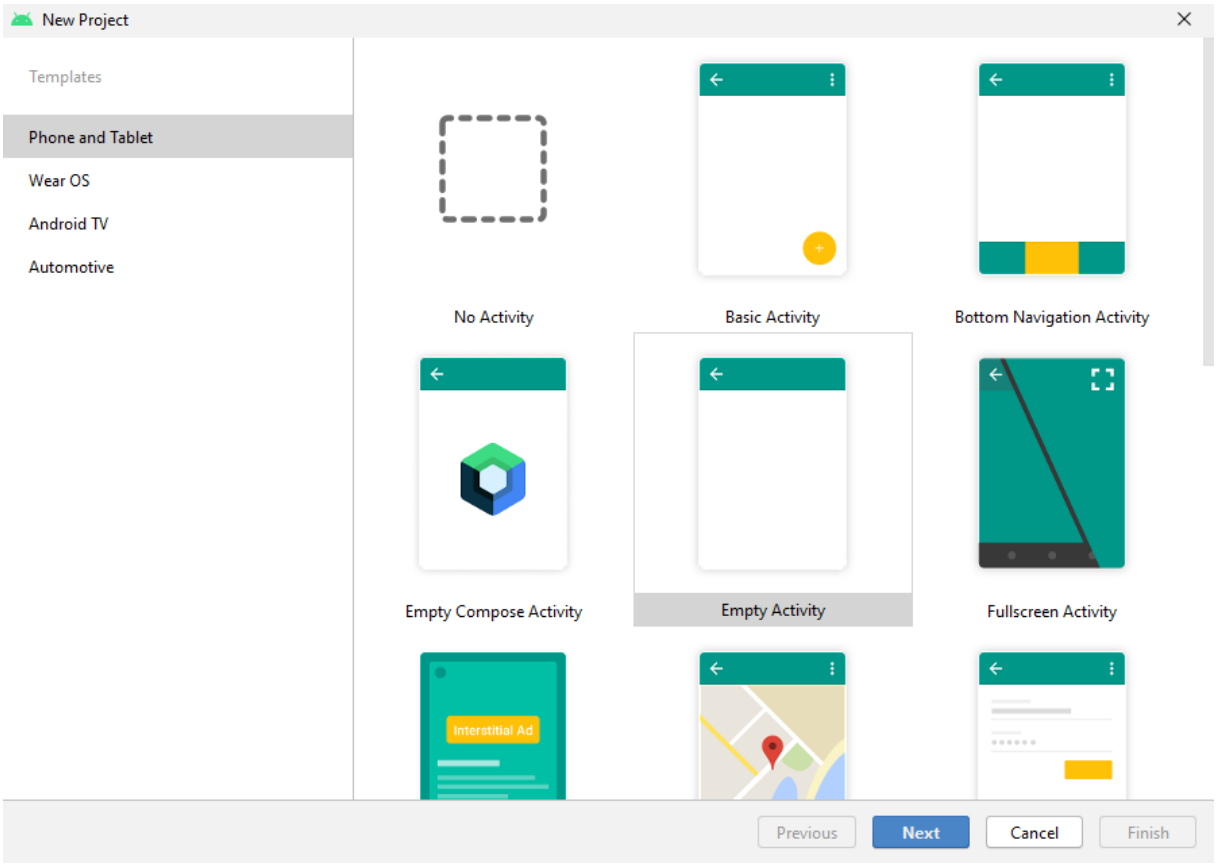
Kreće se od nule, odnosno prikazat će se svi koraci u izradi aplikacije točno onim redosljedom kojim je aplikacija implementirana.

Za izradu aplikacije i pisanje kôda korišten je Android Studio, odličan IDE u kojem postoji opcija izravnog testiranja aplikacije na vlastitom Android uređaju. Početni zaslon Android Studija prikazan je u slici ispod:



Slika 10: Android Studio Arctic Fox

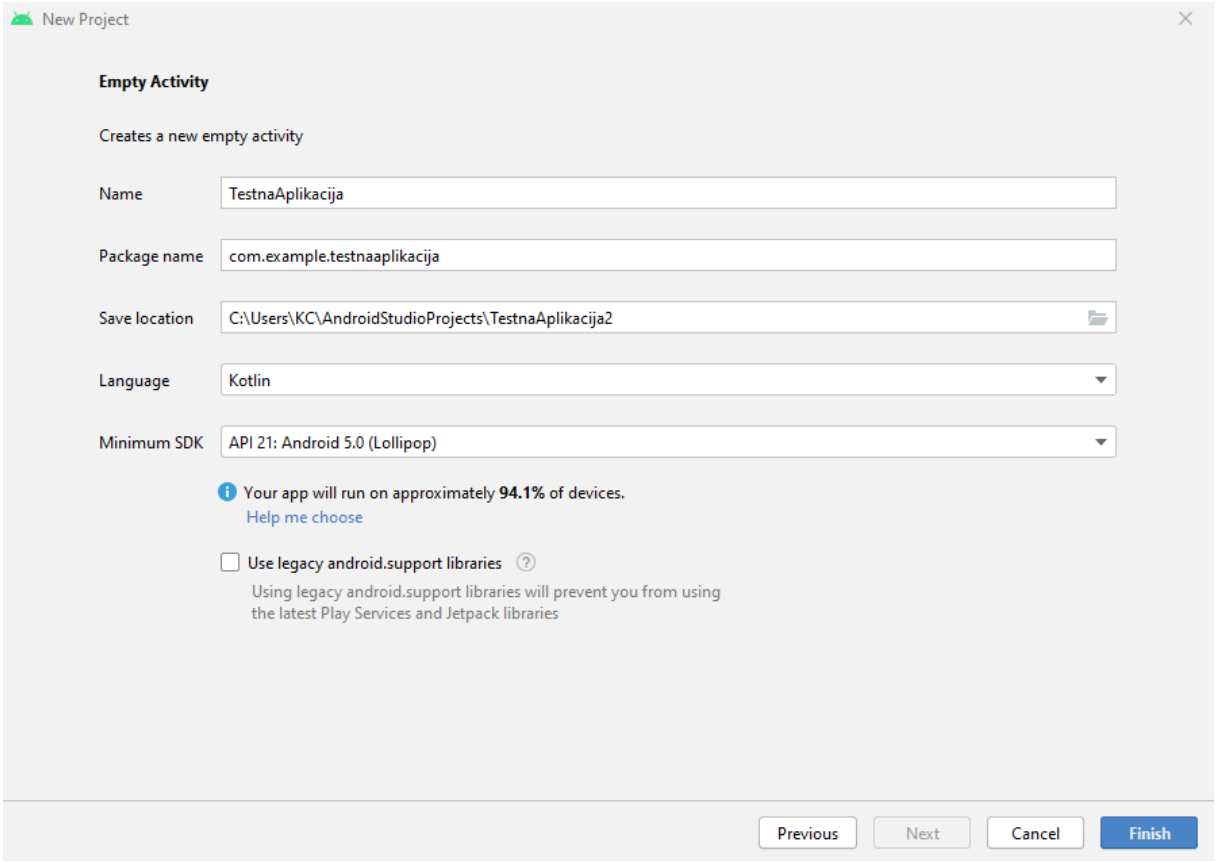
Klikom na Create New Project kreiramo novi projekt te se dolazi do sljedećeg zaslona prikazanom na slici 5 koja se nalazi u nastavku. Tu se odabire Empty Activity te klikom na Next ide se prema sljedećem prozoru:



Slika 11: Android Studio - kreiranje novog projekta

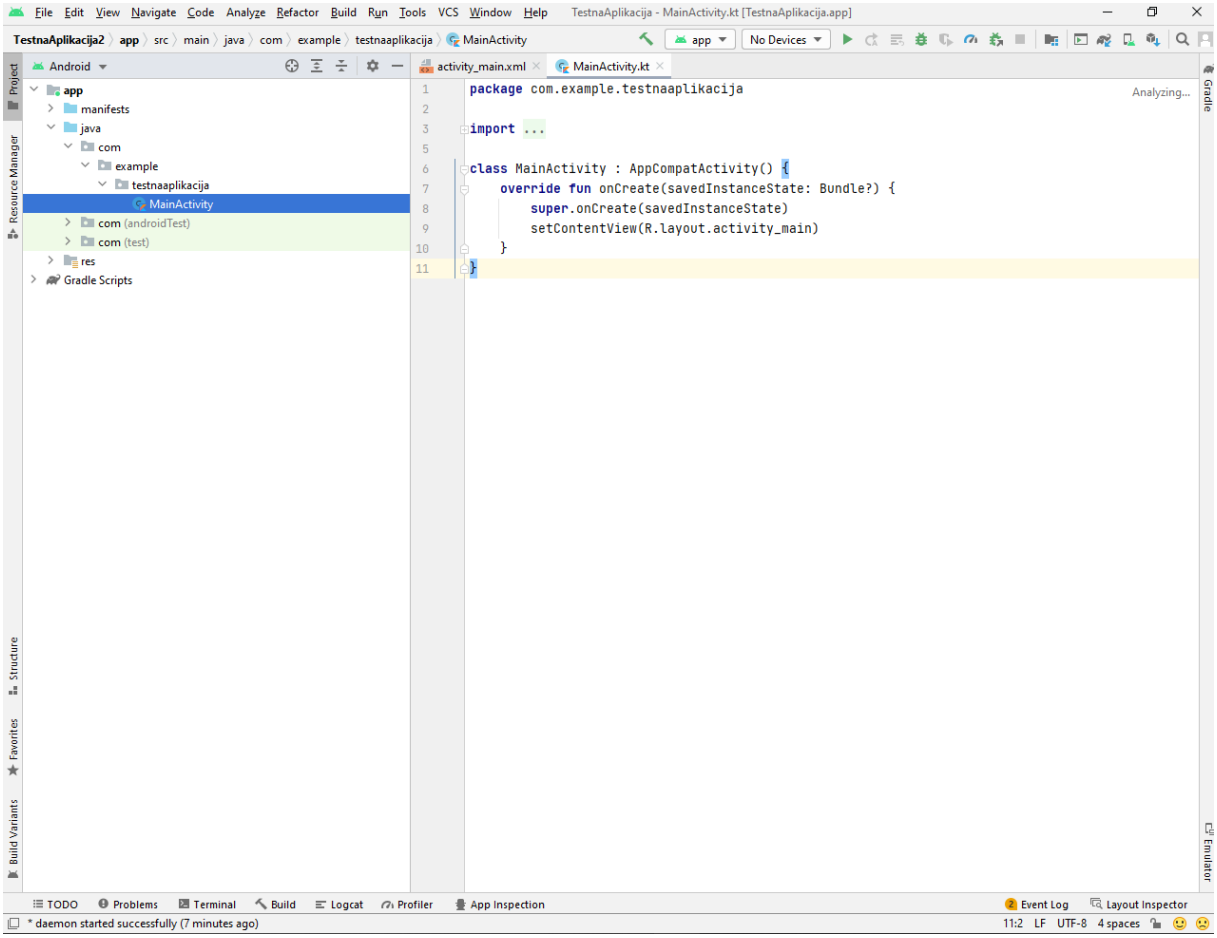
Dolazi se do završnog koraka pri kreiranju novog projekta. Odabiremo ime, u ovom slučaju aplikacije se zove „TestnaAplikacija“, package name u ovom slučaju nije bitno, jer se ova aplikacija neće izvoziti nigdje (npr. Google Play Store). Inače, package name bi trebalo biti unikatno, jer ako već postoji neka aplikacija s istim imenom aplikacija se neće moći izvesti i objaviti na Google Play Store. Nadalje, Save Location označava mjesto gdje će se aplikacija spremiti na disku, u ovom slučaju sprema se na C disk pod TestnaAplikacija. Kod opcije Language bira se programski jezik u kojem će se aplikacija pisati. Android Studio ima opciju odabira Jave ili Kotlin programskog jezika. U ovom slučaju odabran je Kotlin programski jezik. Zadnja stavka koja se treba odabrati prilikom kreiranja novog projekta je Minimum SDK

(software development kit), odnosno, minimalnu verziju Androida koja će biti potrebna da se pokrene ova aplikacija. U ovom slučaju odabrana je verzija Android 5.0 (Lolipop) te ispod vidimo podatak da će se aplikacija uspješno moći pokrenuti na oko 94.1% uređaja, što je jako zadovoljavajući podatak. Klikom na Finish završava se s kreiranjem novog projekta u Android Studiju.



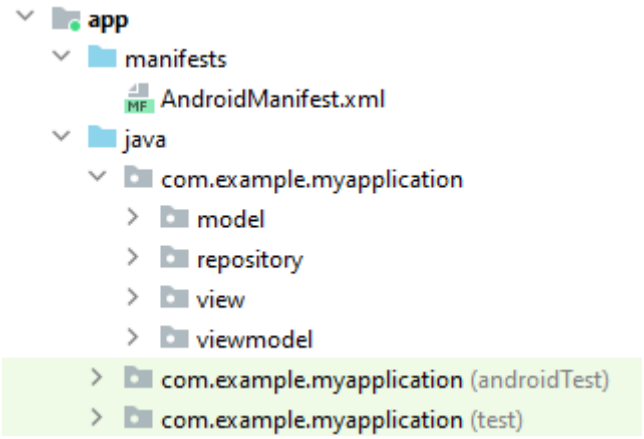
Slika 12: Android Studio - konfiguracija novog projekta

Ispod na slici 13 prikazan je početni zaslon kreiranog projekta. Android Studio je sam kreirao dosta toga, što možemo vidjeti detaljno na slici.



Slika 13: Android Studio - početak rada

Ono što se na početku treba napraviti je napraviti spremnike (eng. Package) kako bi pratili arhitekturalni pristup koji je izabran te odvajali klase i dokumente na svoje propisno mjesto. Na početku poglavlja bio je prikazan arhitekturalni pristup koji će se pratiti. Imat ćemo ukupno četiri spremnika: View (Activity/Fragment), ViewModel, Repository i Model.



Slika 14: Prikaz kreiranih spremnika

Za lakši rad s bazom podataka koristit ću Room. Room Database je dio Android Arhitekturalnih komponenti koji pruža sloj apstrakcije preko SQLitea koji omogućuje robusniji pristup bazi podataka a istovremeno pruža „punu snagu“ SQLitea. Room je dio Android Jetpacka. Room koristi anotacije kako bi izbjegli ponovno pisanje istog kôda. Također, provjerava SQL upite u vrijeme kompajliranja.

Prvo kreiramo podatkovnu klasu. Room kreira tablicu u bazi podataka za svaku klasu sa anotacijom @Entity. Ako napišemo samo @Entity, tablica u bazi podataka će imati isti naziv kao i ime klase. Možemo i ručno postaviti ime tablice korištenjem sintakse: @Entity (tableName = „željeno_ime“). Tablica Student ima svoje atribute: ime, prezime, godine i id koji je primarni ključ, koji definiramo anotacijom @PrimaryKey.

```
@Entity
data class Student(
    val ime: String,
    val prezime: String,
    val godine: Int,
    @PrimaryKey(autoGenerate = true) var id: Int = 0
)
```

Isječak kôda 4: Implementacija podatkovne klase.

Zatim kreiramo Dao (Data Acces Objects) koji je odgovoran za definiranje metoda koje pristupaju bazi podataka. Tu definiramo osnovne funkcionalnosti SQL baze podataka poput umetanja i brisanja unosa. Koristimo i @Query za dohvat svih Studenata iz baze.

```
@Dao
interface StudentDao {
    @Insert
    fun insertStudent(student: Student)

    @Delete
    fun deleteStudent(student: Student)

    @Query("SELECT * FROM Student")
    fun getAllStudents(): List<Student>
}
```

Isječak kôda 5: Implementacija @Dao

Kada je sve spremno za izradu baze podataka, kreiramo Room bazu podataka nasljeđivanjem RoomDatabase. Vidimo da je StudentDatabase apstraktna klasa koja nasljeđuje RoomDatabase te mora imati anotaciju @Database. Prima popis entiteta sa svim klasama koje čine bazu podataka, odnosno imaju anotaciju @Entity. Definiramo i verziju baze podataka, u ovom slučaju to je verzija 1.

```
@Database(entities = [Student::class], version = 1, exportSchema = false)
abstract class StudentDatabase : RoomDatabase() {

    abstract fun userDao() : StudentDao

    companion object{
        @Volatile
        private var instance:  StudentDatabase? = null
        private val LOCK = Any()

        operator fun invoke(context: Context) = instance ?:
synchronized(LOCK){
            instance ?: createDatabase(context).also{  instance = it }
        } //Singleton

        private fun createDatabase(context: Context) =
            Room.databaseBuilder(
                context.applicationContext,
                StudentDatabase::class.java,
                "studentdatabase"
            ).fallbackToDestructiveMigration().build()
    }
}
```

Isječak kôda 6: Implementacija Room baze podataka

Sad je naša Room baza podataka spremna za implementaciju CRUD operacija.

Kreiramo novu klasu StudentRepositoryOne unutar Repository spremnika. Tu implementiramo asinkrone funkcije za komunikaciju s Dao.

```
class StudentRepositoryOne(val db: StudentDatabase){
    suspend fun getAlLlStudents(): List<Student> {
        return withContext(Dispatchers.IO){
            db.userDao().getAllStudents()
        }
    }

    suspend fun insertStudent(student: Student) {
        withContext(Dispatchers.IO){
            db.userDao().insertStudent(student)
        }
    }

    suspend fun deleteStudent(student: Student) {
        withContext(Dispatchers.IO){
            db.userDao().deleteStudent(student)
        }
    }
}
```

Isječak kôda 7: Implementacija StudentRepositoryOne klase.

Dolazimo do implementacije ViewModela. Kreiramo novu klasu StudentViewModel koju smještamo u ViewModel spremnik. StudentViewModel nasljeđuje ViewModel te kao parametar prima StudentRepositoryOne. U kôdu je vidljivo da palimo korutine koje pozivaju funkcije smještene u repozitoriju te ih izvršava.

```
class StudentViewModel(private val repository: StudentRepositoryOne):
ViewModel() {

    val studentList = MutableLiveData<List<Student>>()

    fun getAllStudents(){
        viewModelScope.launch {
            studentList.value = repository.getAlLlStudents()
        }
    }

    fun deleteStudent(student: Student){
        viewModelScope.launch {
            repository.deleteStudent(student)
        }
    }

    fun insertStudent(student: Student){
        viewModelScope.launch {
            repository.insertStudent(student)
        }
    }


}
```

Isječak kôda 8: Implementacija ViewModela.

Također, imat ćemo još jednu klasu koja služi za kreiranje instanci ViewModela. Nazvat ćemo je StudentFactory. Ona kao parametar prima StudentRepositoryOne te nasljeđuje ViewModeProvider.Factory.

```
class RateRestaurantViewModelFactory(private val repository:
StudentRepositoryOne) : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return StudentViewModel(repository) as T
    }
}
```

Isječak kôda 9: Implementacija StudentFactory klase.

Za kraj trebamo koristiti RecyclerView kako bi prikazali podatke na korisničko sučelje. Prvo definiramo activity_main.xml kojeg je već Android Studio automatski kreirao. U kôdu vidimo da ćemo podatke prikazivati preko RecyclerView. Također imat ćemo dva EditTexta koja će nam služit za unos imena i prezimena studenta te gumb za unos.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".view.MainActivity">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerViewStudents"
        android:layout_width="match_parent"
        android:layout_height="580dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/editTextTextStudentName"
        android:layout_width="180dp"
        android:layout_height="50dp"
        android:layout_marginBottom="10dp"
        android:ems="10"
        android:inputType="textPersonName"
        app:layout_constraintBottom_toTopOf="@+id/buttonInsert"
        app:layout_constraintEnd_toStartOf="@+id/editTextStudentSurname"
        app:layout_constraintStart_toStartOf="parent" />
```

```
    <EditText
        android:id="@+id/editTextStudentSurname"
        android:layout_width="180dp"
        android:layout_height="50dp"
        android:layout_marginEnd="10dp"
        android:layout_marginBottom="10dp"
        android:ems="10"
        android:inputType="textPersonName"
        app:layout_constraintBottom_toTopOf="@+id/buttonInsert"
        app:layout_constraintEnd_toEndOf="parent" />

    <Button
        android:id="@+id/buttonInsert"
        android:layout_width="223dp"
        android:layout_height="52dp"
        android:text="Insert"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Isječak kôda 10: Prikaz logike korisničkog sučelja

Nadalje, kreiramo novi layout koji će prikazivati kako će pojedini podatak izgledati unutar RecyclerViewa. Nazovimo ga item_student.xml te ga smjestimo unutar layout spremnika.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    app:cardBackgroundColor="@color/white"
    app:cardCornerRadius="5dp"
    app:cardElevation="4dp"
    app:cardUseCompatPadding="true">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/textViewStudentName"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginLeft="10dp"
            android:layout_marginTop="10dp"
            android:textStyle="bold"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
```

```
        <TextView
            android:id="@+id/textViewStudentSurname"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:textStyle="bold"
            app:layout_constraintLeft_toLeftOf="@id/textViewStudentName"
            app:layout_constraintTop_toBottomOf="@id/textViewStudentName"
/>

        <TextView
            android:id="@+id/textViewStudentYear"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:layout_marginBottom="10dp"
            android:textStyle="bold"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="@id/textViewStudentSurname"

app:layout_constraintTop_toBottomOf="@id/textViewStudentSurname" />


    </androidx.constraintlayout.widget.ConstraintLayout>


</androidx.cardview.widget.CardView>
```

Isječak kôda 11: Prikaz pojedinog podatka unutar RecyclerViewa

Za kraj trebamo napisati logički dio prikaza podataka. Unutar View spremnika kreiramo novu klasu s nazivom StudentAdapter.

```
class StudentAdapter() : ListAdapter<Student,
StudentAdapter.StudentViewHolder>(DiffCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
StudentViewHolder {
        val binding =
ItemStudentBinding.inflate(LayoutInflater.from(parent.context), parent,
false)
        return StudentViewHolder(binding)
    }

    override fun onBindViewHolder(holder: StudentViewHolder, position: Int)
{
        val currentItem = getItem(position)
        holder.bind(currentItem)

    }

    class StudentViewHolder(private val binding: ItemStudentBinding) :
        RecyclerView.ViewHolder(binding.root) {
```

```
        fun bind(student: Student) {
            binding.apply {
                textViewStudentName.text = student.ime
                textViewStudentSurname.text = student.prezime
                textViewStudentYear.text = student.godine.toString()
            }

        }
    }

    class DiffCallback : DiffUtil.ItemCallback<Student>() {

        override fun areItemsTheSame(oldItem: Student, newItem: Student) =
oldItem.id == newItem.id

        override fun areContentsTheSame(oldItem: Student, newItem: Student)
= oldItem == newItem

    }
}
```

Isječak kôda 12: Logički dio implementacije RecyclerViewa

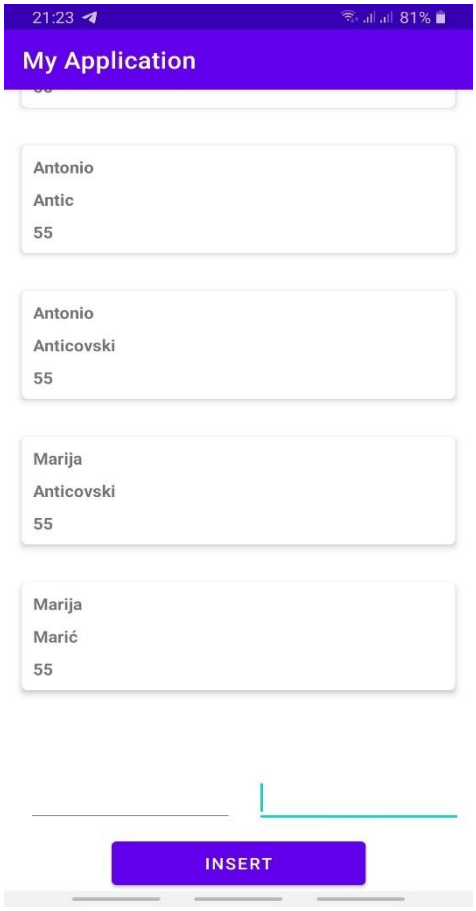
Za kraj ostalo nam je prikazati sve podatke. Za to koristimo MainActivity klasu koju je Android Studio automatski kreirao prilikom kreiranja novog projekta te smo je smjestili u View spremnik. MainActivity izgleda ovako:

```
class MainActivity : AppCompatActivity() {

    private lateinit var viewModel: StudentViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        val repository = StudentRepositoryOne(StudentDatabase(context =
this))
        val viewModelFactory = RateRestaurantViewModelFactory(repository)
        viewModel = ViewModelProvider(this,
viewModelFactory).get(StudentViewModel::class.java)

        val studentAdapter = StudentAdapter()

        binding.apply {
            recyclerViewStudents.apply {
                adapter = studentAdapter
                layoutManager = LinearLayoutManager(this@MainActivity)
            }
        }

        viewModel.getAllStudents()
```

```
        viewModel.studentList.observe(this,{ students ->
            studentAdapter.submitList(students)
            Log.d("Student", students.toString())
        })
        binding.buttonInsert.setOnClickListener {
            var insertStudent =
Student(binding.editTextTextStudentName.text.toString(),
binding.editTextStudentSurname.text.toString(), 55)
            viewModel.insertStudent(insertStudent)
            viewModel.getAllStudents()
        }
    }
}
```

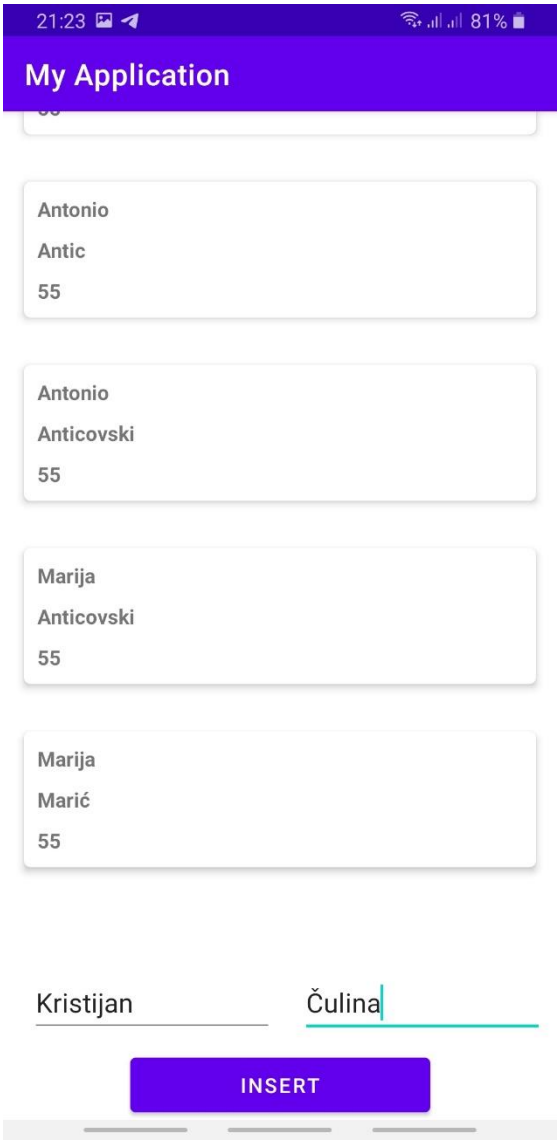
Isječak kôda 13: Implementacija MainActivitya.

Za kraj nam ostaje pokazati kako pokrenuta aplikacija izgleda na našem Android uređaju.

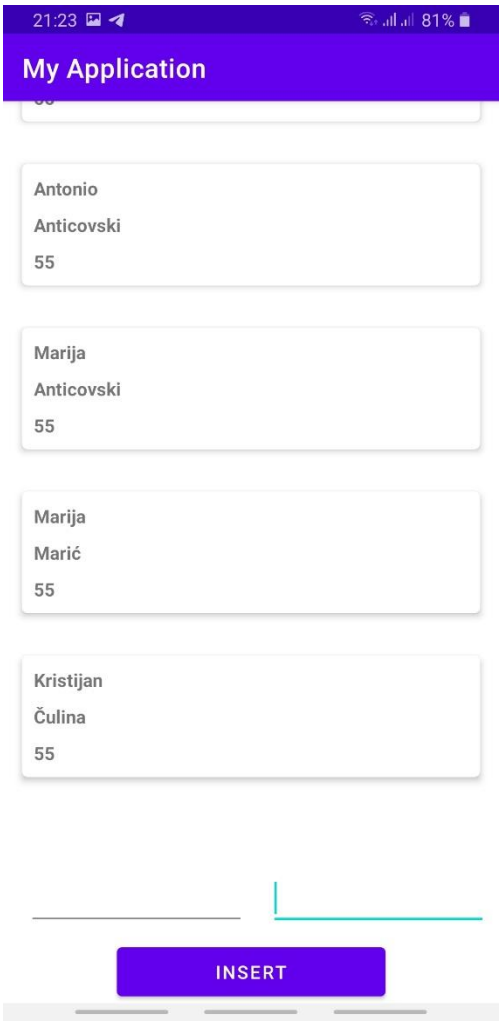


Slika 15: Izgled aplikacije

Možemo unijeti jedan podatak da se uvjerimo da sve funkcionira unutar naše aplikacije.



Slika 16: Unos novog studenta



Slika 17: Prikaz unesenog studenta na popis studenata

8. Zaključak

Danas jepri razvoju aplikacija, pogotovo Android aplikacija, jako bitno pratiti neki arhitekturalni pristup. Praćenje određenog arhitekturalnog pristupa olakšava testiranje, održavanje i nadogradnju aplikacije. Najpoznatiji pristup, koji je ujedno i predložen u sklopu Android Jetpacka, je Model-View-ViewModel.

Cilj ovog rada bilo je objasniti što predstavlja riječ arhitektura te zbog čega je bitno slijediti određenu arhitekturu. Objašnjeni su osnovni arhitekturalni pristupi pri implementraciji aplikacija za Android. Dotaknuo sam se i Android Jetpacka kojeg sam koristio prilikom implementacije aplikacije za koju sam koristio preporučeni arhitekturalni pristup u sklopu Android Jetpacka. Usporedio sam tri glavna arhitekturalna pristupa pri implementaciji aplikacija za Android od kojih se najbolji pokazao MVVM. On je ujedno i najnoviji arhitekturalni pristup, nastao kao rezultat nedostataka MVC i MVP arhitekturalnih pristupa. Za kraj sam implementirao aplikaciju za čiju sam implementaciju pratio arhitekturalni pristup predložen u sklopu Android Jetpacka.

Glavni alat za implementaciju aplikacije bio je Android Studio koji je službeno razvojno okruženje (IDE – integrated development environment) za Googleov operacijski sustav Android izgrađen na JetBrains' InteliJ IDEA softveru i dizajniran posebno za razvoj Androida. Sav programski kôdpisan je u Kotlin programskom jeziku unutar Android Studija.

Programskom kôdu aplikacije koja je izrađena u ovom završnom radu možete pristupiti na sljedećem linku: https://github.com/qbitdone/mvvm-example

9. Popis literature

Akhtar, Nayab & Ghafoor, Sana. (2021). Analysis of Architectural Patterns for Android Development. https://www.researchgate.net/publication/352021976

Android Developers. (2019). Android Jetpack. [online] Preuzeto 30.8.2021. s https://developer.android.com/jetpack

Barbacci, Mario & Klein, Mark & Longstaff, Thomas & Weinstock, Charles. (1995). Quality Attributes. https://www.researchgate.net/publication/242437986

Dabbagh, Mohammad & Lee, Sai. (2014). An Approach for Integrating the Prioritization of Functional and Nonfunctional Requirements. TheScientificWorldJournal. 2014. 737626. 10.1155/2014/737626. https://www.researchgate.net/publication/263585408

Daoudi, Aymen & El-Boussaidi, Ghizlane & Moha, Naouel & Kpodjedo, Sègla. (2019). An exploratory study of MVC-based architectural patterns in Android apps. 1711-1720. 10.1145/3297280.3297447. https://www.researchgate.net/publication/332813894

Developer Android. (2021-a). Android Studio. Preuzeto 23.7.2021. s https://developer.android.com/studio

Developer Android. (2021-b). Documentation for app developers. Preuzeto 23.7.2021. s https://developer.android.com/docs

Developer Android. (2021-c). Guide to app arhitecture. Preuzeto 30.7.2021. s https://developer.android.com/jetpack/guide

Developer Android. (2021-d.). Recommended app architecture. Preuzeto 30.7.2021. s https://developer.android.com/jetpack/guide#recommended-app-arch

Guarav, A. (2019). What is Android Jetpack and why should we use it? Preuzeto 1.9.2021. s https://blog.mindorks.com/what-is-android-jetpack-and-why-should-we-use-it

Harrouk C. (2021). What is Arhitecture? According to our Readers. Preuzeto 24.7.2021. s https://www.archdaily.com/960128/what-is-architecture-according-to-our-readers

Kanat, M. (2008). What is Overengineering? Preuzeto 24.7.2021. s https://www.codesimplicity.com/post/what-is-overengineering/

Kotlin. (2021-a). A modern programming language that makes developers happier. Preuzeto 23.7.2021. s https://kotlinlang.org/

Kotlin. (2021-b). Kotlin for Android. Preuzeto 23.7.2021. s https://kotlinlang.org/docs/android-overview.html

Kotlin. (2021-c). Kotlin Language Documentation 1.5.21. Preuzeto 23.7.2021. s https://kotlinlang.org/docs/kotlin-reference.pdf#page=42&zoom=100,66,96

Maxwell, E. (2017). MVC vs. MVP. Vs MVVM on Android. Preuzeto 30.8.2021. s https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

Mburu, D. (2018). What are pros and cons of the MVC pattern?. Preuzeto 4.8.2021. s https://www.quora.com/What-are-the-pros-and-cons-of-the-MVC-pattern?share=1

Microsoft. (2021). Microsoft Word. Preuzeto 23.7.2021. s https://www.microsoft.com/en-us/microsoft-365/word

Mino, S. (2020). What is Software Arhitecture and Why Is It Important? [Blog post]. Preuzeto 24.7.2021. s https://www.jobsity.com/blog/software-architecture

Mishra, R. (2020). Difference Between MVC, MVP and MVVM Arhitecture Pattern in Android. Preuzeto 5.8.2021. s https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/

Mishra, R. (2020). MVC (Model View Controller) Arhitecture Pattern in Android with Example. Preuzeto 29.7.2021. s https://www.geeksforgeeks.org/mvc-model-view-controller-architecture-pattern-in-android-with-example/

Mishra, R. (2020). MVP (Model View Presenter) Arhitecture Pattern in Android with Example. Preuzeto 4.8.2021. s https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example/

Mishra, R. (2021). MVVM (Model-View-ViewModel) Architecture Pattern in Android. Preuzeto 5.8.2021. s https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/

Perera, R. (2019). How to create an Android project with the MVVM arhitectural pattern [Part 1]. Preuzeto 29.7.2021. s https://medium.com/@rajithaperera_81165/how-to-create-an-android-project-with-the-mvvm-architectural-pattern-part-1-ac7029653056.

Richards, M. (2015). Software Arhitecutre Patterns. O'Reilly Media, Inc.

Sokolova, Karina & Lemercier, Marc. (2013). Android Passive MVC: Novel Architecture Model for the Android Application Development. https://www.researchgate.net/publication/286936435

Srivastava, V. (2019). MVC vs MVP vs MVVM architecture in Android. Preuzeto 5.8.2021. s https://blog.mindorks.com/mvc-mvp-mvvm-architecture-in-android

Svirca, Z. (2020). Model View Presenter (MVP). Preuzeto 5.8.2021. s https://medium.datadriveninvestor.com/model-view-presenter-mvp-5c3439227f83

The DotNet Guide. (bez dat.). MVC Design Pattern. Preuzeto 29.7.2021. s https://thedotnetguide.com/mvc-design-pattern/

TouchGFX. (bez dat.). Model-View-Presenter Desgin Pattern. Preuzeto 5.8.2021. s https://support.touchgfx.com/docs/development/ui-development/software-architecture/model-view-presenter-design-pattern

Wikipedia contributors. (2021). Android Studio. In Wikipedia, The Free Encyclopedia. Preuzeto 23.7.2021. s https://en.wikipedia.org/w/index.php?title=Android_Studio&oldid=1037394287

Wikipedia contributors. (2021). Model-View-Controller. In Wikipedia, The Free Encyclopedia. Preuzeto 29.7.2021. s https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Wikipedia contributors. (2021). Model-View-Presenter. In Wikipedia, The Free Encyclopedia. Preuzeto 4.8.2021. s https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter

Wikipedia contributors. (2021). Model-View-ViewModel. In Wikipedia, The Free Encyclopedia. Preuzeto 5.8.2021. s https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel

10. Popis slika

11. Popis tablica

12. Popis kodova