Usporedba pristupa za izgradnju korisničkog sučelja na Android platformi

Mesinger, Zvonimir

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

Stručni studij

USPOREDBA PRISTUPA ZA IZGRADNJU KORISNIČKOG SUČELJA NA ANDROID PLATFORMI

Završni rad

Zvonimir Mesinger

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1S: Obrazac za ocjenu završnog rada na stručnom prijediplomskom studiju

Ocjena završnog rada na stručnom prijediplomskom studiju

Ime i prezime pristupnika:	Zvonimir Mesinger
Studij, smjer:	Stručni prijediplomski studij Računarstvo
Mat. br. pristupnika, god.	AR 4676, 26.07.2018.
JMBAG:	0165079716
Mentor:	doc. dr. sc. Bruno Zorić
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	doc. dr. sc. Dražen Bajer
Član Povjerenstva 1:	doc. dr. sc. Bruno Zorić
Član Povjerenstva 2:	dr. sc. Mario Dudjak
Naslov završnog rada:	%naziv_rada%
Znanstvena grana završnog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada:	U teorijskom dijelu rada potrebno je opisati različite pristupe izgradnji korisničkih sučelja na mobilnim platformama pri čemu je posebnu pažnju potrebno posvetiti Android platformi. Prikazati prednosti i nedostatke uporabe klasičnog pristupa zasnovanom na označnom jeziku XML i modernom pristupu koji rabi okvir Jetpack Compose. U praktičnom dijelu programski ostvariti aplikaciju za Android platformu korištenjem obaju pristupa te ih usporediti s gledišta razvoja, korištenja i održavanja. Tema rezervirana za: Zvonimir Mesinger

Datum ocjene pismenog dijela završnog rada od strane mentora:	23.06.2024.
Ocjena pismenog dijela završnog rada od strane mentora:	Izvrstan (5)
Datum obrane završnog rada:	11.07.2024.
Ocjena usmenog dijela završnog rada (obrane):	Izvrstan (5)
Ukupna ocjena završnog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije završnog rada čime je pristupnik završio stručni prijediplomski studij:	12.07.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 12.07.2024.

Ime i prezime Pristupnika:	Zvonimir Mesinger
Studij:	Stručni prijediplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	AR 4676, 26.07.2018.
Turnitin podudaranje [%]:	8

Ovom izjavom izjavljujem da je rad pod nazivom**: Usporedba pristupa za izgradnju korisničkog sučelja na Android platformi**

izrađen pod vodstvom mentora doc. dr. sc. Bruno Zorić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

Pri izgradnji aplikacija za Android platformu pojavljuje se problem razdvajanja izgradnje korisničkog sučelja od poslovne logike aplikacije. Problem nastaje kada se programska logika usko veže uz izgradnju korisničkog sučelja. Aplikaciju je teže testirati i mijenjati jer promjene u jednom dijelu koda zahtijevaju promjene u drugim dijelovima, što projekt čini neodrživim. Rješenje ovog problema predložio je Edsger W. Dijkstra 1974. godine. Njegov prijedlog temelji se na principu odvajanja ovisnosti (engl. *separation of concerns*). Ideja je da aplikaciju treba razložiti na odvojene dijelove koji su odgovorni za jedan dio funkcioniranja aplikacije [1]. Taj pristup slijede aplikacijske arhitekture kao što su MVVM, MVC i MVP. Princip odvajanja ovisnosti na web platformi ostvaruje se kombinacijom aplikacijske arhitekture i komponenti u kojima su enkapsulirani logički i prezentacijski sloj. Odvajanje ovisnosti unutar komponenti postiže se labavim vezama između označnog jezika, jezika zaduženog za logiku i stilizacijskog jezika. Mobilne platforme princip odvajanja ovisnosti pokušavaju ostvariti kroz arhitekturalne obrasce, okvirima za ubrizgavanje ovisnosti te korištenjem različitih oblikovnih obrazaca i biblioteka. Korisničko sučelje na Android platformi gradi se na dva načina. Prvi način označnim jezikom XML, koji u kombinaciji s programskim jezicima Java i Kotlin pruža interaktivno korisničko sučelje. Drugi način je okvir Jetpack Compose, novi, moderni i deklarativni način koji minimizira ovisnosti i maksimizira koheziju. Cilj završnog rada je istražiti pristupe za izgradnju korisničkog sučelja, s posebnim naglaskom na Android platformu, te kroz teorijski prikaz i praktičnu izradu aplikacija, usporediti klasični pristup zasnovan na označnom jeziku XML i pristup zasnovan na modernom okviru Jetpack Compose.

U drugom poglavlju opisan je postupak izgradnje korisničkog sučelja na različitim platformama s fokusom na Android platformu. Treće poglavlje donosi usporedbu izgradnje korisničkog sučelja na Android platformi korištenjem označnog jezika XML i okvira Jetpack Compose. Četvrto poglavlje donosi analizu usporedbe na temelju čijih rezultata se donosi zaključak.

1.1 Zadatak završnog rada

U teorijskom dijelu rada potrebno je opisati različite pristupe izgradnji korisničkih sučelja na mobilnim platformama pri čemu je posebnu pažnju potrebno posvetiti Android platformi. Prikazati prednosti i nedostatke uporabe klasičnog pristupa zasnovanom na označnom jeziku XML i modernom pristupu koji rabi okvir Jetpack Compose. U praktičnom dijelu programski ostvariti

aplikaciju za Android platformu korištenjem obaju pristupa te ih usporediti s gledišta razvoja, korištenja i održavanja.

2. IZGRADNJA KORISNIČKOG SUČELJA

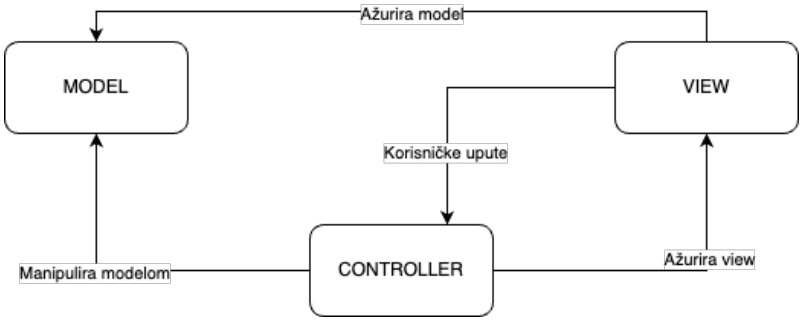
Korisničko sučelje je dio aplikacije gdje se odvija interakcija između krajnjeg korisnika, čovjeka, i aplikacije. Sa strane dizajna ono treba biti praktično, intuitivno i lako za razumijevanje. Zadaća korisničkog sučelja je prikazati podatke na ekranu u obliku koji je korisniku prihvatljiv [2]. Ono se sastoji od grafičkih elemenata koji omogućuju prikaz podataka i interakciju s aplikacijom, poput gumba ili tekstualnih polja, i programskog koda koji se bavi transformacijom podataka i pripremom za prikaz na ekranu [3]. Tijekom procesa izgradnje korisničkog sučelja, grafički se elementi slažu i tvore raspored elemenata (engl. *layout*). Sami grafički elementi nemaju nikakva ponašanja. Važnost ponašanja je ta što korisniku omogućuju interakciju s aplikacijom, na primjer, na pritisak gumba korisnik prelazi s jednog ekrana na drugi ekran. Kombinacijom grafičkih elemenata i ponašanja dobiva se korisničko sučelje. Pri procesu izgradnje korisničkog sučelja, koriste se razni alati i tehnologije kako bi se olakšao posao programera [4]. Osim što olakšavaju izgradnju samog korisničkog sučelja, omogućava se olakšano dodavanje novih značajki, veću razumljivost koda i mogućnost testiranja korisničkog sučelja odvojeno od ostatka aplikacije.

2.1. Razdvajanje izgradnje korisničkog sučelja od programske logike

U procesu izgradnje korisničkog sučelja važno je nastojati ne miješati korisničko sučelje i poslovnu logiku. Jedan od načina za postizanje labave veze između ovih dvaju dijelova aplikacije su moderne aplikacijske arhitekture. Kao što je prethodno navedeno, aplikacijske arhitekture slijede princip odvajanja ovisnosti. To znači da aplikacijske arhitekture pokušavaju velike i kompleksne aplikacije odvojiti na manje smislene dijelove čime se olakšava rad u timu, povećava mogućnost testiranja i nadogradnji postojećeg koda. Kroz primjenu aplikacijskih arhitektura, kod pristupa zasnovanog na označnom jeziku XML te pristupa zasnovanog na okviru Jetpack Compose, uvodi se slojevitost dijelova aplikacije čime se postiže bolja skalabilnost.

Moderne arhitekture dijele dva ista dijela aplikacije, dok treći ima svoje varijacije. Zbog toga popularni naziv za aplikacijske arhitekture je model-pogled-što god (engl. *Model-View-Whatever*, MVW). Neke od najpoznatijih arhitektura su model-pogled-kontroler (engl. *Model-View-Controller,* MVC), model-pogled-prezenter (engl. *Model-View-Presenter*, MVP), model-pogled-model pogleda (engl. *Model-View-ViewModel,* MVVM) i model-pogled-namjera (engl. *Model-View-Intent*, MVI) koje su detaljnije objašnjene u nastavku.

Model-View-Controller (MVC) je aplikacijska arhitektura koja je nastala u kasnim 1990-im godinama, a prvu implementaciju napisao je Trygve Reenskaug [5]. Ova arhitektura sastoji se od tri dijela čija se povezanost prikazuje na slici 2.1.



Slika 2.1. Prikaz komunikacije između dijelova aplikacije u MVC arhitekturi, izrađeno po uzoru na [26]

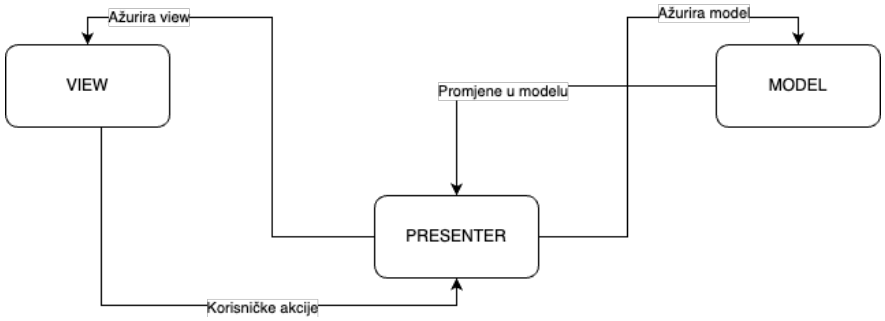
Model dio aplikacije definira podatke koje aplikacija treba sadržavati. U objektno-orijentiranoj terminologiji, *model* se sastoji od niza klasa koje modeliraju problem. Nadalje, *model* ne bi trebao imati nikakvo znanje o postojanju ostatka aplikacije [6]. Na primjer, aplikaciju koristi korisnik, stoga se kreira *model* korisnika koji ima neke atribute poput imena i lozinke. *Model* obavještava *view* da se dogodila promjena, ukoliko korisnik doda novu stavku na listu, *model* obavještava *view* o promjeni.

Drugi dio MVC arhitekture je *View*. Najjednostavniji opis komponente *View* je to da ona definira način na koji će se podaci prikazati korisniku [7]. S obzirom da *View* prezentira podatke, on je svjestan postojanja komponente *model* [6].

Treći dio MVC arhitekture je kontroler *controller*. On se bavi podacima na ulazu, priprema ih za prikaz, koji se na kraju izvršava u komponenti *view*. Pojednostavljeno rečeno, *controller* je odgovaran za podatke na ulazu, dok je *view* izlaz. Slično kako *view* zna za postojanje komponente *model*, tako i komponenta kontroler *controller* zna na postojanje komponente pogleda *view*, a pogled *view* ne zna za postojanje kontrolera *controller* [6].

Primjer komunikacije između dijelova MVC arhitekture koji na pojednostavljenoj razini prikazuje funkcioniranje MVC arhitekture je sljedeći: korisnik aktivacijom gumba dodaje novu stavku na listi, podaci te stavke šalju se iz *view* dijela u *controller* koji manipulira te podatke i prosljeđuje ih u *model*. Nakon toga *model* ima dužnost obavijestiti *view* da se dogodila promjena koju je potrebno prikazati.

Druga arhitektura koja se često koristi i prikazan je na slici 2.2. je MVP (*Model-View-Presenter*). Obrazac koji je 1990. godine korišten prvenstveno od strane američke tvrtke Taligent. MVP je veoma sličan MVC uzorku. Najveća razlika između ova dva uzorka je to što kontroler *controller* zamjenjuje prezenter *presenter*. Iako ga zamjenjuje, oni ne rade iste stvari. MVP je još poznat i pod nazivom složeni uzorak (engl. *compound pattern*) [8].



Slika 2.2. Prikaz komunikacije između dijelova aplikacije u MVP arhitekturi, izrađeno po uzoru na [27]

Model u MVP arhitekturi sadrži podatke aplikacije i pravila koja se koriste za upravljanje podacima. *Model* dio aplikacije neovisan je o komponentama *presenter* i *view*. Iako *presenter* zna za postojanje komponente *model*, *model* ne zna da *presenter* postoji. Isto tako *model* nije svjestan postojanja komponente *view* [9]. *Model* može biti sučelje odgovorno za dohvaćanje podataka iz lokalne ili udaljene baze podataka. Glavna zadaća mu je upravljanje podacima [8].

Kao i kod obrasca MVC, *view* je dio koji predstavlja korisničko sučelje aplikacije. Njegova jedina svrha trebala bi biti prikazivanje podataka i ne bi trebao imati nikakve programske logike [8]. On je dio koji preuzima odgovornost za interakciju s korisnikom. *View* zna za postojanje komponente *presenter*, no ne zna za postojanje komponente *model*.

Dio po kojem se MVP obrazac razlikuje od obrasca MVC, *presenter,* poznat je još kao posrednik. On implementira komunikaciju između komponenti *view* i *model*. S obzirom da oni ne komuniciraju direktno, *presenter* se ponaša poput mosta koji dohvaća podatke iz komponenta *model* te ih nakon toga oblikuje u oblik koji je prihvatljiv komponenti *view* [8]. *Presenter* drži podatke koji će se prikazati i odlučuje koji će podaci biti prikazani u komponenti *view*. Sve korisničke radnje nalaze u ovom dijelu aplikacije [9].

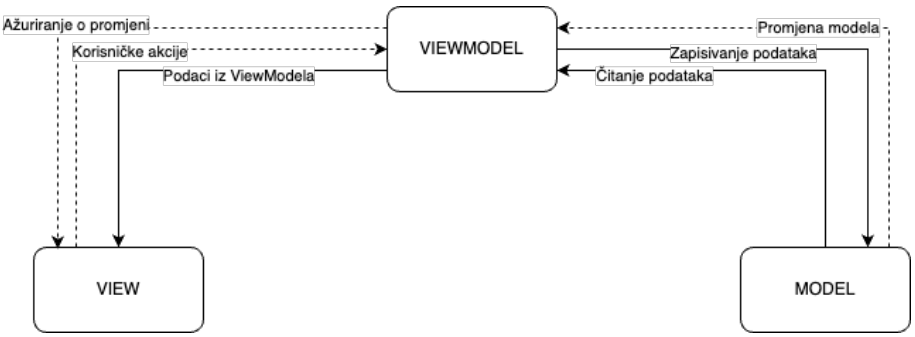
Primjer komunikacije pojednostavljuje se na sljedeći način: korisnik izvršava neku radnju na korisničkom sučelju gdje se iz komponente *presenter* poziva metoda koja predstavlja korisničku radnju. *Presenter* nakon toga javlja sloju *model* da se dogodila promjena i da je potrebno ažurirati

model. *Model* vraća povratnu informaciju da je promjena uspješno obavljena te nakon toga *presenter* zahtijeva od komponente *view* da se izvrši osvježavanje korisničkog sučelja.

Arhitekturalni obrazac koji se najčešće koristi pri razvoju aplikacija za Android platformu je MVVM, odnosno *Model-View-ViewModel*. Kao i kod prethodno navedenih obrazaca ovakvog tipa, *model* predstavlja apstrakciju izvora podataka. *View* je dio aplikacije koji kao zadaću ima samo prikaz podataka te obavještavanje komponente *ViewModel* o korisničkim radnjama. Novitet u usporedbi s ostalima MVx arhitekturalnim obrascima je *ViewModel*. Komponenta *ViewModel* funkcionira kao spojnica između komponenti *model* i view. *ViewModel* dopušta pristup komponenti *view* prema izvoru podatka koji se nalazi u *modelu*, te reagira na korisničke radnje i obavještava *model* o radnji koja treba biti provedena [28].

Komponenta *ViewModel* rješava probleme koje je imao njegov prethodnik u svijetu Android aplikacija, komponenta *Presenter* iz MVP obrasca. Komponenta *ViewModel* kao sastavnica obrasca nema nikakvu referencu na *view*, te jedina komunikacija s komponentom *view* odrađuje se na inicijativu korisnika ili kao posljedica komunikacije s komponentom *model* [28]. Ovakvim načinom slaganja komponenti aplikacije dolazi se do situacije u kojoj se smanjuje ovisnost između komponenti, što omogućuje povećanu skalabilnost i održivost aplikacije.

Slika 2.3. prikazuje komunikaciju između komponenti MVVM arhitekture. Jednostavan primjer komunikacije u aplikaciji je spremanje podataka u bazu podataka. Recimo da korisnik želi pohraniti podatke u bazu. Korisnik na korisničkom sučelju koje je dio komponente *view* popunjava upit s podacima, na pritisak gumba komponenta *view* obavještava komponentu *ViewModel* da je potrebno izvršiti radnju s predanim podacima. Nakon toga, komponenta *ViewModel* prosljeđuje podatke komponenti *model* u kojoj se izvršava zapisivanje podataka u bazu. Po rezultatu pohrane, koji može biti uspješan ili neuspješan, komponenta *model* obavještava komponentu ViewModel i prosljeđuje joj rezultat pohrane koji zatim komponenta *ViewModel* predaje pogledu *view* gdje se rezultat pohrane prikazuje korisniku. Kako bi se smanjila ovisnost između komponente *model* i komponente *view*, primjenjuje se obrazac ponašanja naziva posrednik. Primjenom posrednika postiže se princip odvajanja ovisnosti jer se uklanja izravna komunikacija između komponente *view* i komponente *model*, te se ona preusmjerava kroz komponentu *viewmodel* koja se ponaša kao posrednik između ostalih komponenti MVVM obrasca.



Slika 2.3. Prikaz komunikacije između dijelova aplikacije u MVVM arhitekturi, izrađeno po uzoru na [25]

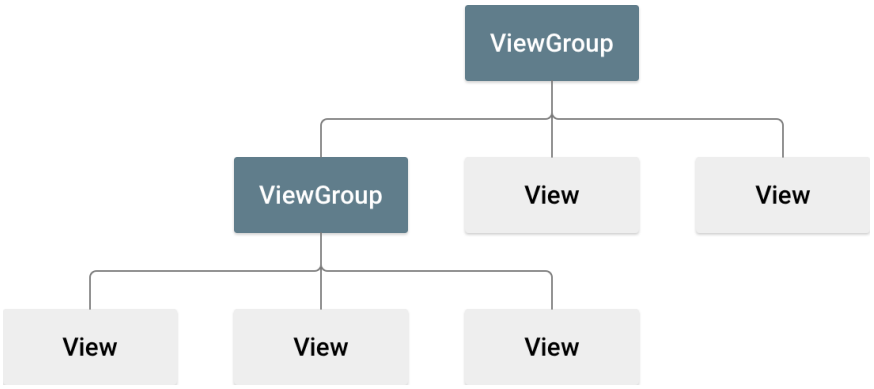
2.2. Izgradnja korisničkog sučelja na Android platformi

U aplikacijama koje su razvijene za Android mobilnu platformu, korisničko sučelje može se graditi na dva načina. Jedan od ta dva načina je pristup zasnovan na označnom jeziku XML koji se temelji na rasporedima elemenata *layout* i instancama klase *View* koji predstavljaju elemente korisničkog sučelja. Drugi način izgradnje korisničkog sučelja na Android platformi zasnovan je na modernom okviru Jetpack Compose. Polazna točka gradnje korisničkog sučelja korištenjem okvira Jetpack Compose su gradbeni blokovi naziva *Composables*. *Composables* su funkcije pisane u programskom jeziku Kotlin koje opisuju dio korisničkog sučelja, nemaju povratni tip, imaju mogućnost primanja podataka i generiranja istih na ekranu te mogu emitirati nekoliko elemenata korisničkog sučelja [15]. Kroz primjenu aplikacijskih arhitekturalnih obrazaca pri izgradnji aplikacija, odvaja se kod zadužen za izgradnju korisničkog sučelja i kod odgovoran za programsku logiku aplikacije.

2.2.1. Pristup zasnovan na označnom jeziku XML

Pristup zasnovan na označnom jeziku XML trenutno je najkorišteniji pristup za izradu korisničkog sučelja na Android platformi. Sastoji se od opisivanja rasporeda elemenata *layout* i elementa objekata *View* koji nastanjuju taj raspored elemenata *layout*. Rasporedi elemenata *layout* su *ViewGroup* objekti koji mogu sadržavati djecu. Oni se ponašaju kao kontejneri koji kontroliraju pozicioniranje djece na ekranu. Djeca koja se nalaze unutar *ViewGroup* kontejnera, zapravo su instance klase *View*. *View* objekti su komponente korisničkog sučelja, korisnicima poznatije kao gumbi, tekstualni okviri [14].

Hijerarhija i odnos pogleda *View* i objekta *ViewGroup* prikazana je slikom 2.4.



Slika 2.4. Prikaz hijerarhije korisničkog sučelja na Android platformi [18]

Na temelju slike 2.4. može se primijetiti kako raspored elemenata *layout* koji su instance *ViewGroup* klase sadrže djecu koji su instance klase *View*. Osim što rasporedi elemenata *layout* mogu sadržavati instance klase *View*, mogu sadržavati i druge rasporede elemenata *layout*. Oblikovni obrazac koji se primjećuje je kompozit čija struktura ima oblik stabla, kao što se prikazuje na slici 2.4. Obrazac kompozit sastoji se od listova, kompozita i komponente. Na primjeru sa slike 2.4., objekt *ViewGroup* predstavlja kompozit, instance klase *view* listove, a klasa *View* komponentu Listovi ne mogu imati djecu, dok kompoziti imaju tu mogućnost.

Rasporedi elemenata su objekti koji pružaju različite strukture rasporeda komponenata, kao na primjer raspored elemenata *LinearLayout ili RelativeLayout*. Deklariranje elemenata korisničkog sučelja korištenjem označnog jezika XML omogućuje odvajanje prezentacije aplikacije i programskog koda koji kontrolira ponašanja [18].

Primjer na slici 2.5. prikazuje jednostavno korisničko sučelje koje se sastoji od rasporeda elemenata *LinearLayout*, koje je korijenski element i sadrži *TextView* i *Button* elemente koji su djeca tog korijenskog elementa. Korijenski element mora biti *View* ili *ViewGroup* objekt. U taj korijenski element može se dodavati i druge rasporede elemenata kako bi kreirali korisničko sučelje koje odgovara našim potrebama [18].

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Slika 2.5. Primjer jednostavnog korisničkog sučelja korištenjem označnog jezika XML

2.2.2. Pristup zasnovan na okviru Jetpack Compose

Veliki problem s pristupom zasnovanom na označnom jeziku XML je božanska klasa *View*. Svaka komponenta korisničkog sučelja direktna je ili indirektna podklasa *View* klase. S razvitkom Android platforme, klasa *View* dobivala je sve više funkcionalnosti. Trenutno klasa *View* sadrži preko 29000 linija koda i prešla je točku u kojoj se može refaktorirati. To znači da je cijeli alat za izgradnju korisničkog sučelja teško održavati te je njegova skalabilnost znatno narušena [19].

Okvir Jetpack Compose razvijen je kako bi se riješili problemi vezani uz klasu *View*, smanjila količina koda potrebnog za izgradnju korisničkog sučelja i ubrzao proces razvijanja aplikacije. Compose slijedi modernu deklarativnu paradigmu koja je se već neko vrijeme koristi u raznim okvirima za izgradnju aplikacija na web platformi [20].

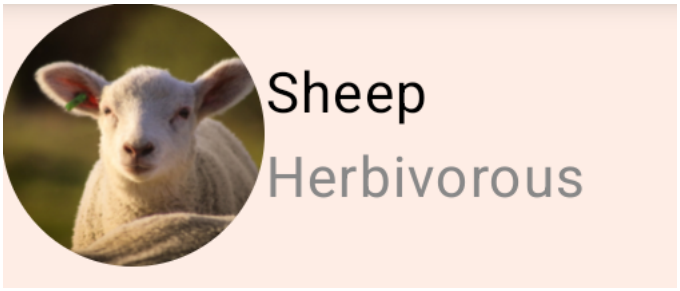
Temeljni gradbeni blok korisničkog sučelja u okviru Jetpack Compose su *Composable* funkcije. To su metode pisane u Kotlin programskom jeziku koje nemaju povratni tip, unutar jedne *Composable* funkcije može se imati nekoliko elemenata korisničkog sučelja. *Composable* metode potrebno je anotirati s *@Composable* anotacijom. Anotacije su dio biblioteke AndroidAnnotations i omogućuju smanjivanje nepotrebnog koda automatskom generacijom istog . Raspored elemenata unutar *Composable* metode definiran je pomoću stupaca i redaka. Koristeći *Composable* funkciju *Column*, elementi korisničkog sučelja unutar funkcije raspoređeni su vertikalno unutar stupca, dok funkcija *Row* omogućuje raspored horizontalno unutar retka. Također, postoji i *Box* funkcija koja elemente korisničkog sučelja raspoređuje jedne preko drugih. Moguće je i kombiniranje *Column, Row* i *Box* funkcija. Isječak koda prikazan na slici 2.6. prikazuje kako se koristi kombinacija *Column* i *Row* metoda. Unutar metode *AnimalCard* poziva se metoda *Row* s vertikalnim poravnanjem u središtu roditeljskog elemenata, nakon toga vidljiv je slikovni element korisničkog

sučelja koji je prikazan prvi. Poslije slike poziva se *Column* metoda unutar koje se nalaze tekstualni elementi koji su raspoređeni vertikalno.

```
@Composable
fun AnimalCard(animal: Animal) {
    Row (
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.Center
    ) {
        AsyncImage(...)
        Column {
            Text(text = animal.name)
            Text(text = animal.diet,
                color = Color.Gray)
        }
    }
}
```

Slika 2.6. Primjer jednostavnog elementa korisničkog sučelja korištenjem okvira Jetpack Compose

Funkcija *AnimalCard* opisana na slici 2.6. za rezultat ima element korisničkog sučelja prikazanog na slici 2.7. Sučelje se sastoji od slike i dva tekstualna polja.



Slika 2.7. Rezultat jednostavnog elementa korisničkog sučelja

2.2.3. Prednosti i mane različitih pristupa

Postoji nekoliko načina za razvoj korisničkog sučelja na Android platformi, neki od njih odnose se na nativne aplikacije dok se drugi koriste za razvijanje aplikacija koje se mogu pokretati na različitim platformama. Nativne aplikacije koriste označni jezik XML, Jetpack Compose, React Native i Xamarin okvire. Prednost nativnih aplikacija je to što omogućuju izradu aplikacija koje imaju visoke performanse, no znaju bit vremenski i financijski zahtjevne. Jeftinija alternativa nativnim aplikacijama je razvijanje aplikacija za različite platforme, u ovom slučaju Android i iOS. Takve aplikacije zahtijevaju manje vremena i financijskih sredstava za razvijanje jer je dovoljan samo jedan programer, ali kao rezultat imaju otežano pružanje jednakog korisničkog dojma kao nativne aplikacije [21].

Dva pristupa na kojima je fokus u ovom radu, pristup zasnovan na označnom jeziku XML i pristup zasnovan na okviru Jetpack Compose, također imaju svoje prednosti i mane.

U tablici 2.1. navedene su prednosti i mane pristupa zasnovanog na označnom jeziku XML, od kojih je potrebno istaknuti činjenicu da klasa *View* sadrži preko 29000 linija koda, zbog čega je njezino održavanje značajno otežano, a proširivanje gotovo nemoguće.

Tablica 2.1. Prednosti i mane pristupa zasnovanog na označnom jeziku XML

PREDNOSTI	**MANE**
Omogućuje odvajanje ovisnost između izgleda korisničkog sučelja i ponašanja	Klasa *View* koja sadrži preko 29000 linija koda, je neodrživa
Brzi dizajn i raspored elemenata korisničkog sučelja	Alat zasnovan na označnom jeziku XML je neodrživ zbog veličine i kompleksnosti klase *View*
Stabilnost i podrška zajednice	Potrebno je napuhati XML datoteku u Kotlin ili Java objekt, što smanjuje koheziju
	Ovisnosti između dijelova korisničkog sučelja su implicitne
	Zbog korištenja dva jezika (XML i Java/Kotlin) otežana je komunikacija i kohezija
	Potrebno je ručno upravljati promjenama stanja

Glavna prednost pristupa zasnovanog na modernom okviru Jetpack Compose je činjenica da je pisan u programskom jeziku Kotlin te da je kod kojim se opisuje korisničko sučelje i implementiraju ponašanja pisan u programskom jeziku Kotlin. Okvir Jetpack Compose još uvijek je relativno nov te iz tog razloga neke stvari još uvijek nisu implementirane i ne može pružiti jednaku stabilnost kao pristup zasnovan na označnom jeziku XML koji se već godinama koristi. U tablici 2.2. prikazane su preostale prednosti i mane pristupa zasnovanog na okviru Jetpack Compose.

Tablica 2.2. Prednosti i mane pristupa zasnovanog na okviru Jetpack Compose

PREDNOSTI	**MANE**
Pisano u programskom jeziku Kotlin što omogućuje olakšano refaktoriranje ovisnosti između korisničkog sučelja i poslovne logike aplikacije	Relativno nov pristup koji još uvijek evoluira
Povećava koheziju koda zbog činjenice da je sve pisano u programskom jeziku Kotlin	Neke značajke još uvijek nisu implementirane
Deklarativna paradigma smanjuje količinu koda	
Poboljšane performanse aplikacije	
Brine se za tranziciju stanja iz jednog u drugo	

2.2.4. Korištenje standardnih biblioteka i obrazaca prilikom uporabe oba pristupa

Prilikom uporabe oba pristupa korisno je koristiti različite standardne biblioteke koje olakšavaju razvijanje aplikacije i smanjuju količinu koda. Skup standardnih biblioteka naziva Android Jetpack stvoren je kako bi programerima pomogao pri praćenju najboljih praksi za razvijanje aplikacija, smanjenje nepotrebnog koda i mogućnost korištenja aplikacije neovisno o verziji Androida i uređaju [22].

Jedna od standardnih biblioteka iz tog seta je Navigation Component biblioteka. Navigation Component biblioteka omogućava olakšano kretanje između različitih dijelova aplikacije. Na primjer, nakon prijave u aplikaciju, potrebno je prikazati zaslon s objavama svih korisnika aplikacije. Tu do izražaja dolazi biblioteka Navigation Component. Kreiranjem grafa *navigation graph* olakšava se kretnja između različitih destinacija, uz to omogućuje se prosljeđivanje različitih tipova podataka između destinacija korištenjem biblioteke Safe Arguments.

S obzirom da je graf *navigation graph* XML datoteka, prilikom korištenja okvira Jetpack Compose on postaje nepotreban. Stoga se navigacija u okviru Jetpack Compose korištenjem biblioteke Navigation Component piše u Kotlin programskom jeziku. Kreiranje kontejnera *NavHost* zahtjeva kontroler *NavController* te mu se zadaje početna destinacija i ostale destinacije. Kontejner *NavHost* je *Composable* funkcija u kojoj se prikazuju ostale *Composable* funkcije, odnosno destinacije. Unutar *NavHost* funkcije definira se navigacijski graf. Navigacija između destinacija definira se u *Composable* funkcijama pomoću metode *navigate* [23].

Kao nadogradnja na biblioteku Jetpack Compose Navigation, moguće je koristiti biblioteku Compose Destinations koja smanjuje nepotreban kod, pojednostavljuje kompleksnost Jetpack Compose Navigation i osigurava sigurnost tipova kod navigacijskih argumenata [24].

2.2.5. MVVM u pristupu zasnovanom na označnom jeziku XML

Arhitekturalni obrazac MVVM ključni je dio u ostvarivanju principa odvajanja ovisnosti u pristupu zasnovanom na označnom jeziku XML. Na primjeru dohvaćanja fotografija s udaljenog poslužitelja agencije NASA prikazana je primjena obrasca. Na slici 2.8 prikazano je dohvaćanje instance komponente ViewModel za čije se kreiranje brine okvir za ubrizgavanje ovisnosti Koin unutar *LibraryImagesFragment* fragmenta. Instanca komponente ViewModel dohvaća se u View komponenti MVVM obrasca. Time se osigurava da View nije svjestan postojanja komponente Model te se komunikacija vrši kroz ViewModel.

```
private val viewModel: APIViewModel by viewModels()
```

Slika 2.8. Dohvaćanje instance komponente ViewModel

Unutar klase *APIViewModel* definirane su funkcije koje kreiraju HTTP poziv putem biblioteke Retrofit te vraćaju odgovor poslužitelja. Funkcije koje kreiraju HTTP poziv putem biblioteke Retrofit nalaze se unutar komponente *model* MVVM obrasca. Na slici 2.9. prikazana je implementacije klase *APIViewModel*. U primjeru na slici 2.9. koristi se funkcija *getLibraryImages*.

```
class APIViewModel(): ViewModel() {
    suspend fun getAPOD(): Response<PictureOfTheDay> {
        return LibraryRetrofit.apod.getData()
    }
    suspend fun getLibraryImages(keyword: String): Response<Base> {
        return LibraryRetrofit.library.getLibraryImages(keyword)
    }
    suspend fun getMarsPhotos(earthDate: String) : Response<MarsPhoto> {
        return LibraryRetrofit.apod.getRoverPhotos(earthDate, API_KEY)
    }
}
```

Slika 2.9. Implementacija komponente *viewmodel*

Na slici 2.10. prikazano je pozivanje metode *loadData* nakon unošenja teksta u tekstualni okvir. Nakon što se pozove metoda *loadData*, pokreće se funkcija Kotlin Coroutine unutar koje se vrši poziv na *getLibraryImages* metodu koja zatim kontaktira *model* kako bi dobila podatke s poslužitelja. Dobiveni podaci se zatim prikazuju u elementu korisničkog sučelja *RecyclerView.*

```
    private fun search(){
        binding.searchEditText.doAfterTextChanged {
            loadData(binding.searchEditText.text.toString())
        }
    }
    private fun loadData(keyword: String) {
        lifecycleScope.launchWhenCreated {
            val response = viewModel.getLibraryImages(keyword)
            if(response.isSuccessful && response.body() != null) {
                adapter.setItems(response.body()!!.collection.items)
            }else{
                Log.d(TAG, "loadData: failure")
            }
        }
    }
```

Slika 2.10. Pozivanje metode iz komponente *viewmodel* u *view* komponenti MVVM obrasca

U prethodnom primjeru vidi se konkretna implementacija MVVM obrasca u pristupu zasnovanom na označnom jeziku XML. Implementaciju dobivanja podataka s udaljenog poslužitelja odvaja se

u *model* sloj MVVM obrasca. Poziv na tu implementaciju definiran je u komponenti *viewmodel* koja se zatim poziva u *view* sloju. U slučaju da korisnik želi pohraniti podatke, komunikacija bi započela od *view* sloja, preko *viewmodel* sloja sve do *model* sloja koji se brine za komunikaciju s udaljenim poslužiteljem.

2.2.6. MVVM u pristupu zasnovanom na okviru Jetpack Compose

Kod pristupa zasnovanog na okviru Jetpack Compose postoje znatne razlike u korištenju MVVM obrasca u odnosu na pristup zasnovan na označnom jeziku XML. Razlike se najviše očituju u načinu dohvaćanja podataka. Implementacija u sloju *model* je jednaka implementaciji u pristupu zasnovanom na označnom jeziku XML. Na slici 2.11. prikazana je implementacija komponente *viewmodel*. Na prvi pogled su primjetne razlike u odnosu na implementaciju zasnovanu na označnom jeziku XML. Primjećuje se deklaracija varijable *_libraryImages* tipa i njezine varijable za dohvaćanje *libraryImages*. U deklaraciji varijable za dohvaćanje primjećuje se da je tip varijable *StateFlow* koji se brine za obavještavanje korisničkog sučelja da su se promjene dogodile. U metodi *getLibraryImages* pokreće se funkcija *Kotlin Coroutine* u kojoj se izvršava poziv na udaljenog poslužitelja kroz sloj *model*, te se postavlja vrijednost varijable *_libraryImages*.

```
class APIViewModel() : ViewModel() {
    /…/
    private val _libraryImages = MutableStateFlow(Base())
    val libraryImages: StateFlow<Base> get() = _libraryImages
    suspend fun getAPOD(){…}
    suspend fun getLibraryImages(keyword: String) {
        viewModelScope.launch {
            try {
                val libApi = LibraryRetrofit.nasaApi.getLibraryImages(keyword)

                Log.d(TAG, "Response: ${libApi.body()},  code: ${libApi.code()}")

                if(libApi.isSuccessful && libApi.body() != null) {
                    _libraryImages.value = libApi.body()!!;
                }
            } catch (e: Exception) {
                Log.d(TAG, e.toString())

            }
        }
    }
    suspend fun getMarsPhotos(earthDate: String) : Response<MarsPhoto> {…}
}
```

Slika 2.11. Implementacija komponente *viewmodel* u pristupu zasnovanom na okviru Jetpack Compose

Na slici 2.12. prikazana je implementacija *view* sloja MVVM obrasca na primjeru zaslona sa

slikama iz biblioteke agencije NASA. Ovdje se također primjećuje varijabla *viewModel* tipa *APIViewModel* za čiju se kreaciju i u ovom slučaju brine okvir za ubrizgavanje ovisnosti Koin, stoga se instanca dohvaća putem njega. U sloju *view* nakon što korisnik unese upit, pokreće se *Composable* funkcija *LaunchedEffect* koja prima blok koda koji će se izvesti po njezinom pokretanju. U ovom slučaju izvodi se kod iz sloja *viewmodel* zadužen za pozivanje sloja *model* preko koje se kontaktira udaljeni poslužitelj s upitom. Nema eksplicitnog postavljanja podataka već se oni postavljaju kroz varijablu *libImages* koja iz sloja *viewmodel* prima stanje varijable *libraryImages*, čiji se podaci zatim prikazuju u listi.

```
@Composable
@Destination
fun NASALibraryScreen() {
    val viewModel: APIViewModel = koinViewModel()
    val libImages = viewModel.libraryImages.collectAsState()
    var query by remember { mutableStateOf("") }
    var startedSearch by remember { mutableStateOf(false) }
    if(startedSearch) {
        LaunchedEffect(Unit) {
            if(query.isNotEmpty()) {
                viewModel.getLibraryImages(query)
            }
            startedSearch = false
        }
    }
    Column{
        Row (horizontalArrangement = Arrangement.Center) {…}
            LazyColumn {
                items(libImages.value.collection.items.size) {
                    LibraryImage(photo = libImages.value.collection.items[it])
                }
            }
    }
}
```

Slika 2.12. Implementacija komponente *viewmodel* u pristupu zasnovanom na okviru Jetpack Compose

Iako je moguće koristiti MVVM u pristupu zasnovanom na okviru Jetpack Compose, zbog reaktivne prirode okvira Jetpack Compose pruža se alternativa u obrascu MVI čija je značajka pristup pogonjen stanjima. MVI se sastoji od klasičnih komponenti *model* i *view*, novitet je komponenta *intent* kojom se predstavljaju korisničke radnje. MVI ima jednosmjeran tok podataka, što znači da se komunikacija između komponenata odvija samo u jednom smjeru, na primjer, korisnik pokrene akciju, komponenta *intent* prima akciju te zatim prosljeđuje stanje u *model* u kojem se odrađuje poslovna logika. Naposljetku se rezultat poslovne logike prosljeđuje u obliku stanja do komponente *view* te prezentira krajnjem korisniku. Uz to, MVI primjenjuje se kao upravitelj za stanja (engl. *State Management*).

2.3. Kratak prikaz izgradnje korisničkog sučelja na drugim platformama

Različite platforme često imaju alate koji su specijalizirani za izgradnju korisničkog sučelja samo na toj platformi, no postoje i hibridna rješenja koje se mogu koristiti na različitim platformama. Slijedi kratak prikaz alata koji se u današnje doba koriste za izgradnju korisničkog sučelja na popularnim platformama. Izgradnja korisničkog sučelja na web platformi u današnje doba provodi se korištenjem raznih biblioteka od kojih su najkorištenije React, Angular i Vue [10].

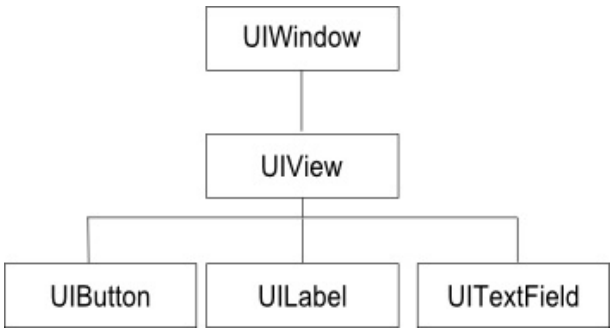
React je biblioteka otvorenog koda za izgradnju korisničkog sučelja pisana u programskom jeziku Javascript. React ima dvije glavne značajke. Deklarativnost omogućava lakše kreiranje interaktivnog korisničkog sučelja te efikasno ažuriranje komponenti pri promjeni podataka. Deklarativnost omogućuje kod koji je lakši za razumijevanje i otklanjanje pogrešaka, te predvidljiviji. Problem više jezika potrebnih za izgradnju korisničkog sučelja na Android platformi korištenjem pristupa zasnovanog na označnom jeziku XML, postoji i na web platformi. Biblioteka React taj problem rješava enkapsuliranjem Javascript koda, HTML koda i CSS koda u jednu komponentu koja koristi ekstenziju sintakse naziva Javascript XML. Korištenje Javascript XML ekstenzije omogućuje pisanje HTML i CSS koda unutar Javascript funkcija koje kao povratni tip imaju Javascript XML element. S obzirom da je cijela logika pisana u istom programskom jeziku, olakšano je prosljeđivanje podataka kroz aplikaciju. Filozofija biblioteke React je „nauči jednom, koristi svugdje“, s čime se referira na sposobnost biblioteke React da gradi korisničko sučelje i na mobilnoj platformi koristeći React Native [11].

Kao i React, Angular je biblioteka otvorenog koda pisana u programskom jeziku Typescript koja se također koristi za izgradnju korisničkog sučelja na web platformi. Typescript je jezik koji se temelji na programskom jeziku Javascript, koji kao glavnu značajku i razliku u odnosu na Javascript ima mogućnost deklariranja tipova podataka. Okvir Angular koristi komponente kao blokove od koji se gradi sučelje. Komponente se sastoje od CSS selektora i HTML elemenata koji odgovaraju tim selektorima. Problem kombiniranja različitih jezika potrebnih za izgradnju korisničkog sučelja rješava se proširivanjem HTML predložaka u kojima se mogu koristiti Javascript klase dekorirane s *@Component* dekoracijom u kojima se nalazi Javascript kod zadužen za manipulaciju podacima. U slučaju promjene podataka u Javascript klasi, zbog dekoracije *@Component*, sve promjene se odmah odražavaju na korisničkom sučelju definiranom u HTML predlošku. Dekoraciji *@Component* predaju se putanje do odgovarajućeg HTML predloška te CSS stilizacijske klase. U suštini, HTML predlošci zaduženi su za prikazivanje podataka u HTML

elementima, CSS stilizacijske klase za izgled HTML elemenata dok je dekorirana Javascript klasa zadužena za poslovnu logiku. Jedna od glavnih značajki okvira Angular je riješen problem ubrizgavanja ovisnosti. Angular koristi vlastiti okvir za ubrizgavanje ovisnosti što uvelike olakšava održavanje koda i povećava skalabilnost projekta [12].

Okvir Vue je također temeljen na Javascript programskom jeziku te pruža deklarativni načina izgradnje korisničkog sučelja i uporabu komponenti koje se mogu kombinirati kako bi se izgradilo kompleksno korisničko sučelje. Okvir Vue problem implementacije korisničkog sučelja rješava s komponentama naziva *Single-File Components* ili skraćeno SFC. Komponente SFC predstavljaju kombinaciju HTML predloška, implementacije logike u programskom jeziku Javascript i stilizacije CSS selektorima. Za svaki od dijelova SFC komponente koriste se različite oznake. Oznaka *script* koristi se kako bi se koristio programski jezik Javascript u kojem se implementira logika. Oznaka *template* je rezervirana za HTML predložak, a oznaka *style* za stilizacijski jezik CSS.

Druga popularna mobilna platforma uz Android je iOS tvrtke Apple. Korisničko sučelje gradi se pomoću pogleda (engl. *Views*) [16]. Na slici 2.13 prikazana je hijerarhija pogleda na iOS platformi. *UIWindow* je korijenska komponenta na kojoj se prikazuju sva njegova djeca. *UIView* je kontejner za *UIButton* i ostale klase koje omogućuju prikaz komponenti koje omogućuju interakciju korisnika s aplikacijom. Bitno je naglasiti kako sva djeca mogu imati samo jednog izravnog roditelja [17]. Prije uvođenja okvira SwiftUI, korisničko sučelje na iOS platformi gradilo se korištenjem okvira UIKit. Nakon uspjeha deklarativnih okvira za web platformu, uvodi se okvir SwiftUI kao zamjena za okvir UIKit. Korisničko sučelje koristeći okvir SwiftUI implementira se u cijelosti u programskom jeziku Swift, čime se uklanja potreba za upraviteljem *UIStoryboard* putem kojeg su se slagali elementi korisničkog sučelja te se zatim u klasama *ViewController* implementiralo ponašanje i ažuriranje podataka. Korisničko sučelje koristeći okvir SwiftUI implementira se prethodno definiranim funkcijama okvira SwiftUI koji predstavljaju elemente korisničkog sučelja. Također, moguće je kreirati i vlastite funkcije kojima se implementiraju dijelovi zaslona. Stoga, okvir SwiftUI može se gledati kao iOS pandan okviru Jetpack Compose koji se koristi na Android platformi.



Slika 2.13. Hijerarhija pogleda na iOS platformi [17]

Kao jedno od rješenja za izgradnju korisničkog sučelja na mobilnim platformama, mora se u obzir uzeti i Flutter. Okvir koji je u posljednjih nekoliko godina postao najkorištenije rješenje za izgradnju aplikacija na različitim platformama. U okviru koristi se objektno orijentirani programski jezik Dart. Flutter aplikacije se kompiliraju izravno u strojni jezik, zbog čega Flutter aplikacije ne gube previše na performansama. Samo korisničko sučelje gradi se od klasa koje se nazivaju elementi *widget*. Također, kao i kod okvira Jetpack Compose, zbog deklarativne paradigme i istog programskog jezika za izgradnju korisničkog sučelja i implementacije logike omogućeno je jednostavno ponovno korištenje elemenata, a olakšano je održavanje i testiranje. Slično kao okviri Jetpack Compose i SwiftUI, u okviru Flutter koriste se prethodno definirani elementi korisničkog sučelja naziva *widget*, samo oni u slučaju okvira Flutter elementi, umjesto putem funkcija, implementirani putem klasa.

Analizirajući prethodno navedena rješenja za web platforme zaključuje se da problem povezivanja samog korisničkog sučelja, prezentacije podataka i implementacije ponašanja nije prisutan samo na Android platformi, već je to problem koji obuhvaća sve platforme koje za prikaz podataka koriste grafičko korisničko sučelje. Iz perspektive web platformi može se vidjeti da deklarativni okviri postoje već duže vrijeme i da su slijedeći korak evolucije alata za izgradnju grafičkog korisničkog sučelja. Okvir Jetpack Compose predstavlja taj slijedeći korak evolucije alata za izgradnju korisničkog sučelja na Android platformi. Kao što su okviri React, Angular i Vue, iako slični, svaki na svoj način riješili problem povezivanja različitih sastavnica korisničkog sučelja, tako i okvir Jetpack Compose uvođenjem funkcija kao gradbenih blokova i objedinjavanjem koda u jedan jezik rješava manjkavosti pristupa zasnovanog na označnom jeziku XML. Važno je napomenuti da iako se čini da moderni deklarativni okviri kombiniranjem različitih sastavnica korisničkog sučelja u jednu komponentu narušavaju princip odvajanja ovisnosti, pravilnim korištenjem okvira postižu se labave veze koje pridonose skalabilnosti i proširivanju.

3. USPOREDBA POSTUPAKA IZGRADNJE KORISNIČKOG SUČELJA NA ANDROID PLATFORMI

Pristupi za izgradnju korisničkog sučelja na Android platformi koji se obrađuju u ovom radu imaju različite postupke koji se primjenjuju na aplikaciji naziva SpaceApp. Aplikacija ima dvije inačice, prva inačica izgrađena je korištenjem klasičnog pristupa zasnovanog na označnom jeziku XML, a druga korištenjem modernog pristupa koji koristi okvir Jetpack Compose.

3.1. Opis aplikacije i specifikacija zahtjeva

Svrha aplikacije je pružanje korisnicima mogućnosti dijeljenja i pregledavanja slika sa sadržajem svemira i svemirskih tijela. Korisnicima je omogućen centralizirani pristup bazi podataka agencije NASA korištenjem aplikacijskog programskog sučelja koje je dostupno na stranicama agencije NASA. Prilikom korištenja aplikacije korisnici se moraju registrirati koristeći elektroničku adresu i lozinku. Nakon registracije, korisnici imaju mogućnost prijave u aplikaciju te pregledavanja slika drugih korisnika te mogućnost dijeljenja vlastitih slika. Nadalje, korisnici imaju mogućnost označavanja slika oznakom „sviđa mi se“ pritiskom na gumb. Korisnici mogu komentirati vlastite slike ili slike drugih korisnika. Korisnici mogu pristupiti bazi podataka agencije NASA koja sadrži slike svemira i svemirskih tijela te pretraživati unošenjem ključnih riječi. Korisnici mogu pristupiti slikama s istraživačkih rovera agencije NASA koji se nalaze na Marsu. Omogućeno je pretraživanje prema datumu postavljanja slike na poslužitelj. Korisnici mogu pregledavati sliku „*Astronomy picture of the day“* i njezin opis. Slika se mijenja svakog dana i službena je slika dana agencije NASA.

Funkcionalnosti aplikacije koje treba implementirati i prioriteti implementacije istih prikazani su tablicom 3.1. S lijeve strane prikazana je funkcionalnost aplikacije, dok je s desne prikazan prioritet implementacije.

Tablica 3.1. Funkcionalnost aplikacije i prioriteti implementacije istih

NAZIV	**PRIORITET**
Registracija korisnika	Srednji
Prijava korisnika	Srednji
Prikaz fotografija	Visok
Pretraživanje baze podataka agencije NASA	Visok
Pretraživanje (npr. „Orion“)	Srednji
Dodavanje vlastitih fotografija	Visoki
Komentiranje fotografija	Srednji
Označavanje slika sa „sviđa mi se"	Niski

Za izradu aplikacija korištene su razne standardne biblioteke koje su uvelike olakšale izgradnju aplikacije. Tablica 3.2. prikazuje naziv biblioteke te kratki opis za što se koristi u implementaciji aplikacije.

Tablica 3.2. Standardne biblioteke korištene prilikom izrade aplikacije

NAZIV	OPIS
Retrofit	Biblioteka za rukovanje aplikacijskim programskim sučeljima
GSON	Biblioteka za parsiranje prijenosnog formata JSON u objekte
Jetpack Compose	Biblioteka za izradu deklarativnog korisničkog sučelja.
Google Firebase	Baza podataka u stvarnom vremenu za funkcionalnosti registracije i prijave korisnika.
Material Design	Biblioteka s komponentama korisničkog sučelja
Koin	Biblioteka za ubrizgavanje ovisnosti
Glide	Prikazivanje slika

Za dohvaćanje podataka s poslužitelja agencije NASA korištena su aplikacijska programska sučelja prikazana u tablici 3.3. S lijeve strane prikazan je naziv aplikacijskog programskog sučelja. Desna strana tablice daje nam kratki opis na koju uslugu se omogućuje pristup putem aplikacijskog programskog sučelja.

Tablica 3.3. Aplikacijska programska sučelja korištena za dohvaćanje slika s poslužitelja

NAZIV	OPIS
APOD	Dohvaća sliku dana sa stranice Astronomy Picture of the Day.
NASA Image and Video Library	Omogućava pretraživanje i dohvaćanje slika i videozapisa iz baze podataka agencije NASA.
Mars Rover Photos	Skuplja slike s rovera Curiosity, Spirit i Opportunity koji se nalaze na Marsu.

3.2. Slučajevi korištenja

Aplikacija ima različite slučajeve korištenja. Slučajevi korištenja i ishodi istih opisani su u tablici 3.4. Slučajevi korištenja opisani u tablici olakšavaju implementaciju funkcionalnosti i testiranje. Pod stupcem ulaz u tablici 3.4. prikazani su ulazi, odnosno radnje koje korisnik može napraviti. Drugi stupac naziva ishod predstavlja očekivani ishod odgovarajuće radnje korisnika.

Tablica 3.4. Slučajevi korištenja i ishodi istih

ULAZ	ISHOD
Korisnik se registrira koristeći elektroničku adresu i lozinku	Korisnik uspješno kreiran i pohranjen u bazu podataka
Korisnik se prijavljuje u aplikaciju	Korisnik uspješno prijavljen u aplikaciju
Korisnik pregledava sadržaj početnog zaslona aplikacije	Korisniku se prikazuje sav dostupan sadržaj na poslužitelju
Korisnik otvara pojedinačan element na početnom zaslonu	Prikazuje se sadržaj s detaljima i komentarima
Korisnik unosi komentar	Komentar se pohranjuje u bazu podataka i prikazuje u komentarima na zaslonu

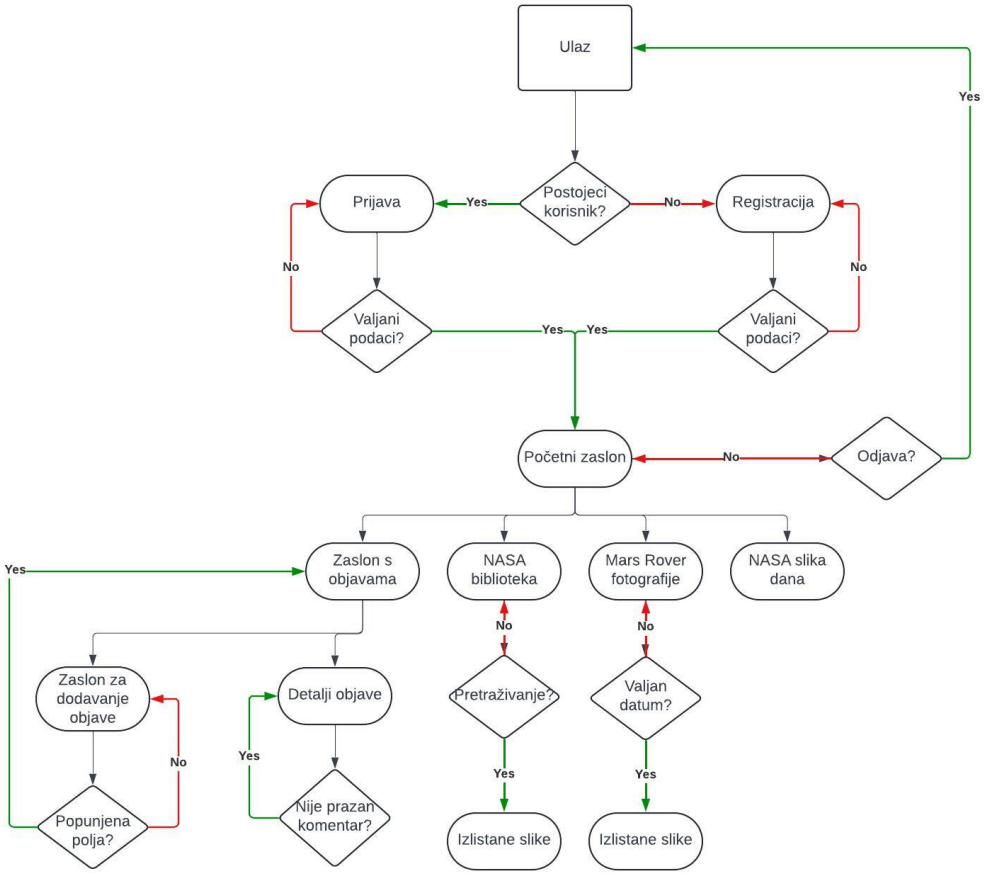
Korisnik otvara zaslon za dodavanje vlastitog sadržaja	Otvara se zaslon s praznim poljima za unos teksta i gumbom za odabir slike
Korisnik pritišće gumb za odabir slike	Otvara se galerija u kojoj korisnik odabire sliku koju želi objaviti
Korisnik pritišće gumb za objavu sadržaja	Sadržaj se pohranjuje u bazu podataka i prikazuje na početnom zaslonu
Korisnik odabire element „APOD“ na navigacijskoj traci	Otvara se zaslon sa slikom dana i učitava sadržaj
Korisnik odabire element „Mars Rover“ na navigacijskoj traci	Otvara se zaslon na kojem se nalazi dijalog za odabir datuma za pretraživanje slika
Korisnik unosi datum u dijalog	Korisniku se prikazuju slike koje su snimljene na uneseni datum
Korisnik odabire element „NASA Library“ na navigacijskoj traci	Otvara se zaslon na kojem se nalaze slike iz baze podataka agencije NASA i okvir za pretraživanje
Korisnik unosi unos u okvir za pretraživanje	Korisniku se prikazuju rezultati pretraživanja
Korisnik pritišće gumb za odjavu	Korisnik se odjavljuje i aplikacija se zatvara
Korisnik pritišće na gumb „sviđa mi se“	Sadržaj se označuje sa „sviđa mi se“ i mijenja se boja gumba

3.3. Prikaz rada aplikacije

Kroz dijagram toka, slikovni prikaz i praktični opis korištenja aplikacije prikazan je rad aplikacije i njezin izgled. Kako bi se korisnik mogao uspješno koristiti aplikacijom detaljno su opisani postupci dolaženja do željenog rezultata korištenja neke od funkcionalnosti aplikacije. Također, kako bi se prikazalo da se korištenjem oba okvira za izgradnju korisničkog sučelja na Android platformi, okvirom Jetpack Compose i označnim jezikom XML, može izgraditi identično korisničko sučelje, koristi se kombinacija fotografija korisničkog sučelja aplikacija koje su izgrađene s navedenim okvirima.

3.3.1. Dijagram toka

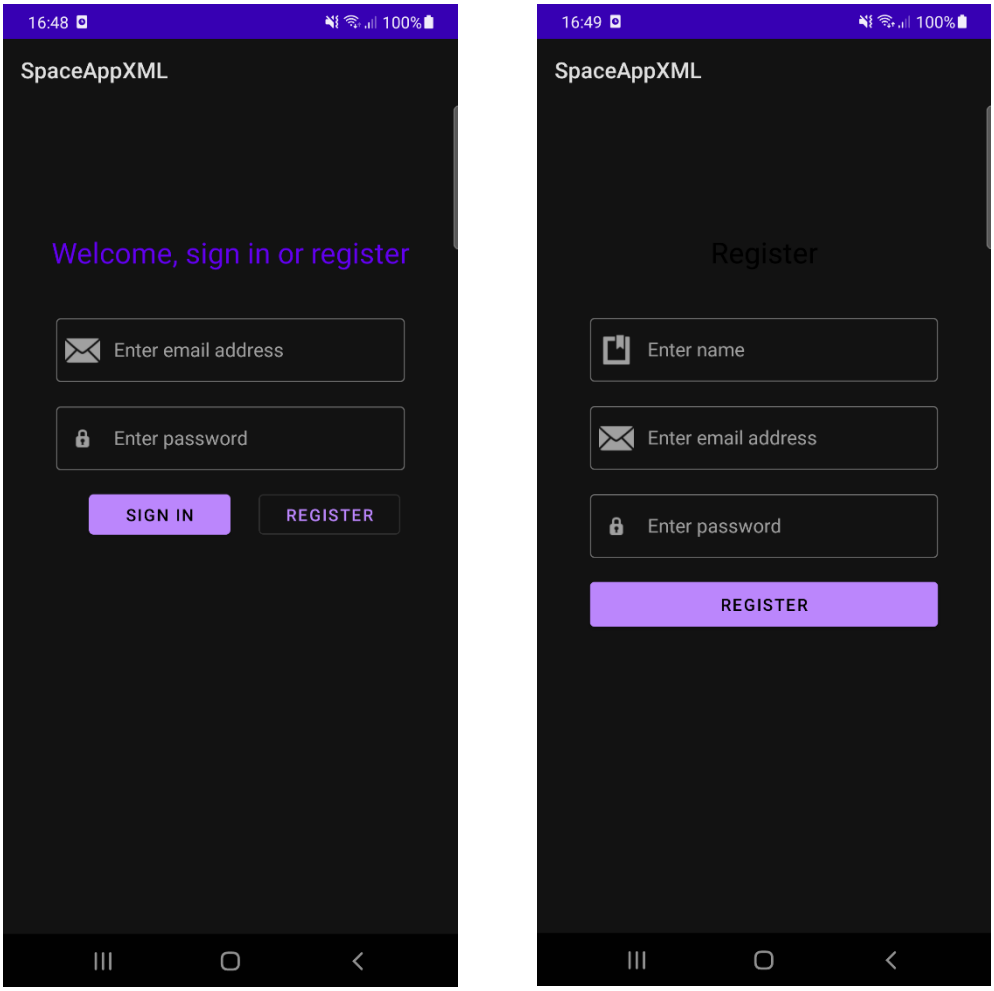
Na slici 3.1. prikazan je dijagram toka aplikacije. Četverokut s oznakom „Ulaz“ predstavlja ulaznu točku aplikacije. Ovalni oblici poput „Prijava“ i „Registracija“ su zasloni aplikacije, a rombovi su čvorišta akcija korisnika. Zelene linije predstavljaju pozitivan ishod akcije, dok su crvene linije negativan ishod.



Slika 3.1. Dijagram toka aplikacije

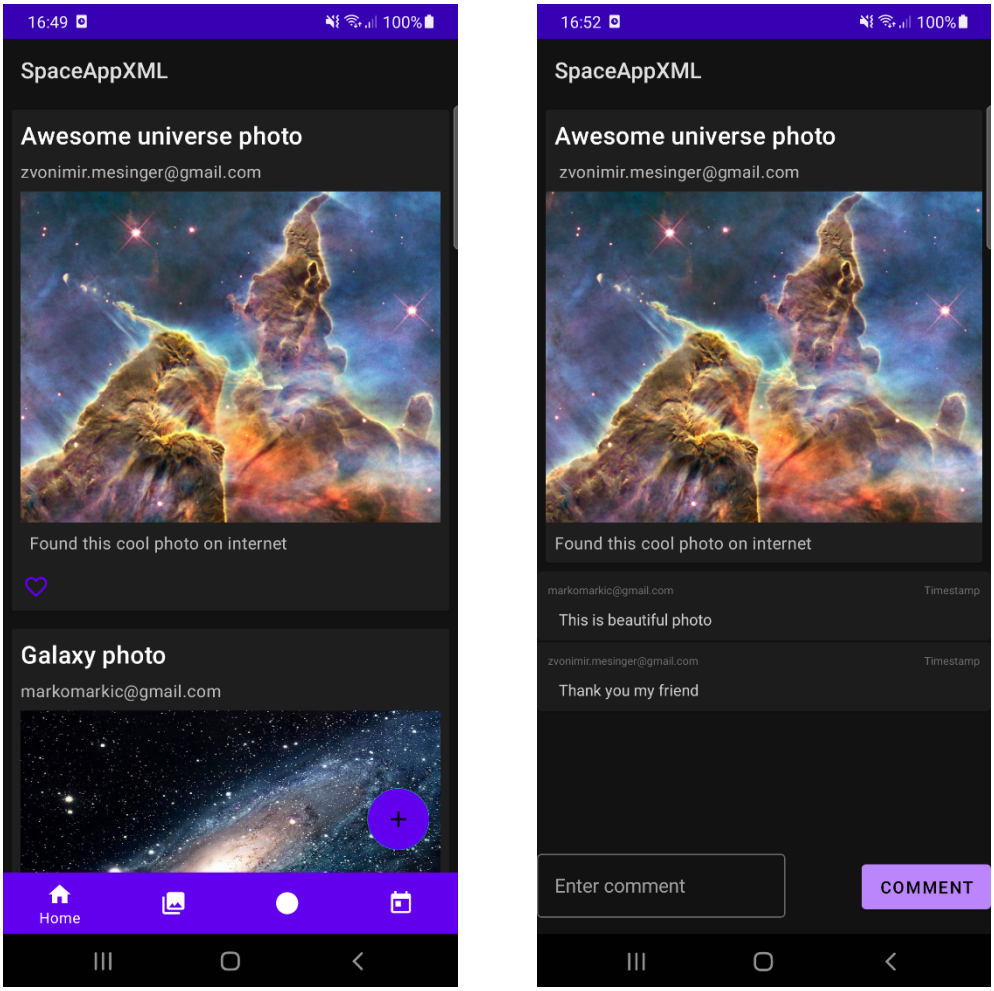
3.3.2. Korištenje aplikacije

Nakon što se aplikacija pokrene, otvara se zaslon za prijavu korisnika prikazan na slici 3.2a. Na zaslonu se nalaze tekstualni okviri za unos elektroničke adrese i korisničke lozinke. Pritiskom na gumb „*Sign in*“ korisnik se prijavljuje u aplikaciju i odlazi na početni zaslon. Pritiskom na gumb „*Register*“, odlazi se na zaslon za registraciju prikazan na slici 3.2b. Na zaslonu za registraciju korisnika, korisnik unosi ime koje želi da se prikazuje, elektroničku adresu i korisničku lozinku. Pritiskom na gumb „*Register*“ korisnik se registrira i odlazi na početni zaslon.



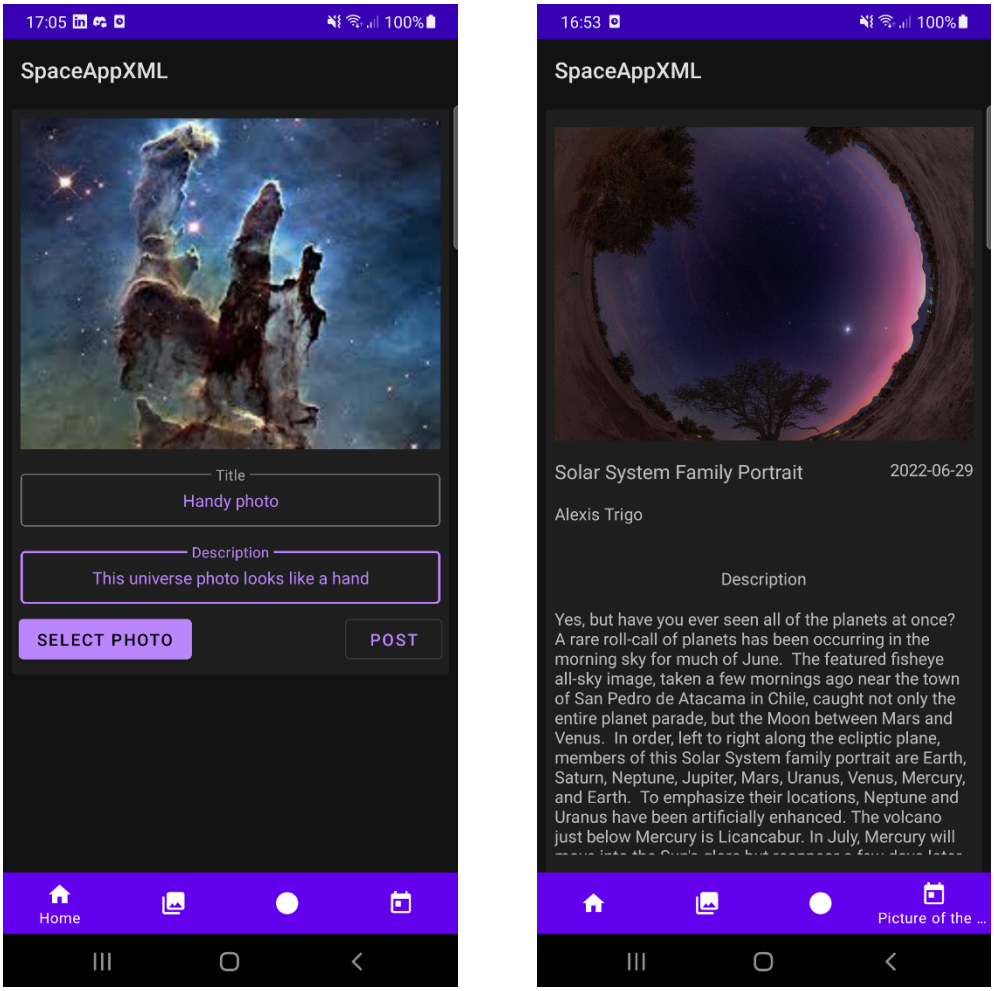
Slika 3.2. Zasloni za prijavu (a) i registraciju korisnika (b)

Na početnom zaslonu prikazanom na slici 3.3a. nalazi se sadržaj koji objavljuju korisnici aplikacije, pritiskom na sliku korisnik odlazi na zaslon s detaljima o slici i komentarima prikazan na slici 3.3b. Pritiskom na gumb „+“, korisnik odlazi na zaslon gdje može odabrati sliku i postaviti vlastiti sadržaj na poslužitelj. Pritiskom na elemente navigacijske trake, korisnik odlazi na zaslon sa slikom dana. Na zaslonu s detaljima i komentarima korisnik može pregledavati komentare i dodavati vlastite komentare unošenjem teksta u tekstualni okvir i pritiskom na gumb „*Comment*“. Također, prikazani su pojedinosti objave koja se prikazuje.



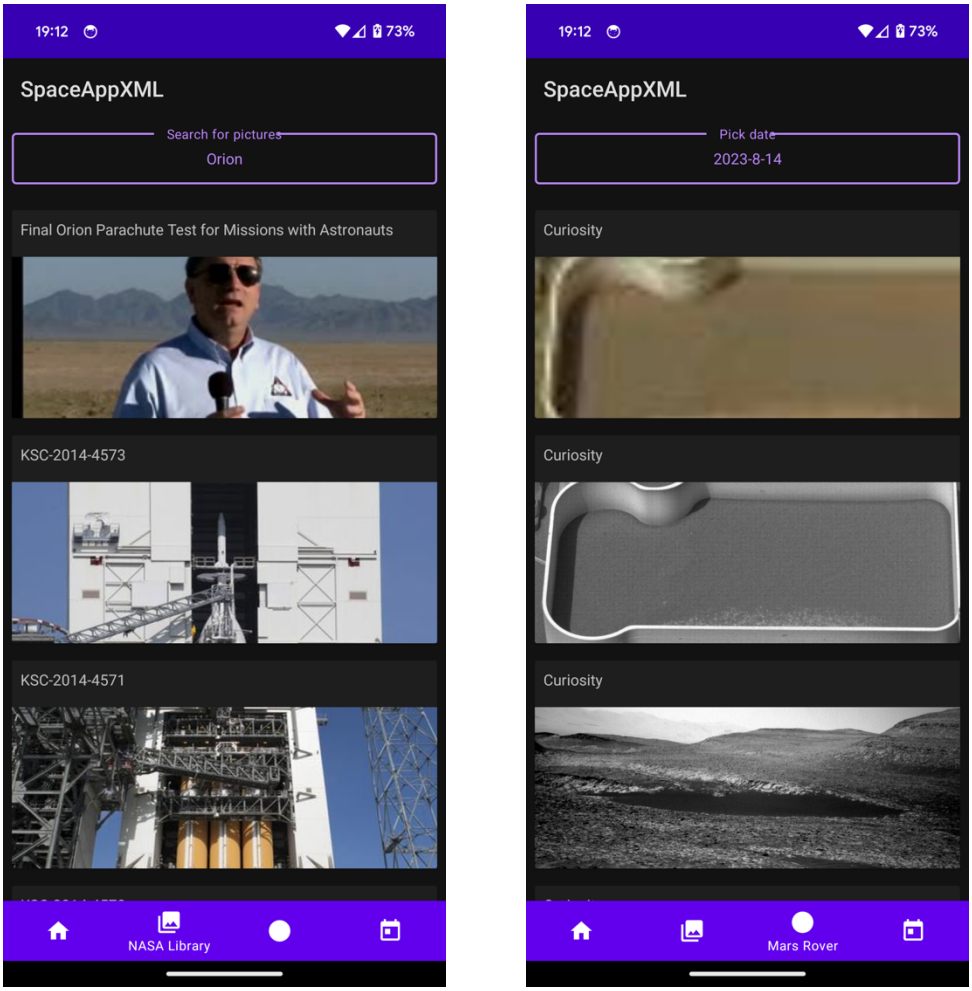
Slika 3.3. Početni zaslon (a) i zaslon s detaljima (b)

Zaslon za dodavanje novog sadržaja prikazanom na slici 3.4a omogućuje korisniku odabir slike koju želi podijeliti pritiskom na gumb „*Select photo*“, unošenjem naslova fotografije i kratki opis iste. Nakon pritiska na „*Select photo*“ otvara se galerija uređaja te je potrebno odabrati sliku koju se želi objaviti. Pritiskom na gumb „*Post*“ korisnik objavljuje sliku na poslužitelj i ona se pojavljuje na početnom zaslonu. Na zaslonu sa slikom dana prikazanom na slici 3.4b prikazana je slika koju je agencija NASA odabrala za sliku dana, datum, autor i opis slike.



Slika 3.4. Zaslon za dodavanje novog sadržaja (a) i zaslon sa slikom dana (b)

Zaslon na slici 3.5a. je lista fotografija koja se dobije na upit „Orion“. Prikazana je fotografija te naslov pojedinačne fotografije. Kako bi se dobili podaci, potrebno je odabrati tekstualni okvir s oznakom „Search for pictures“ te unijeti upit na koji želimo dobiti slike iz NASA biblioteke. Slika 3.4b. prikazuje rezultat filtriranja Mars Rover biblioteke po datumu. Rezultat upita su sve slike koje su snimljene roverima koji se nalaze na planetu Marsu. Datum na ovom primjeru je 14. kolovoza 2023. Kako bi se poslao upit na Mars Rover biblioteku, potrebno je odabrati element korisničkog sučelja s oznakom „Pick date“, nakon čega se otvara skočni prozor u kojem je prikazan kalendar. Odabirom datuma prije datuma dana u kojem se odrađuje upit, šalje se validan upit na Mars Rover biblioteku i prikazuju se fotografije koje su snimljene i poslane na taj dan. Ako korisnik kojim slučajem odabere datum koji se nalazi u budućnosti, upit se neće izvesti te će se prikazati prazna lista s porukom pogreške.



Slika 3.5. Zasloni NASA biblioteke (a) i Mars Rover biblioteke (b)

3.4. Usporedba pristupa izgradnje korisničkog sučelja

Izgradnju korisničkog sučelja može se promatrati s gledišta razvoja, korištenja i održavanja.

Iz perspektive razvoja korisničkog sučelja, razlika između ova dva pristupa prvenstveno se očituje u paradigmi kojom se vode okviri. Označni jezik XML vodi se imperativnim pristupom što znači da se elementi korisničkog sučelja inicijalizirani u kodu te se na njih postavljaju promatrači za događaje definirani u programskim jezicima Java i Kotlin. Kako bi se aktualizirali podaci koji su rezultat nekog događaja, potrebno je ručno osvježavati korisničko sučelje. Za razliku od pristupa koji se koristi označnim jezikom XML, pristup zasnovan na modernom okviru Jetpack Compose vodi se deklarativnom paradigmom koja kao ideju ima opisati kako bi korisničko sučelje trebalo izgledati bez detaljnog opisivanja svih koraka potrebnih za prikaz krajnjeg rezultata.

U praksi, kreiranje korisničkog sučelja korištenjem označnog jezika XML, odnosno okvirom Jetpack Compose, prikazano je u nastavku.

Primjer prikazan slikom 3.6. korišten je za definiranje raspored elemenata *layout* s detaljima i komentarima. On prikazuje korijenski element tog zaslona. *LinearLayout* je *ViewGroup* objekt koji prikazuje elemente jedan za drugim, može imati orijentaciju horizontalno ili vertikalno, na ovom primjeru ona je vertikalna. Visina i širina rasporeda elemenata *layout* jednaki su veličini zaslona uređaja.

```
<LinearLayout 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.ui.fragments.PostDetailFragment"
    android:orientation="vertical"
```

Slika 3.6. Prikaz definiranja rasporeda elemenata *layout*

Nakon korijenskog elementa dodaju se ostali elementi ili po potrebi još rasporeda elemenata *layout*. U slučaju sa slike 3.7., za opisivanje korisničkog sučelja potreban je dodatni raspored elemenata *layout*, nakon njega dolaze elementi korisničkog sučelja. Na primjeru prikazana je tekstualna oznaka za koju je potrebno definirati jedinstveni identifikator *id* prema kojem mu se omogućava pristup.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:id="@+id/titleTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/title"
        android:paddingTop="8dp"
        android:paddingStart="8dp"
        android:textAppearance="?attr/textAppearanceHeadline6"
        />
```

Slika 3.7. Prikaz definiranja rasporeda elemenata *layout*

Moguće je koristiti element korisničkog sučelja. U sljedećem primjeru korišten je *view binding. View binding* značajka je kojom se olakšava pisanje koda zaduženog za interakciju s elementima korisničkog sučelja definiranim u označnom jeziku XML. Značajkom *View binding*, za svaku datoteku u označnom jeziku XML generira se vezna klasa (engl. *binding class*). U veznoj klasi sadržana je referenca na sve elemente s pripadajućim jedinstvenim identifikatorom. Na slici 3.8. u metodi *registerNewUser* putem objekta naziva *binding,* pristupa se elementu korisničkog sučelja

koji ima jedinstveni identifikator *id registerFragmentButton* i na njemu postavlja se slušatelj koji kada se pozove, izvodi akciju registriranja korisnika.

```
private fun registerNewUser(){
    binding.registerFragmentButton.setOnClickListener { viewModel.register() }
}
```

Slika 3.8. Prikaz definiranja rasporeda elemenata *layout*

U slučaju kompleksnih korisničkih sučelja potrebno je više rasporeda elemenata *layout*, kombiniranjem istih može se postići željeni izgled. S gledišta održavanja tu se javlja problem. Kako bi postigao željeni izgled potrebno je ugnijezditi više rasporeda elemenata *layout* i elemenata *widget*. Taj proces se može ponoviti nekoliko puta, rezultat toga je raspored elemenata *layout* koji ima više rasporeda elemenata *layout* unutar njega. Takav raspored elemenata *layout* nije održiv te su mu smanjene čitljivost i razumljivost.

U modernom okviru Jetpack Compose, cijelo korisničko sučelje, odnosno zaslon, je funkcija anotirana anotacijom *Composable* koja u svojem tijelu sadrži pozive drugim anotiranim funkcijama. Pozivanjem takve funkcije odvija se rekompozicija korisničkog sučelja. Primjer korisničkog sučelja izvedenog u okviru Jetpack Compose je zaslon u kojem se vrši registracija novih korisnika aplikacije.

Na slici 3.12. prikazana je funkcija koja opisuje korisničko sučelje zaslona za registraciju novih korisnika. Anotirana funkcija *RegisterScreen* kao argument prima *RegisterViewModel* koji se ubrizgava koristeći okvir za ubrizgavanje ovisnosti Koin. U tijelu ove funkcije nalazi se još jedna funkcija *Surface* koja je dio okvira Jetpack Compose. Unutar funkcije *Surface* nalaze se ostale funkcije koje opisuju korisničko sučelje a sama funkcija predstavlja podlogu na kojoj se nalaze ostali elementi korisničkog sučelja. Kako bi prilagodili izgled podloge, kao argumente funkcije *Surface* predaju se modifikator kojim definiramo veličinu podloge te boju. Također, deklarirane su i dvije varijable potrebne za registraciju korisnika. Zbog korištenja funkcije *rememberSaveable* nije se potrebno brinuti o spremanju stanja varijabla jer se taj problem rješava unutar okvira Jetpack Compose.

```
@Composable
@Destination
fun RegisterScreen(
    registerViewModel: RegisterViewModel = koinViewModel()
) {
    var username: String by rememberSaveable {
        mutableStateOf("")
    }
    var password: String by rememberSaveable {
        mutableStateOf("")
    }

    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colors.background
    ) {…}
}
```

Slika 3.9. Prikaz funkcije korisničkog sučelja zaslona za registraciju

3.4.2. Ponovno korištenje elemenata

Iznimna prednost okvira *Jetpack Compose* je to što omogućuje ono za što su funkcije izvorno namijenjene, a to je ponovno korištenje dijelova koda. U ovom projektu, ta prednost može se primijetiti kod izgradnje korisničkog sučelja zaslona za prijavu korisnika i zaslona registracije novih korisnika. S obzirom da su ta dva zaslona slična i dijele elemente, dio korisničkog sučelja u koji se upisuju adresa elektroničke pošte i lozinka korisnika, izveden je kao zasebna funkcija anotirana anotacijom *Composable* te se koristi u prethodno spomenutim zaslonima.

Slika 3.10. prikazuje implementaciju funkcije koja se koristi na zaslonu za prijave te zaslonu za regstraciju korisnika. Funkcija se sastoji od dva tekstualna polja i dvije varijable za korisničko ime i lozinku. Kako bi se osiguralo aplikaciju od nepotrebnog postavljanja vrijednosti u komponenti *viewmodel* za prijavu korisnika, uz komponentu *viewmodel* za registraciju i prijavu ubirzganih od strane okvira za ubrizgavanje ovisnosti Koin*,* funkciji se predaje varijabla provjere radi li se o prijavi korisnika. Funkcionalnost same prijave i registracije implementira se u zaslonima za prijavu i registraciju, dok se u ovom dijelu koda postavljaju vrijednosti varijabli potrebnih za uspješno izvođenje.

Izdvajanjem ovog dijela koda u funckiju osigurava se smanjenje količine koda potrebnog za implementaciju ta dva zaslona, povećavaju se čitljivost i razumljivost koda te olakšava održavanje zaslona na kojima se funkcija koristi.

```
@Composable
fun UserInputForm(
    isLogin: Boolean,
    signInViewModel: SignInViewModel = koinViewModel(),
    registerViewModel: RegisterViewModel = koinViewModel()
){
    var username: String by rememberSaveable { mutableStateOf("") }
    var password: String by rememberSaveable { mutableStateOf("") }
    OutlinedTextField(
        value = username,
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email),
        onValueChange = {
            username = it
            if(isLogin) {
                signInViewModel.setEmail(username)
            }
            registerViewModel.setEmail(username)
        },
        label = { Text(text = stringResource(id = R.string.email_address)) },
        leadingIcon = {
            Icon(
                Icons.Default.Email,
                contentDescription = "",
            )
        },
        modifier = Modifier.padding(vertical = 12.0.dp)
    )
    OutlinedTextField(
        value = password,
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password),
        onValueChange = {
            password = it
            if(isLogin) {
                signInViewModel.setPassword(password)
            }
            registerViewModel.setPassword(password)
        },
        label = { Text(stringResource(id = R.string.password)) },
        leadingIcon = {
            Icon(
                Icons.Default.Lock,
                contentDescription = "",
            )
        },
        modifier = Modifier.padding(vertical = 12.0.dp)
    )
}
```

Slika 3.10. Prikaz implementacije funkcije za unos korisničkog imena i lozinke

Slika 3.11. prikazuje korištenje funkcije za unos korisničkog imena i lozinke na zaslonu za prijavu korisnika. Korištenje na zaslonu za registraciju isto je kao i na ovom zaslonu. Zaslon za prijavu korisnika sastoji se od teksta dobrodošlice, forme za unos korisničkog imena i lozinke te gumba za prijavu koji se nalazi unutra retka čiji je kod skriven zbog preglednosti.

```
@Composable
@Destination
@RootNavGraph(start = true)
fun LoginScreen(
    navigator: DestinationsNavigator,
    signInViewModel: SignInViewModel = koinViewModel(),
) {
    Surface (
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colors.background
    ) {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            WelcomeText()
            UserInputForm(isLogin = true)
            Row(
                modifier = Modifier.wrapContentSize(),
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.spacedBy(Dp(20F))
            ) {…}
        }
    }
}
```

Slika 3.11. Prikaz korištenja funkcije za unos korisničkog imena i lozinke

Ponovno korištenje elemenata korisničkog sučelja moguće je i korištenjem označnog jezika XML. Međutim, tijekom izvođenja takvih elemenata, manifestira se problem povećane kompleksnosti i čitljivosti koda korisničkog sučelja zbog ugniježđivanja rasporeda elemenata unutar drugog rasporeda elementa. Nadalje, ponovno korištenje elemenata u označnom jezika XML limitirano je iz razloga što kako bi osigurali dinamičnost takvih elemenata, potrebno je njihove promjene implementirati u programskom jeziku Java ili Kotlin. Uz prehtodno navedeno, ugniježđeni rasporedi elemenata pridonose otežanom otklanjanju pogrešaka u kodu, što rezultira povećanjem troška održavanja aplikacije. Otežano otklanjanje pogrešaka manifestira se u otežanoj navigaciji kroz ugniježđene XML datoteke.

Za razliku od označnog jezika XML, u pristupu zasnovanom na modernom okviru Jetpack Compose omogućuje se ponovno korištenje elemenata korisničkog sučelja na isti način na koji to pruža i programski jezik Kotlin. Rezultat toga je kod koji je u cijelosti pisan u jednom programskom jeziku. Time se osigurava jednostavna implementacija dinamičnosti elemenata, čitljivost koda i puno lakše otklanjanje problema. Otklanjanje pogrešaka u pristupu zasnovanom na okviru Jetpack Compose olakšava se jer kombiniranjem prezentacijske logike i elemenata

korisničkog sučelja u ponovnom korištenju dijelova koda omogućava se lakše razumijevanje koda, zbog čega je olakšano i pronalaženje pogrešaka.

Uz to, kod kompleksnih korisničkih sučelja koja su implementirana u modernom okviru Jetpack Compose, performanse su bolje nego kod korisničkih sučelja implementiranih u označnom jeziku XML. Ta činjenica pogotovo dolazi do izražaja kada se radi o uređajima koji imaju limitrane resurse.

3.4.3. Prikaz podataka u listama

Krajnji korisnik, u slučaju aplikacija čovjek, navikao je na prikaz podataka u listama. Iz tog razloga, liste su sastavni dio svake mobilne aplikacije. Jedna od ključnih razlika između pristupa zasnovanom na označnom jeziku XML i pristupa zasnovanog na modernom okviru Jetpack Compose je upravo implementacija prikaza podataka u listama.

Kako bi se podaci prikazali u listi koristeći označni jezik XML, koristi se element naziva *RecyclerView*. Glavna značajka kao što mu i ime govori, je recikliranje pogleda u kojem se mijenjaju samo podaci. Na primjer, na zaslonu može se prikazati maksimalno 10 elemenata liste, no lista sa stvarnim podacima može sadržavati tisuće elemenata. Da bi se riješio problem kreiranja pogleda svakog od unosa liste, element *RecyclerView* kreira fiksni broj elemenata te im mijenja samo sadržaj, odnosno učitava nove unose podataka iz liste podataka koji se prikazuju.

Implementacija listi korištenjem pristupa zasnovanog na označnom jeziku XML kompleksan je proces koji zahtjeva kreiranje više klasa i definiranje elementa liste. Proces kreće od definiranja elementa liste korištenjem označnog jezika XML.

Na slici 3.12. prikazan je definirani izgled jednog elementa liste. Sastoji se od tekstualnog polja te fotografije koji su vertikalno raspoređeni u rasporedu elemenata. Naziv ove datoteke u označnom jeziku XML je „*item_library_image.xml*“ te će se dalje koristiti u tom obliku.

```
<com.google.android.material.card.MaterialCardView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/cardView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_margin="8dp"
    android:elevation="2dp"
    app:cardCornerRadius="2dp"
    android:clickable="false">

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

    <TextView
        android:id="@+id/titleTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"/>

    <ImageView
        android:id="@+id/libraryImageView"
        android:layout_width="match_parent"
        android:layout_height="150dp"
        android:layout_marginTop="8dp"
        android:scaleType="centerCrop"
        android:src="@drawable/ic_photo_library"
        android:contentDescription="@string/nasa_image_library_item"
        />
    </LinearLayout>
</com.google.android.material.card.MaterialCardView>
```

Slika 3.12. Definiranje elementa liste

Sljedeći korak je definiranje adaptera i gledatelja *viewholder*. Adapter je klasa koja pruža povezivanje podataka koji se prikazuju i pogleda u kojima se oni trebaju prikazati. Gledatelj *viewholder* odgovaran je za opisivanje pojedinačnog elementa liste i njegovo mjesto unutar nje.

Slika 3.13. prikazuje implementaciju adaptera koji se koristi pri prikazivanju dohvaćenih fotografija s aplikacijskog programskog sučelja. Za implementaciju potrebno je pregaziti tri metode. Metoda *onCreateViewHolder* brine se za napuhivanje elementa liste definiranog u označnom jeziku XML. Da bi element *RecyclerView* znao ukupan broj elemenata, potrebno je pregaziti metodu *getItemCount*. Metoda u kojoj se napuhanom elementu liste pridodaju podaci zove se *onBindViewHolder*. Metoda *setItems* brine se o ažuriranju podataka unutar liste koja se prikazuje. Po promjeni podataka potrebno je obavijestiti adapter.

```
class LibraryListAdapter: RecyclerView.Adapter<LibraryListViewHolder>() {
    private val items = mutableListOf<Item>()
    fun setItems(items: List<Item>){
        this.items.clear()
        this.items.addAll(items)
        this.notifyDataSetChanged()
    }
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
LibraryListViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_library_image, parent, false)

        return LibraryListViewHolder(view)
    }
    override fun onBindViewHolder(holder: LibraryListViewHolder, position: Int) {
        val item = items[position]
        holder.bind(item)
    }
    override fun getItemCount(): Int {
        return items.count()
    }
}
```

Slika 3.13. Implementacija adaptera

Posljednji dio slagalice, prikazan na slici 3.14., potreban za prikaz liste podataka je klasa gledatelja *ViewHolder*. U klasi *LibraryListViewHolder* se tekstualnom polju i fotografiji postavljaju podaci koji su predani pregaženoj metodi adaptera *onBindViewHolder*. Jedino što preostaje je postaviti element *RecyclerView* u fragmentu i predati mu podatke

```
class LibraryListViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    fun bind(item: Item){
        val photo = item.links[0].href
        val binding = ItemLibraryImageBinding.bind(itemView)
        binding.titleTextView.text = item.data[0].title
        Glide.with(itemView)
            .load(photo)
            .into(binding.libraryImageView)
    }
}
```

Slika 3.14. Klasa gledatelja *ViewHolder*

Slika 3.15. predstavlja posljednji korak implementacije liste u pristupu zasnovanom na označnom jeziku XML. Pregažena metoda *onCreateView* poziva se pri kreiranju fragmenta te se tada i postavlja element *RecyclerView*. Kako bi se element *RecyclerView* popunio s podacima, u metodi *search* koja se poziva nakon upisivanja teksta u tekstualno polje, poziva se metoda *loadData* kojoj se predaje tekst prema kojem se filtrira baza podataka na poslužitelju. Po uspješnom odgovoru aplikacijskog programskog sučelja postavlja se vrijednost liste adaptera s vrijednostima primljenim od strane poslužitelja.

```
class LibraryImagesFragment : Fragment() {

    private lateinit var binding: FragmentLibraryImagesBinding
    private lateinit var adapter: LibraryListAdapter
    private val viewModel: APIViewModel by viewModels()

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = FragmentLibraryImagesBinding.inflate(inflater, container, false)
        setupRecyclerView()
        search()

        return binding.root
    }
    private fun search(){
        binding.searchEditText.doAfterTextChanged {
            loadData(binding.searchEditText.text.toString())
        }
    }
    private fun setupRecyclerView() {
        binding.libraryRecyclerView.layoutManager = LinearLayoutManager(
            context,
            LinearLayoutManager.VERTICAL,
            false
        )
        adapter = LibraryListAdapter()
        binding.libraryRecyclerView.adapter = adapter
    }
    private fun loadData(keyword: String) {
        lifecycleScope.launchWhenCreated {
            val response = viewModel.getLibraryImages(keyword)
            if(response.isSuccessful){
                adapter.setItems(response.body()!!.collection.items)
            }else{
                Log.d(TAG, "loadData: failure")
            }
        }
    }
}
```

Slika 3.15. Implementacija fragmenta s listom podataka

Iz gledišta održivosti, ovakva implementacija predstavlja problem zbog velike količine tzv. *boilerplate* koda. Koda koji je potrebno pisati no on je postao standardan i ponavlja se kod svake implementacije liste podataka. Kao posljedica više klasa potrebnih za izvedbu liste podataka javlja se problem pri promjeni liste. Kako bi se modificirao postojeći kod, potrebno je na više mjesta praviti izmjene, što rezultira potencijalnim greškama u kodu.

Pristup zasnovan na modernom okviru Jetpack Compose rješava problem elementa *RecyclerView*. Anotirana funkcija *LazyColumn* funkcionira na istom principu recikliranja elemenata te mijenjanja njihovog sadržaja. Horizontalni ekvivalent funkcije *LazyColumn* je funkcija *LazyRow*. Za

implementaciju liste u pristupu zasnovanom na modernom okviru Jetpack Compose potrebno je definirati element liste te napuniti listu podacima.

Slika 3.16. prikazuje implementaciju elementa u pristupu zasnovanom na okviru Jetpack Compose. Funkcija *LibraryImage* prima podatkovni element liste te prikazuje fotografiju i naslov fotografije.

```
@Composable
fun LibraryImage(photo: Item) {
    Card {
        val imageRequest = ImageRequest.Builder(LocalContext.current)
            .(…)
        Column(
            modifier = Modifier.(…)
        ) {
            Text(text = photo.data[0].title, modifier = Modifier.padding(all =
8.dp))
            AsyncImage(
                model = imageRequest,
                contentDescription = null,
                contentScale = ContentScale.Crop,
                modifier = Modifier.fillMaxSize()
            )
        }
    }
    Spacer(modifier = Modifier.height(10.dp))
}
```

Slika 3.16. Implementacija elementa liste

Na slici 3.17. prikazan je posljednji korak implementacije liste korištenjem okvira Jetpack Compose. Prilikom unošenja teksta u tekstualno polje, varijabla *startedSearch* postavlja se na istinitu vrijednost te se pokreće asinkroni poziv unutar bloka funkcije *LaunchedEffect*. Nakon dohvaćanja podataka s poslužitelja, postavlja se varijabla *libImages* koja se zatim predaje funkciji *items* gdje se kreiraju elementi liste čiji se izgled prethodno definirao.

Iz ovog primjera može se primijetiti kako kod implementacije liste u pristupu zasnovanom na okviru Jetpack Compose dolazi do znatnog smanjena količine koda. Rezultat tog smanjenja količine koda je smanjen trošak održavanja te lakše promjene. Ukoliko je potrebna izmjena ili proširenje postojećeg koda u listama, zbog smanjenja koda, olakšanja tih procesa ispoljavaju je u broju mjesta na kojima je potrebno proširiti ili izmijeniti kod. U slučaju izmjene postojećeg koda elementa liste implementirane pristupom zasnovanom na okviru Jetpack Compose, tu izmjenu je potrebno učiniti samo na jednom mjestu, dok se u slučaju pristupa zasnovanog na označnom jeziku XML ta izmjena treba napraviti u svakoj od sastavnica kojima se prikazuje lista.

```
@Composable
@Destination
fun NASALibraryScreen() {
    val viewModel: APIViewModel = koinViewModel()
    val libImages = viewModel.libraryImages.collectAsState()
    var query by remember { mutableStateOf("") }
    var startedSearch by remember { mutableStateOf(false) }
    if(startedSearch) {
        LaunchedEffect(Unit) {
            if(query.isNotEmpty()) {
                viewModel.getLibraryImages(query)
            }
            startedSearch = false
        }
    }
    Column{
        Row (horizontalArrangement = Arrangement.Center) {
            OutlinedTextField(value = query, onValueChange = {
                query = it
                startedSearch = true
            },
                label = { Text("Search for pictures") }
            )
        }
            LazyColumn {
                items(libImages.value.collection.items.size) {
                    LibraryImage(photo = libImages.value.collection.items[it])
                }
            }
    }
}
```

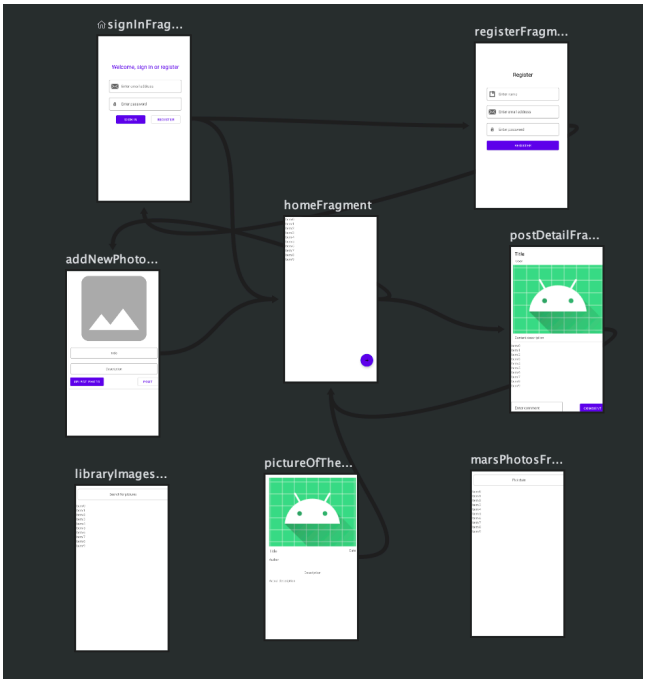
Slika 3.17. Implementacija liste u okviru Jetpack Compose

3.4.3. Navigacija kroz aplikaciju

Navigacija kroz aplikaciju sastavni je dio svake aplikacije. Tako ova dva pristupa koriste posebne biblioteke koje se brinu za to, u pristupu zasnovanom na označnom jeziku XML koristi se biblioteka imena Navigation Component, dok se u pristupu zasnovanom na okviru Jetpack Compose koristi biblioteka Navigation Compose. Kroz povijest pristupa zasnovanog na označnom jeziku koristila se navigacija putem klase *Intent* koja se zatim zamjenjuje bibliotekom Navigation Component. Da bi se postigla navigacija koristeći biblioteku Navigation Component prvo se definira navigacijski graf (engl. *navigation graph*) čiji se primjer prikazuje na slici 3.18.

Navigacijski graf definira se dodavanjem svih mogućih destinacija koje se pojavljuju u aplikaciji u njega. Nakon dodavanja svih destinacija, definiraju se veze između destinacija. Veze između destinacija predstavljaju se kao navigacijske radnje koje korisnik može izvesti unutar aplikacije. Ukoliko postoji potreba za prosljeđivanjem podataka iz destinacije u destinaciju, unutar alata za

kreiranje navigacijskog grafa mogu se definirati argumenti koji se prosljeđuju putem biblioteke naziva Safe Args koja je također dio Android Jetpack biblioteke.



Slika 3.18. Navigacijski graf

Sljedeći korak u navigaciji je izvođenje navigacijske radnje. Na slici 3.19. se može vidjeti kod potreban za izvođenje navigacijske radnje. U ovom slučaju gazi se funkcija definirana u sučelju slušatelja za Post element liste. Nakon što se dodirne jedan od elemenata liste prikazane u elementu korisničkog sučelja *RecyclerView*, izvodi se kod na slici 3.19. Prvo se postavlja radnja koja će se izvesti, radnji koja se skupa s njezinim argumentima prethodno definira u navigacijskom grafu predaju se argumenti koje radnja zahtjeva. Nakon što se postavi radnja, funkcija iz biblioteke Navigation Component naziva *findNavController* brine se za navigacijski kontroler koji izvodi navigaciju.

```
    override fun onItemSelectedListener(id: String) {
        val action =
HomeFragmentDirections.actionHomeFragmentToPostDetailFragment(id)
        findNavController().navigate(action)
    }
```

Slika 3.19. Navigacija između *HomeFragment* fragmenta i *PostDetailFragment* fragmenta

Navigacija u okviru Jetpack Compose u prvoj inačici sadržavala je rute poput onih u razvoju na web platformi. S obzirom da se rute prvenstveno smatraju rješenjem za web platforme, Jetpack Compose dobiva novu verziju navigacije koja je teoretski slična navigaciji u pristupu zasnovanom

na označnom jeziku XML. Ona također sadrži navigacijski graf, iako ne u grafičkom obliku, te navigacijske radnje.

Proces navigacije kreće od kreiranja serijaliziranih destinacija. Serijalizirane destinacije mogu primati argumente različitih tipova. Potrebno je definirati i početnu destinaciju. Na slici 3.20. prikazane su destinacije *Home* i *PostDetail*.

```
@Serializable
object Home

@Serializable
data class PostDetail (val id: String)
```

Slika 3.20. Definicija destinacija *Home* i *PostDetail*

Sljedeći korak je kreacija navigacijskog grafa. Ovaj graf predstavlja isto što i navigacijski graf u pristupu zasnovanom na označnom jeziku XML, međutim nema grafički prikaz nego je definiram Kotlin kodom. Definicija grafa prikazana je slikom 3.21.

```
NavHost(navController, startDestination = HomeScreen) {
     composable<Home> {
         HomeScreen(onNavigateToProfile = { id ->
             navController.navigate(PostDetail(id))
         })
     }
     composable<PostDetail> { backStackEntry ->
         val profile: PostDetail = backStackEntry.toRoute()
         PostDetailScreen(profile.id)
     }
}
```

Slika 3.21. Navigacijski graf u okviru Jetpack Compose

Ono što se posljednje odrađuje je navigacijska radnja nakon pritiska na element liste. Kao što je prikazano na slici 3.22., navigacijska radnja izvodi se putem navigacijskog kontrolera *navController* i metode navigate kojoj se predaje prethodno definirana destinacija i njezin argument.

```
navController.navigate(PostDetail(id = posts[it].postID))
```

Slika 3.22. Definicija destinacija *Home* i *PostDetail*

Rješenje za navigaciju aplikacijom koje se koristi u pristupu zasnovanom na označnom jeziku XML, zbog navigacijskog grafa pruža pregledno i lako kreiranje veza između navigacijskih destinacija te definiranja argumenata potrebnih u navigacijskim destinacijama. Međutim, u slučaju iznimno komplekse aplikacije koja sadrži mnogo destinacija, navigacijski graf može postati

kompleksan i zbog toga težak za održavanje. Navigacijsko rješenje u pristupu zasnovanom na okviru Jetpack Compose pruža sigurnost tipova i programsko definiranje grafa i navigacijskih destinacija u programskom jeziku Kotlin. Zbog toga je kreiranje istih jednostavnije i pripomaže održavanju aplikacije.

Uspoređujući dva pristupa dolazi se do zaključka da okvir Jetpack Compose znatno smanjuje količinu nepotrebnog koda u odnosu na pristup zasnovan na označnom jeziku XML, pogotovo kada se radi o prikazu podataka u listama, gdje se umjesto kreacije adaptera, jednostavno kreira lista unutar stupca. Uz smanjenje koda, pojednostavljeno ponovno korištenje elementa putem funkcija olakšava proširivanje i održavanje. S obzirom da je cijeli kod pisan u jednom jeziku, otklanjanje pogrešaka je također olakšano. Međutim, na primjeru navigacije može se zaključiti da okvir još uvijek evoluira. Pristup zasnovan na označnom jeziku XML pruža stabilnost koja se razvila kroz godine aktivnog korištenja od strane programera, što se manifestira u podršci zajednice u rješavanju problema.

4. ZAKLJUČAK

Dva su načina za izgradnju korisničkog sučelja na Android platformi, pristup zasnovan na modernom okviru Jetpack Compose i pristup zasnovan na označnom jeziku XML. Svaki od navedenih pristupa imaju svoje prednosti i nedostatke. Pristup zasnovan na označnom jeziku XML je tradicionalni pristup koji je već dobro poznat programerima. Pristup zasnovan na modernom okviru Jetpack Compose donosi novu modernu deklarativnu paradigmu koju koriste razni okviri za izgradnju korisničkog sučelja na različitim platformama, čime programerima olakšava izgradnju kompleksnih korisničkih sučelja.

Iako je glavna prednost pristupa zasnovanog na označnom jeziku XML njegova stabilnost i široka podrška zajednice koja je riješila već mnoge postojeće probleme. Moderni okvir Jetpack Compose ipak donosi brojne prednosti kao što su brža izgradnja korisničkog sučelja, lakše održavanje i dinamičko ažuriranje korisničkog sučelja. Važno je napomenuti da iako okvir Jetpack Compose ima brojne prednosti, učenje okvira zahtijeva veliki napor, pogotovo ako osoba koja uči okvir nema iskustva s deklarativnom paradigmom već dolazi iz svijeta pristupa zasnovanog na označnom jeziku XML. Iako se primarno koristi kao rješenje za izgradnju korisničkog sučelja na Android platformi, okvir Jetpack Compose pruža i višeplatformsku podršku kroz Compose Multiplatform inačicu okvira.

U konačnici, izbor između ova dva pristupa ovisi o potrebama projekta. Projekti kojima je potrebna stabilnost i kompatibilnost sa starijim verzijama Android platforme, gradit će svoje korisničko sučelje koristeći pristup zasnovan na označnom jeziku XML, dok će projekti koji trebaju brži rezultat, povećanu produktivnost i smanjen trošak održavanja graditi svoje korisničko sučelje koristeći okvir Jetpack Compose.

LITERATURA

[1] Vetter IT solutions Schweiz GmbH, How to decouple your business logic from UI components? , Basel, Švicarska, veljača 2021., dostupno na: https://www.codequality.rocks/post/how-to-decouple-your-business-logic-from-ui-components [17.5.2022.]

[2] Indeed Editorial Team, What is a User Interface?, Indeed Ireland Operations Limited, srpanj, 2022., dostupno na: https://www.indeed.com/career-advice/career-development/user-interface [20.5.2022.]

[3] UC Berkeley Extension, What Does a Front End Web Developer Do?, Berkely, SAD, travanj 2023., dostupno na: https://bootcamp.berkeley.edu/resources/coding/learn-web-development/what-does-a-front-end-web-developer-do/ [20.5.2022.]

[4] C. Lindley, Web Technologies Employed by Front-End Developers, MJG International LLC, Kopenhagen, Danska, 2019., dostupno na: https://frontendmasters.com/guides/front-end-handbook/2018/practice/tech-employed-by-fd.html [20.5.2022.]

[5] T. Reenskaug, MVC, University of Oslo, Oslo, Švedska, kolovoz 2003., dostupno na: https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html [10.6.2024.]

[6] P. Efthymiou, „Clean Mobile Architecture“, Petros Efthymiou, str. 195-197, travanj 2022.

[7] Mozilla, MVC, Mozilla, prosinac 2023., dostupno na: https://developer.mozilla.org/en-US/docs/Glossary/MVC [9.6.2024.]

[8] A. Dumbravan, „Clean Anrdoid Architecture“, Packt Publishing, str. 252, 2022.

[9] R. F. Garcia, iOS Architecture Patterns, Apress Media LLC, str. 107-111, veljača 2023.

[10] B. Panchal, Frontend Statistics, Radixweb, travanj 2024., dostupno na: https://radixweb.com/blog/frontend-statistics [10.6.2024.]

[11] Meta Open Source, React, 2024., dostupno na: https://react.dev/learn [10.6.2024.]

[12] Google, What is Angular, 2024., dostupno na: https://angular.io/guide/what-is-angular [17.6.2022]

[13] Vue.js, What is Vue, 2024., dostupno na: https://vuejs.org/guide/introduction.html#what-is-vue [17.6.2022.]

[14] Google Developers Training team, Build a simple user interface, Google, travanj 2024., dostupno na: https://developer.android.com/training/basics/firstapp/building-ui [18.6.2022.]

[15] Google Developers Training team, Build a simple app with text composables, Google, travanj 2024., dostupno na: https://developer.android.com/codelabs/basic-android-kotlin-compose-text-composables#3

[10.6.2024.]

[16] N. Smith, „iOS 16 App Development Essentials“, Payload Media, str. 15-16, veljača 2023.

[17] Techtopia, Understanding iPhone Views, Windows and the View Hierarchy, Payload Media Inc., 2023., dostupno na: https://www.techotopia.com/index.php/Understanding_iPhone_Views,_Windows_and_the_View_Hierarchy [10.6.2024.]

[18] Google, Layouts, Google, svibanj 2024., dostupno na: https://developer.android.com/guide/topics/ui/declaring-layout [9.6.2024.]

[19] T. Balint, D. Buketa, „Jetpack Compose by Tutorials: Building Beautiful UI With Jetpack Compose“, Razeware LLC, 2021.

[20] Google, Thinking in Compose, Google, svibanj 2024., dostupno na: https://developer.android.com/jetpack/compose/mental-model [10.6.2024.]

[21] D. You, M. Hu, A Comparative Study of Cross-platform Mobile Application Development, CITRENZ 2021, u Wellington, Novi Zeland, 2021.

[22] Google, Getting started with Android Jetpack, Google, svibanj 2024., dostupno na: https://developer.android.com/jetpack/getting-started [10.6.2024.]

[23] Google, Navigation with Compose, Google, svibanj 2024., dostupno na: https://developer.android.com/jetpack/compose/navigation [17.6.2024.]

[24] R. Costa, Compose Destinations, Github, svibanj 2024., dostupno na: https://github.com/raamcosta/compose-destinations [11.6.2024.]

[25] Geekforgeeks, MVVM Architecture Pattern in Android, Sanchhaya Education Private Limited, listopad 2022., dostupno na: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/ [10.6.2024.]

[26] Visual Paradigm, How to model MVC Framework with UML Sequence Diagram, Visual Paradigm, 2022., dostupno na: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/how-to-model-mvc-with-uml-sequence-diagram/ [7.5.2024.]

[27] Geekforgeeks, MVP Architecture Pattern, Sanchhaya Education Private Limited, listopad 2020., dostupno na: https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example/ [7.5.2024.]

[28] J. Smith, „Advanced MVVM“, Lulu.com, str. 9-10, veljača 2010.

SAŽETAK

Usporedbom pristupa izgradnje korisničkog sučelja na Android platformi analiziraju se pristup zasnovan na označnom jeziku XML te pristup zasnovan na okviru Jetpack Compose. Prikazom rješenja na ostalim platformama povlači se paralela s Android platformom te se analiziraju načini na koje se ostvaruje odvajanje ovisnosti. Razvojem dvaju aplikacija koristeći pristup zasnovan na označnom jeziku XML i pristup zasnovanom na okviru Jetpack Compose, na primjerima analiziraju se razlike između pristupa te navode prednosti i nedostatci pristupa. Usporedba pristupa vrši se s gledišta razvoja, korištenja i održavanja. Razvojem aplikacija dolazi se do saznanja kako pristup zasnovan na okviru Jetpack Compose uveliko smanjuje količinu koda i vremena potrebnog za implementaciju korisničkog sučelja, zbog čega se olakšava i održavanje. Zbog široke podrške zajednice i dugog godina korištenja, pristup zasnovan na označnom jeziku pruža stabilno rješenje pogodno za starije uređaje, čije učenje zahtjeva manji napor od pristupa zasnovanog na okviru Jetpack Compose.

Ključne riječi: Android, korisničko sučelje, odvajanje ovisnosti, okvir Jetpack Compose, označni jezik XML

ABSTRACT

An approach comparison to user interface creation on the Android platform

By comparing the approaches to user interface creation on the Android platform, the XML-based approach and the Jetpack Compose framework-based approach are analyzed. The discussion extends to other platforms that use graphical user interface, drawing parallels with the Android platform and analyzing methods for acheiving separation of concerns. Using examples, the differences between an XML markup language approach and a Jetpack Compose framework approach are analyzed, highlighting their respective strengths and weaknesses. The comparison is evaluated from the perspectives of development, usage and maintenance. The study reveals that the Jetpack Compose framework significantly reduces code and implementation time, thereby easing maintenance. In contrast, the markup language approach offers a stable solution suitable for older devices, supported by wide community and requiring less learning effort compared to Jetpack Compose framework approach.

Keywords: Android, user interface, separation of concerns, Jetpack Compose framework, XML markup language

ŽIVOTOPIS

Zvonimir Mesinger rođen je 8.7.1999. godine u Našicama, s prebivalištem u Starom Petrovom Polju. Nakon završene srednje škole u Orahovici, smjer opća gimnazija, 2018. godine uspješno se upisuje na Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku na Stručni prijediplomski studij Računarstva.

Trenutno je zaposlen kao Mobile Engineer u tvrtki „Eviden Global Delivery Center“. Zainteresiran je za moderne mobilne tehnologije te se usavršava u području razvoja mobilnih aplikacija za različite platforme.

PRILOZI

1. „Usporedba pristupa za izgradnju korisničkog sučelja na Android platformi“ u .docx formatu
2. „Usporedba pristupa za izgradnju korisničkog sučelja na Android platformi“ u .pdf formatu
3. Izvorni kod programskog rješenja