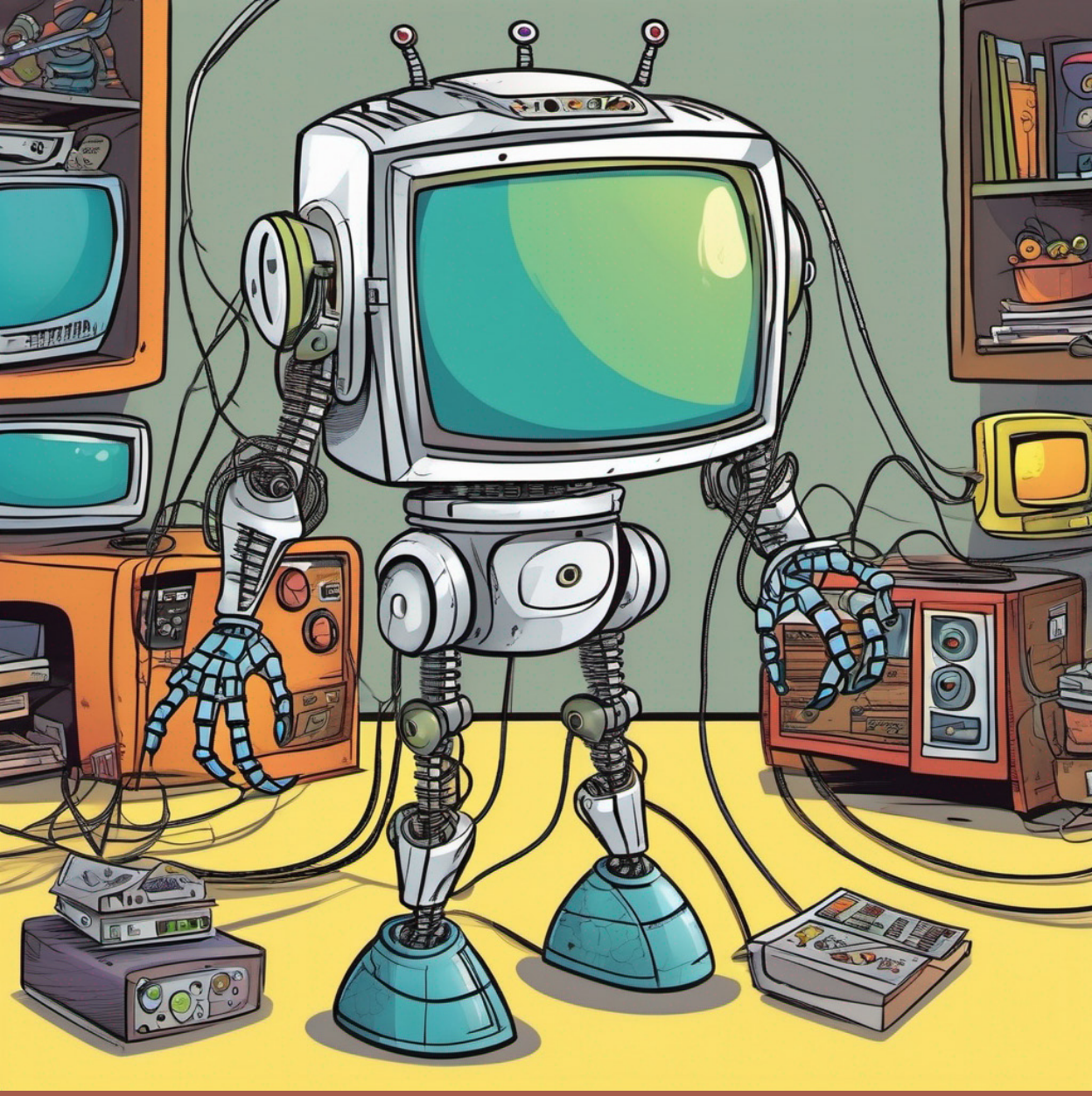


Vensada Okanović, Irfan Prazina
Šeila Bećirović Ramić, Jasmina Čeligija

RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

Vensada Okanović, Irfan Prazina, Šeila Bećirović Ramić, Jasmina Čeligija

RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

Vensada Okanović, Irfan Prazina, Šeila Bećirović Ramić, Jasmina Čeligija
RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

Izdavač
TDP Sarajevo

Za izdavača
Narcis Pozderac

Recenzenti
Prof. dr Selma Rizvić
Prof. dr Belma Ramić-Brkić

Dizajn korica
Irfan Prazina

Tehničko uređenje i računarska obrada
TDP, Sarajevo

1 elektronsko izdanje

Sarajevo, 2023.

ISBN 978-9958-553-67-7
CIP zapis dostupan u COBISS sistemu Nacionalne i univerzitetske biblioteke
BiH pod ID brojem 56974598

Vensada Okanović, Irfan Prazina
Šeila Bećirović Ramić, Jasmina Čeligija

RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

Sarajevo, 2023.

SADRŽAJ

1.
UVOD

Popularnost mobilnih aplikacija različitog sadržaja i namjene iz dana u dan se neprestano povećava. Mobilne tehnologije se trenutno razvijaju brže nego ikad. Sve veći broj ljudi koristi mobilne uređaje u svrhu komuniciranja i obavljanja svakodnevnih poslova. U ovoj knjizi će biti opisan razvoj Android mobilnih aplikacija koristeći programski jezik – Kotlin.

Android je operativni sistem zasnovan na Linux jezgru, razvijen od strane kompanije Google primarno za mobilne uređaje. Ranije je za razvoj Android mobilnih aplikacija najčešće bio korišten Java programski jezik. 2011. godine se pojavljuje novi programski jezik – Kotlin, koji donosi mnoštvo različitih funkcionalnosti. Kotlin je koncizan, ekspresivan jezik, koji je svoju popularnost najbrže stekao na polju razvoja Android mobilnih aplikacija. Vrlo je važna njegova kompatibilnost sa Java programskim jezikom koja omogućava nasljeđivanje i nadgrađivanje već napisanog kôda. Također, Kotlin je, uz Java jezik, proglašen službenim razvojnim jezikom za Android platformu što mu je dodatno osiguralo sigurnu budućost.

Pored izbora jezika, koji će se koristiti u razvoju Android mobilnih aplikacija, važno je i obratiti pažnju na najbolje prakse i biblioteke koje će doprinijeti razvoju kvalitetnih i *user-friendly* aplikacija koje je lako održavati, tesitrati i nadgrađivati. Kao rezultat dugogodišnjeg rada i iskustva developera, Google je 2017. godine predstavio Android *Jetpack* komponente koje su kreirane koristeći najbolje prakse s ciljem da maksimalno iskoriste potencijal Kotlin programskog jezika i da ujedno riješe najčešće probleme koji se pojavljuju u razvoju Android mobilnih aplikacija, kao što su rad sa bazom, izvršavanje asinhronih operacija i slično. Neke od tih komponenata će biti detaljno opisane u ovoj knjizi.

Važno je uzeti u obzir da se razvoj mobilne aplikacije ne svede samo na to da ona radi ono što se od nje očekuje, već je bitno osigurati da se na jednostavan način mogu izmijeniti već postojeće i dodati nove funkcionalnosti. Kao rezultat takvog pristupa, vremenom su nastale različite arhitekture koje imaju isti cilj, a to je razdvajanje nadležnosti komponenata koje čine aplikaciju. Svaka komponenta bi trebala biti zadužena za obavljanje jednog zadatka, pri čemu s drugim komponenatama treba na odgovarajući način ostvariti komunikaciju i razmjenjivati podatke. Stoga, izbor odgovarajuće arhitekture je jedan od bitnih koraka u razvoju mobilnih aplikacija i zbog toga će jedan dio ove knjige biti posvećen upravo ovoj temi.

Cilj ove knjige je da pomogne studentima usvajanje osnovnih koncepata vezanih za razvoj Android mobilnih aplikacija koristeći Kotlin programski jezik. Čitalac bi trebao stvoriti jasnu sliku o tome šta je Kotlin, kako ga koristiti, koje su njegove prednosti i zašto je stekao tako veliku popularnost u kratkom vremenskom periodu. Nakon toga je bitno shvatiti važnost korištenja arhitekture u procesu razvoja mobilne aplikacije kao i primjenu dobrih praksi koje su rezultat dugogodišnjeg rada i iskustva developera širom svijeta.

U ovoj knjizi je korišteno nekoliko stilova kako bi se napravila jasna razlika između različitih vrsta informacija koje se pojavljuju. Dijelovi kôda, koji se navode unutar paragrafa, nazivi klasa kao i ključne riječi Kotlin programskog jezika su pisani koristeći `Consolas` font sa veličinom teksta 10pt kako bi se razlikovali od ostatka teksta. Nazivi svih varijabli, kao i tehnički termini, za koje nije pronađen adekvatan prijevod u bosanskom jeziku, su pisani u *Italic* stilu. Pojedini dijelovi teksta ili pojmovi koji se prvi put spominju su boldirani kako bi se istakla njihova važnost.

2.
KOTLIN

Kotlin je moderni programski jezik kreiran od strane softverske kompanije JetBrains [1] čiji je razvoj počeo 2011. godine, a u upotrebu je pušten pet godina kasnije. Nakon što ga je Google proglasio vodećim jezikom za razvoj Android mobilnih aplikacija ubrzo je stekao pažnju velikog broja korisnika širom svijeta. Kotlin važi kao vrlo moćan programski jezik koji u svakom smislu može parirati Swiftu koji se koristi za razvoj iOS mobilnih aplikacija. Kotlin je općenamjenski, *open-source*, statički tipiziran jezik koji predstavlja savršenu kombinaciju koncepata funkcionalnog programiranja sa objektno – orijentisanim principima.

Mnogo je razloga zašto je Kotlin dobar izbor kada je u pitanju razvoj Android mobilnih aplikacija, a u nastavku će biti navedeni neki od njih [1]:

1. **Kotlin je statički i jako tipiziran jezik** – Tip varijable je poznat već u toku kompajliranja programa, a ne tek u vrijeme njegovog izvršavanja. Prilikom deklaracije varijabli nije potrebno eksplicitno navoditi tip podataka, jer Kotlin ima mogućnost automatskog određivanja tipa varijable na osnovu izraza koji se nalazi sa desne strane znaka jednakosti (eng. *type inference*). Programski jezik sa "jaki tipom" ja onaj u kojem je svaka vrsta podataka unaprijed definisana kao dio programskog jezika i sve konstante i varijable koje se deklarišu u programu moraju biti definisane nekim od tih tipova.
2. **Kotlin je i funkcionalni i objektno–orijentisani jezik** – Pored svih objektno-orijentisanih principa, u Kotlinu funkcije imaju izuzetno važnu ulogu. Funkcije se mogu koristiti na svim mjestima umjesto varijabli i predstavljaju jedan od osnovnih gradivnih blokova aplikacije. Mogu se prosljeđivati kao parametri drugim funkcijama. Takođe, funkcija može vratiti drugu funkciju kao rezultat. U Kotlinu se pojavljuju pojmovi ekstenzijske funkcije i funkcije konteksta čija upotreba je veoma široka.

Upotrebom ekstenzijskih funkcija je moguće već postojeću klasu dodatno proširiti bez promjene izvornog kôda.

3. **Interoperabilnost sa Java jezikom** – Kotlin može koristiti biblioteke koje su pisane u Java programskom jeziku. Na ovaj način se olakšava developerima prelazak na korištenje Kotlina umjesto Java jezika. Mobilna aplikacija može sadržavati objekte pisane u oba jezika.
4. **U Kotlinu se često može izbjeći pisanje *boilerplate* kôda** – Dok u Java jeziku proces kreiranja klase zahtijeva i pisanje svih metoda za dobavljanje i postavljanje vrijednosti atributa, to je u Kotlinu izbjegnuto i njihovo postojanje se jednostavno podrazumijeva.
5. **Kotin je siguran** – Prilikom deklaracije varijabli podrazumijeva se da one nikad ne mogu sadržavati null vrijednost, osim ako se to eksplicitno ne navede. Na taj način je rukovanje izuzecima mnogo jednostavnije i predvidljivije.

2.1. Osnovne karakteristike

Prilikom učenja bilo kojeg novog programskog jezika uvijek je najlakše početi sa upoznavanjem osnovnih elemenata koji ga sačinjavaju. Kotlin programski jezik je vrlo sličan Java jeziku, ali kako kažu njegovi kreatori, pokupio je samo najbolje osobine nekih već zrelih jezika što ga čini elegantnim, modernim i pragmatičnim miksom poznatih modela [2]. U ovom poglavlju će biti objašnjeni osnovni pojmovi koji se susreću u svim programskim jezicima i čije je razumijevanje od velikog značaja za početak učenja. Najprije će biti govora o literalima, varijablama i osnovnim tipovima podataka, a zatim o izrazima i iskazima. Bit će objašnjeno i koji se operatori koriste za formulaciju izraza i iskaza, te će ukratko biti spomenuto koje su to ključne riječi u Kotlinu koje nije moguće ili ne bi trebalo koristiti prilikom imenovanja varijabli, klasa ili funkcija.

2.1.1. Varijable

Riječ varijabla je nastala od engleske riječi *variable* što znači "promjenljiva vrijednost"[1]. Varijable (promjenljive) su memorijske lokacije u programu čiji sadržaj u toku izvršavanja programa može biti promijenjen. One predstavljaju neizostavni dio svakog programskog jezika, stoga ih je vrlo važno razumjeti na samom početku.

[1] https://dictionary.cambridge.org/dictionary/english/variable

U Kotlin programskom jeziku postoje dva načina deklaracije varijabli. Prvi način jeste korištenjem ključne riječi var koja označava varijablu koja u budućnosti može poprimiti neku drugu vrijednost (eng. *mutable variable*). Drugi način jeste korištenjem ključne riječi `val` koja služi za deklaraciju varijabli koje mogu biti inicijalizirane samo jednom unutar nekog bloka izvršavanja (eng. *immutable variable*). U Kotlinu svaka varijabla prije korištenja mora biti prethodno deklarisana, u suprotnom će doći do sintaksne greške. Na ovaj način je zaštićeno da ne dođe do upotrebe pogrešno imenovane varijable. Varijable se deklarišu na sljedeći način:

```
var day: String
```

Ključna riječ `var` označava da se radi o varijabli, zatim slijedi njeno ime (identifikator), separator dvotačka, te specifikacija tipa. Lokalne varijable se često i inicijaliziraju odmah nakon deklaracije:

```
var day: String = “Friday”
```

Moguće je primijetiti da se, za razliku od Java programskog jezika, u Kotlinu tip varijable piše sa desne strane identifikatora. To se radi zato što je Kotlin dovoljno napredan jezik da može sam odrediti tip varijable na osnovu vrijednosti kojom je ona inicijalizirana, tako da je moguće izostaviti eksplicitno specificiranje tipa:

```
var day = “Friday” // Kompajler zna da je izraz sa desne strane
tipa String, tako da i varijablu day tretira kao String
```

Kao što je već rečeno, postoji još jedan način deklaracije varijabli, a to je korištenjem ključne riječi `val`. Ako poredimo s Java programskim jezikom, riječ `val` u Kotlinu ima slične karakteristike kao i riječ `final` u Java jeziku. Ovako označene varijable mogu biti inicijalizirane samo jednom i njihova vrijednost se kasnije ne može promijeniti. Za razliku od konstanti, o kojima će biti govora kasnije, varijable označene sa ključnom riječju `val` ne moraju biti inicijalizirane odmah prilikom njihove deklaracije. To je moguće uraditi i naknadno, ali bitno je da kada se jednom postavi njena vrijednost ne može se više mijenjati.

```
val day: String = “Friday”  // Deklaracija i inicijalizacija u
istoj liniji
val day: String
day = “Friday” // Inicijalizacija prethodno deklarisane
varijable day
```

Nemogućnost promjene stanja objekta nakon prve inicijalizacije je važan koncept u modernim programskim jezicima. Kreatori Kotlina preporučuju korištenje varijabli označenih ključnom riječju `val` kad god je to moguće, zato što na taj način tok programa postaje predvidljiviji i otporniji na greške. Kada bilo koji dio kôda može promijeniti stanje objekta, vrlo lako može doći do pojave nepredviđenih grešaka koje će utjecati i na ostale dijelove aplikacije u kojima se taj objekat koristi, tako da i sam proces otklanjanja grešaka postaje komplikovaniji.

Prilikom imenovanja varijabli treba uzeti u obzir da njihovi nazivi ne mogu počinjati brojem ili nekim simbolom (npr. '#', '$', '?' i dr.), kao i da razmak u nazivu ili korištenje ključnih riječi kao identifikatora nisu dozvoljeni.

U Kotlin programskom jeziku se varijable označene ključnom riječju `val` razlikuju od konstanti po tome što je konstante potrebno odmah inicijalizirati prilikom njihove deklaracije. Ako se neka vrijednost neće mijenjati i ako spada u neki od primitivnih tipova (npr. Int, Long, Boolean i sl.) ili je tipa String, a njena vrijednost je poznata u toku kompajliranja programa, moguće ju je označiti da bude konstanta na sljedeći način:

```
const val pi = 3.14
```

Vrijednost varijable *pi* više nije moguće mijenjati, jer će u suprotnom doći do kompajlerske greške.

2.1.2. Tipovi podataka

U Kotlin programskom jeziku svaka varijabla se smatra objektom nad kojim možemo pozvati bilo koju funkciju članicu ili tzv. "property" odnosno atribut. Iako interno može doći do pretvaranja nekih objekata u njihove primitivne tipove (npr. brojevi, karakteri i sl.), korisniku to nije vidljivo i on sve vrijeme ima osjećaj kao da radi sa objektima [3]. U nastavku će biti opisani osnovni tipovi podataka koji se koriste u Kotlinu, a to su: brojevi, stringovi, karakteri, boolean vrijednosti, te nizovi.

2.1.2.1. Brojevi

Predstavljanje brojeva u Kotlinu je vrlo slično kao i u Java jeziku, s tim da Kotlin ne dozvoljava interno pretvaranje iz jednog tipa u drugi.

Ukoliko se eksplicitno ne navede tip varijable i ukoliko njena vrijednost ne prelazi minimalnu ili maksimalnu vrijednost koju mogu imati varijable

tipa `Int`, onda će Kotlin smatrati da je tipa `Int`. Ako vrijednost prelazi opseg koji mogu imati varijable tipa `Int`, onda spada u kategoriju varijabli tipa `Long` [3].

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

Ukoliko je potrebno raditi sa decimalnim brojevima, dostupna su dva tipa podataka: `Float` i `Double`, koji se razlikuju po broju bita iza zareza koje mogu čuvati. Decimalne vrijednosti će se automatski posmatrati kao da su tipa `Double` ukoliko se ne definiše suprotno. Ako je potrebno da varijabla bude tipa `Float`, to je moguće specificirati na dva načina. Prvi način je da se iza dodijeljene vrijednosti piše slovo "F" ili "f" što označava da će varijabla biti tipa `Float` [3].

```
val pi: Float = 3.14
val pi = 3.14F
```

2.1.2.2. Boolean

Bulov tip podataka (eng. *Boolean*) se u Kotlin programskom jeziku predstavlja koristeći literale `true` i `false`. U konstrukcijama koje očekuju da im se preda vrijednost tipa `Boolean`, nije moguće poslati brojeve 1 ili 0 i očekivati da će se izvršiti njihova konverzija u `true`, odnosno `false`. Iz toga zaključujemo da sljedeći dio kôda neće raditi:

```
val isEnabled = 1
if (isEnabled)
   doSomething()
```

Unutar `if` uslova se očekuju samo vrijednosti `true`, `false`, ili izraz koji će proizvesti jednu od prve dvije vrijednosti. Sve druge opcije neće raditi ono što se očekuje.

2.1.2.3. Nizovi

Nizovi (eng. *arrays*) spadaju u skupinu fundamentalnih struktura podataka u gotovo svim programskim jezicima. Ideja nizova jeste pohranjivanje više elemenata istog tipa.

Deklaracija i inicijalizacija niza

U Kotlinu, nizove nije moguće kreirati koristeći par uglastih zagrada kao u mnogim drugim programskim jezicima, ali zato postoje dvije posebne dvije funkcije koje se koriste u tu svrhu, a to su: `arrayOf()` i `arrayOfNulls()`.

Funkcija `arrayOf()` očekuje da joj proslijedimo elemente niza odvojene zarezom.

```
val myArray = arrayOf(1, 2, 3) // Kreira niz elemenata [1, 2, 3]
čiji je tip automatski određen.
```

Moguće je i prije navođenja elemenata niza specificirati o kojem tipu podataka se radi kako se ne bi moralo vršiti automatsko određivanje.

```
val myArray = arrayOf<Int>(1, 2, 3) // Kreira niz elemenata tipa
Int [1, 2, 3]
```

Alternativno, funkcija `arrayOfNulls()` će kreirati niz veličine koju proslijedimo kao parametar funkcije, a svi elementi niza će imati početnu vrijednost `null`, koju je kasnije moguće promijeniti.

```
val myArrayOfNulls = arrayOfNulls(3) // Kreira niz elemenata
[null, null, null]
```

Pristup elementima niza

Kao što je prethodno rečeno, u Kotlinu se vrlo često koriste uglaste zagrade za pristup elementima niza. Uglaste zagrade predstavljaju alternativu funkcijama `get()` i `set()`, iako je i ove dvije funkcije moguće koristiti.

Primjer korištenja uglastih zagrada:

```
myArray[0] = 7 // Postavlja vrijednost prvog elementa u nizu na 7
println(myArray[0]) // Ispisuje se broj 7 na ekranu
```

Primjer korištenja funkcija `get()` i `set()`:

```
myArray.set(0, 7) // Prvi parametar je indeks elementa kojeg
želimo promijeniti, a drugi nova vrijednost koju mu dodjeljujemo
println(myArray.get(0)) // Ispisuje se broj 7 na ekranu
```

2.1.2.4. Stringovi

String predstavlja jednodimenzionalni niz karaktera koji se nalaze između dvostrukih navodnika. Osnovna razlika između primitivnog tipa `string`,

koji se koristi u mnogim modernim jezicima, i tipa `String`, koji se koristi u Kotlin programskom jeziku, je u tome što su vrijednosti tipa `String` objekti, nad kojima je moguće pozivati veliki broj već implementiranih funkcija što uveliko olakšava rad s njima.

Deklaracija i inicijalizacija varijabli tipa String

Varijabla tipa `String` u Kotlinu se definiše na neki od sljedećih načina:

```
val message = "Hello world" // implicitno – bez navođenja o
kojem tipu podataka se radi
```

```
val message: String = "Hello world!" // eksplicitno –
specificiranjem koji tip vrijednosti se očekuje
```

```
val empty = String() // Kreira se prazan string
```

Pristup elementima stringa

S obzirom da je string niz karaktera, moguće je koristiti uglaste zagrade ili funkciju `get()` (kojoj je potrebno proslijediti indeks traženog elementa) za pristup njegovim elementima. Međutim, ono što nije moguće je promjena vrijednosti pojedinačnih elemenata, odnosno karaktera koristeći uglaste zagrade. Stringovi su u Kotlin programskom jeziku *immutable* strukture podataka, što znači da dio kôda prikazan u nastavku neće raditi:

```
var message = "Hello world!"
```

```
message[0] = 'h' // Greška
```

Iako nije moguće promijeniti pojedinačni karakter unutar stringa, varijablama koje su kreirane koristeći ključnu riječ `var` je moguće ponovo dodijeliti neki potpuno novi literal tipa `String`.

```
var message = "Hello world"
println(message) // Ispisuje se poruka Hello word u konzoli
message = "New message" // Dodjela novog literala tipa String
varijabli message
println(message) // Ispisuje se poruka New message u konzoli
```

String templates

String template je najjednostavnije definisati kao string literal koji sadrži ugrađene izraze [1]. Takav izraz (eng. *template expression*) počinje simbolim $ iza kojeg se piše ime varijable čija vrijednost se dodaje umjesto simbola $.

```
val x =  10
println(“x = $x”) // U konzoli će se ispisati: x = 10
```

Simbol “$” je desno asocijativan što znači da će evaluirati prvi izraz koji se nalazi s njegove desne strane. Stoga, ukoliko je u string potrebno umetnuti izraz koji se sastoji od više operanada, onda se takav izraz treba ograničiti parom vitičastih zagrada.

```
val str = “Hello”
println(“Length is: ${str.lenght}) // U konzoli će se ispisati:
Length is: 5
```

Raw string

Raw String je tip stringa koji je specifičan za Kotlin. Kreira se koristeći par trostrukih navodnika unutar kojih se može pisati tekst u više linija bez potrebe za prethodnim spajanjem svake od njih i bez potrebe za korištenjem *escape* karaktera [3]. Odličan je izbor za pisanje dužih rečenica ili paragrafa.

```
val rawString = ””” Raw String je tip stringa koji je specifičan
za Kotlin.
Kreira se koristeći par trostrukih navodnika unutar kojih se
može pisati tekst u više linija bez potrebe za prethodnim
spajanjem svake od njih. ”””
```

```
println(rawString) // Naredba za ispis stringa na ekran
```

2.2. Kontrola toka programa

Pored `if` i `when` izraza, u nastavku će biti govora o tome kako koristiti `while` i `for` petlje koje predstavljaju neizostavni dio modernih jezika. Sposobnost programa da obavlja složene zadatke zasniva se na svega nekoliko načina kombinovanja jednostavnih naredbi u upravljačke strukture.

2.2.1. IF uslov

Najčešći način za implementaciju grananja u programima jeste korištenje `if`/`else` uslova. Uslovne naredbe obavljaju različite akcije u ovisnosti od rezultata evaluacije uslova. S obzirom na to da je u Kotlinu skoro sve izraz, tako i `if` uslovi predstavljaju izraze, jer vraćaju neku vrijednost [4].

Sada će najprije biti prikazan tradicionalni način korištenja `if` uslova, a zatim ćemo vidjeti kako je taj proces u Kotlinu znatno olakšan i kako se često većina naredbi može napisati u samo jednoj liniji kôda.

```
// Tradicionalno korištenje if uslova
var minValue: Int
if (a < b) {
    minValue = a
}
else {
    minValue = b
}

// Korištenje if uslova u Kotlinu
val minValue = if (a < b) a else b
```

`If` uslovi mogu biti i blokovi, a posljednji izraz koji se nalazi unutar bloka će biti dodijeljen varijabli [4].

```
val minValue = if (a < b) {
    println("a is less than b")
    a
}
else {
    println("b is less than a")
    b
}
```

Ukoliko postoji više uslova koje je potrebno ispitati onda iza prvog `if` uslova pišemo jednu ili više `if else` naredbi. `If` blok može, ali ne mora završavati s `else`.

```
var nextValue: Int
if (a < b) {
    nextValue = a
}
else if (a > b) {
    nextValue = b
}
else if (a == b) {
    nextValue = a + b
}
```

Važno je napomenuti da ako je rezultat `if` uslova potrebno dodijeliti nekoj varijabli, onda taj uslov mora imati i `else` granu. `If` uslovi mogu biti korišteni kao i iskazi, kao što je slučaj u Java programskom jeziku. To znači da nije uvijek potrebno dodijeliti rezultat njihovog izvršavanja nekoj varijabli, već se vrijednost koju vrate može jednostavno zanemariti.

2.2.2. WHEN uslov

`When` uslov je moguće posmatrati kao `switch/case` iskaz koji se koristi u Java programskom jeziku, ali sa snažnijim karakteristikama. `When` uslov je, također, izraz koji će pokušati sekvencijalno uskladiti svoj argument sa svim mogućim granama dok ne pronađe onu koja mu odgovara [4].

```
val number = 12
when (number) {
   5 ->  println(“number is equal to 5)
   10 -> {
         println(“number is equal to 10)
         println(“10 is a composite number)
   }
   else -> {
         println(“Unknown number)
   }
```

Rezultat ispisa u konzoli:

```
Unknown number
```

Može se primijetiti da poslije znaka “`->`” unutar `when` bloka nije potrebno stavljati vitičaste zagrade ako slijedi samo jedna linija kôda, ali ako slijedi blok naredbi onda ga je potrebno ograničiti zagradama.

Kao što je već rečeno, `when` uslov predstavlja izraz u Kotlin programskom jeziku, pa je rezultat njegovog izvršavanja moguće dodijeliti nekoj varijabli.

```
val result = when (number) {
        3, 5, 7 -> “prime number”
        10, 12 -> “composite number”
        else -> “error”
}
```

Ukoliko se vrijednost `when` uslova dodijeljuje nekoj varijabli onda je potrebno implementirati i `else` blok, jer će se desiti da se provjere sve opcije, pa ukoliko ni jedna opcija ne zadovoljava uslov, onda će se doći do `else` bloka.

Jedan od razloga zašto koristiti `when` uslov je taj što će se se automatski pokušati izvršiti konverzija argumenta u neku od opcija unutar `when` uslova i ako ni u jednom slučaju ne uspije, doći će se do `else` bloka ukoliko je implementiran [4].

```
when (selectedClass) {
     is animalClass -> println(“Animal”)
     is numberClass -> println(“Number”)
     else -> println(“Cannot cast selectedClass”)
```

Koristeći `when` uslov je na jednostavan način moguće ispitati da li se neki broj nalazi u traženom opsegu ili unutar neke kolekcije na sljedeći način [1]:

```
val const = when (x) {
in 1...10 -> “cheep”
in 10..100 -> “regular”
in 100..1000 -> “expensive”
in specialValues -> “special value”
else -> “not rated”
```

Kod `when` uslova argument uopšte ne mora postojati. Moguće je uraditi bilo kakve vrste provjera za koje bi inače trebalo više `if/else` uslova.

```
val result = when {
   x in 1..100 -> “cheep”
   num is Int -> “$num is Integer”
}
```

2.2.3. FOR i WHILE petlje

`For` petlja u Kotlinprogramskom jeziku nije klasična petlja na koju su mnogi navikli, ali je vrlo slična onoj koju koriste ljubitelji Pythona. `For` petlju je moguće koristiti kod svih struktura koje su iterabilne.

Lista je primjer iterabilne strukture, te je moguće korisiti `for` petlju za obilazak njenih elemenata.

```
val cities = listOf(“Istanbul”, “London”, “Paris”)
for (city in cities) {
      println(city)
}
```

`For` petlja uvijek implicitno kreira novu *read-only* varijablu (u ovom primjeru *city*). Ako već negdje u programu postoji varijabla sa istim imenom, postat će zasjenjena od strane varijable koja se koristi u `for` petlji.

Kada su u pitanju `for` petlje, unutar njih je moguće koristiti nekoliko različitih operatora. Na primjer ispis brojeva od 0 do 100 je moguće postići koristeći "`..`" operator [4].

```
for (x in 0..100) println(x) // Ispišu se brojevi od 0 do 100 u
konzoli (uključivo)
```

Ako želimo isključiti posljednju vrijednost u petlji poželjno je koristiti ključnu riječ `until`

```
for (x in 0 until 100) println(x) // Ispiše brojeve od 0 do 99 u
konzoli
```

Moguće je kontrolisati korak za koji će se povećati vrijednost brojača na način da se iza ključne riječi `until` doda ključna riječ `step` te njegova vrijednost.

```
for (x in 0 until 100 step 2) println(x) // Ispiše se svaki
drugi broj
```

Što se tiče `while` petlje, ona je slična `while` petlji iz Pythona, s tim da se unutar `while` uslova mora nalaziti Boolean vrijednost ili izraz koji je proizvodi. Nije moguće koristiti broj 1 u zamjenu za `true` ili 0 umjesto `false`.

```
var x = 0
while (x < 100) {
   println(x)
   x++
}
```

2.3. Upravljanje izuzecima

Svaki program će raditi stabilno samo ako su u njegovom izvornom kôdu otklonjene greške i ne postoje uslovi koji mogu dovesti do pojave nepredviđenih situacija. Da bi se osigurao rad aplikacije i u slučaju pojave nepredviđenih grešaka, programer mora unaprijed razmišljati koje su to greške do kojih bi moglo doći i da li se efekat greške može ograničiti. Neke od situacija koje mogu prouzrokovati nepredviđeno ponašanje aplikacija ili potpuni prestanak njihovog rada su: iznenadno isključivanje pristupa Internetu

kod aplikacija kojima je pristup potreban da pravilno funkcionišu, rad sa bazom, čitanje i pisanje datoteka, rad sa varijablama čija je vrijednost `null` i slično. Ukoliko dođe do bilo koje od ovih situacija, aplikacija treba nesmetano nastaviti sa radom ili eventualno obavijestiti korisnika da je došlo do pojave greške. Kotlin ima jako dobar sistem u kojem je osigurano da će većina grešaka biti uhvaćena već u fazi kompajliranja aplikacije, ali šta u slučajevima kada nije tako? Tada je potrebno ručno napisati `try/catch` blok. `Try/catch` blok se sastoji od tri komponente:

1. `try` – Ključna riječ koja označava početak bloka i znači da dio koja koji slijedi može prouzrokovati izuzetak
2. `catch` – Dio koji služi za obradu izuzetka
3. `finally` – Ovaj dio je opcionalan, a služi za izvršavanje dijela kôda koji je neophodan u svakom slučaju, čak i ako se ne uhvati odgovarajući izuzetak

```
try {
    // dio kôda koji može izazvati izuzetak
}
catch (e: SomeException) {
    // Place to handle exception
}
finally {
    // Opcinalni dio
}
```

Svaki izuzetak (eng. *exception*) u Kotlinu je naslijeđen iz klase `Throwable` i svaki sadrži poruku, te niz koraka koji su ga prouzrokovali. `Catch` dijelova može biti više za obradu različitih vrsta izuzetaka, ali barem jedan je neophodan. Kao i većina stvari u Kotlinu i `try/catch` blok spada u izraze, te rezultat njegovog izvršavanja može biti dodijeljen varijabli [5].

```
val result = try { parseString(str) }
    catch (e: NumberFormatException) { null }
```

Korištenjem ključne riječi `throw` moguće je ručno "baciti" izuzetak, odnosno "Exception object". Izraz `throw` je specijalnog tipa `Nothing`. Ovaj tip podataka je karakterističan za Kotlin i ne pojavljuje se u Java jeziku. Nema nikakvu vrijednost i služi da označi nedostižne dijelove kôda. Funkcija koja vraća `Nothing` u suštini nikad ne vrati vrijednost [5].

```
fun error (message: String) : Nothing {
    throw IllegalArgumentException(message)
}
```

Nakon poziva prethodno deklarisane funkcije, kompajler će znati da ne treba nastavljati sa radom i da je došlo do neke greške.

```
val name = person.name ?: error("Unknown name")
```

2.4. Upravljanje *null* vrijednostima

U procesu razvoja bilo koje vrste aplikacija najviše problema donose `null` vrijednosti, odnosno način na koji se s njima rukuje. Jedan od glavnih ciljeva Kotlin programskog jezika jeste eliminacija rizika od pojave *NullPointerException*-a.

Neke od situacije u kojima može doći do pojave *NullPointerException*-a su sljedeće [6]:

- Eksplicitno bacanje izuzetka (`throw NullPointerException()`)
- Korištenje operatora "!!" (Više o operatorima će biti govora kasnije)
- U konstruktor se proslijedi neinicijalizirana `this` refrenca i ukoliko se negdje pokuša iskorititi doći će do problema

U Kotlinu se pravi jasna razlika između objekata koji mogu biti nulabilni i onih koji to nisu. Po *dafaultu*, kada se kreira varijabla, njoj se ne može dodijeliti `null` vrijednost, jer će odmah doći do pojave kompajlerske greške. Na ovaj način je zagarantovan bezbijedan pristup svim atributima varijable *x*.

```
var x: String = "Hello"
x = null // Greška
```

Ukoliko je iz nekog razloga potrebno da varijabla bude nulabilna, odnosno da može čuvati vrijednost koja je `null`, to se mora eksplicitno navesti koristeći operator "?" koji se piše iza definisanja tipa varijable.

```
var y: String? = "Hello"
y = null // Neće doći do greške
```

Ukoliko pristupimo nekom od atributa objekta *x* garantovano je da neće doći do pojave *NullPointerException*-a. Međutim, to nije slučaj sa objektom *y*. Njegovim atributima moramo pristupati vrlo pažljivo.

```
val a = x.length // Neće doći do greške
val b = y.length // Greška jer y može biti null
```

Naravno, prijavljivanje greške nije rješenje u ovoj situaciji, jer je potrebno dobiti dužinu stringa koji može, a ne mora, biti `null`. Postoji nekoliko načina provjere da li neki objekat sadrži `null` vrijednost ili ne.

Prvi način je eksplicitno poređenje posmatrane varijable sa vrijednošću `null` koristeći operator "==".

```
val result = if (str == null) -1 else str.length
```

Ovaj način provjere je jednostavan i dobar izbor u slučaju da je objekat *immutable*. Međutim, ako je potrebno pristupati atributima objekta koji je *mutable* onda se koriste drugi načini provjere.

2.4.1. *Safe-call* operator

U Kotlinu se pojavljuje novi tip operatora "?.", koji se naziva *safe-call* operator, koji se dobije kombinacijom dva znaka: upitnika i tačke. Korištenje ovog operatora predstavlja elegantan način provjere da li neki objekat sadrži `null` vrijednost ili ne.

```
val x: String? = "Hello"
println (x?.length) // Ispisat će dužinu stringa x samo ako on
nije null, u suprotnom će ovaj dio kôda biti zanemaren i neće
doći do NullPointerException-a
```

Ovaj operator je posebno pogodan za korištenje ukoliko imamo lanac vrijednosti koje je potrebno provjeriti. To se može desiti ako jedna klasa sadrži instancu druge klase koja ima svoje atribute koji su možda nulabilni. Kao primjer možemo posmatrati klasu `ShoppingCart`. Ova klasa može, a ne mora, sadržavati instancu npr. klase `Book`. Klasa `Book` može imati atribut koji je nulabilan, pa je i to potrebno provjeriti prije nego što mu pristupimo. Upotrebom ovog operatora je osigurano da neće doći do pojave *NullPointerException*-a [6].

```
println(shoppingCart?.book?.description)
```

2.4.2. *Elvis* operator

Nekada proces provjere da li neka varijabla sadrži `null` vrijednost ili ne može biti vrlo dosadan. Često je potrebno koristiti varijablu ako i samo ako ona ne sadrži `null` vrijednost, u suprotnom se tok programa nastavlja u drugom pravcu. U većini modernih programskih jezika se ta provjera vrši korištenjem `if` uslova kao u primjeru koji slijedi:

```
val x: String? = getStringValue()
val strLenght: Int = if (x != null)
                        x.length
                     else
                        -1
```

Koristeći *Elvis* operator, koji se pojavljuje u Kotlin programskom jeziku, prethodno napisani kôd je moguće jednostavnije i mnogo preglednije napisati. *Elvis* operator "?:" se koristi za kraće pisanje `if` uslova, a dobije se kombinacijom upitnika i dvotačke.

```
val x: String? = getStringValue()
val strLenght = x?.length ?: -1
```

Ukoliko izraz koji se nalazi sa lijeve strane operatora nije `null`, onda se on vraća kao rezultat, u suprotnom se vrati vrijednost koja se nalazi sa desne strane operatora. U ovom primjeru će se varijabli *strLength* dodijeliti dužina stringa *x* ako i samo ako *x* nije `null`, inače se dodjeljuje -1.

Primijetimo da se sa desne strane *Elvis* operatora može pisati sve što je neki izraz, odnosno što u konačnici postaje neka vrijednost, pa tako da i izrazi `return` i `throw` koji se vrlo često susreću u kombinaciji sa `if` uslovom [6].

```
fun foo (employee: Employee): Int? {
val department = str.getDepartment() ?: return null
val name = department.getName() ?: throw
IllegalArgumentException("Name required")
// ostali kôd
}
```

2.4.3. Operator "!"

Još jedan operator, koji se koristi u procesu rukovanja `null` vrijednostima, jeste "!!" – *not null assertion* operator. Ovaj operator funkcioniše na način da operand koji se nalazi sa njegove lijeve strane pokuša konvertovati u non-null tip. Ukoliko konverzija ne uspije doći će do pojave izuzetka.

Na ovaj način se dopušta da dođe do pojave *NullPointerException*-a, iako se ova vrsta izuzetka u Kotlinu nastoji izbjegavati. Upravo zbog toga je potrebno vrlo oprezno rukovati s ovim operatorom.

```
val x = str!!.length // Ako str nije null onda se pristupa
njegovom property-u length, u suprotnom se pojavljuje izuzetak
```

2.5. Funkcije

U nastavku će biti govora o funkcijama kao jednom od sastavnih dijelova svih programskih jezika. Svi složeni zadaci se obično rješavaju na način da se podijele na niz jednostavnijih, više ili manje nezavisnih zadataka, koje je moguće pojedinačno rješavati. Takva hijerarhijska dekompozicija stvara mogućnost da se neki dijelovi kôda ponovo iskoriste ukoliko bude potrebe za tim. Najjednostavnije rečeno, funkcije su programske cjeline koje nad dobijenim ulaznim podacima izvrše niz naredbi i po potrebi vrate rezultat izvršavanja. Svaka funkcija bi trebala biti osmišljena na način da obavlja jednu dobro definisanu funkcionalnost, te da korisnik ne mora poznavati detalje njene implementacije kako bi je mogao iskorititi. U nastavku će biti opisan način na koji se vrši deklaracija funckija u Kotlin programskom jeziku, zatim će biti govora o podrazumijevanim vrijednostima i imenovanim argumentima.

2.5.1. Deklaracija funkcije

Deklaracija funkcije je moguća na tri mjesta, unutar klase, van klase I unutar druge klase [1]. U Kotlinu se funkcije deklarišu koristeći ključnu riječ `fun` nakon koje slijedi naziv funkcije, odnosno identifikator. Unutar zagrada nalazi se deklaracija formalnih argumenata. Funkcija može, ali i ne mora imati argumente. Također, specificiranje povratnog tipa (ukoliko funkcija nešto vraća) je opcionalno s obzirom na to da Kotlin na osnovu vrijednosti koja se vraća može odrediti kojeg je ona tipa. Na kraju se piše par vitičastih zagrada i sve što se nalazi između njih predstavlja tijelo funkcije. Primjer jednostavne funkcije slijedi u nastavku:

```
fun sayHello(name: String): String {
      return ”Hello $name!” // Ako je ime bilo Bob, bit će
      vraćen string “Hello Bob”
}
```

Ono što Kotlin čini posebno zanimljivim jeste da prethodno napisana funkcija može biti svedena na svega jednu liniju kôda. S obzirom na to da se u njenom tijelu nalazi samo jedan izraz čiji je rezultat potrebno vratiti, moguće je koristiti znak "=" nakon čega slijedi taj izraz. Tip rezultata se automatski određuje. Ovakve funkcije se nazivaju "expression functions" [7].

```
fun sayHello(name: String) = "Hello $name!"
```

Što se tiče imenovanja funkcija, bitno je napomenuti da naziv ne smije počinjati brojem, niti smije sadržavati specijalne karaktere. Dobra praksa je koristiti standardne konvencije o imenovanju koje se koriste i u većini modernih programskih jezika kako bi kôd bio čitljiviji i kako ne bi dolazilo do zabuna. Dobro poznati *Camel case* stil je odličan izbor u ovoj situaciji.

Već je spomenuto da funkcije ne moraju uvijek vraćati neku vrijednost. U Java jeziku se takve funkcije označavaju koristeći ključnu riječ `void`. Kotlin ne zahtijeva da se eksplicitno navede da funkcija neće imati povratnu vrijednost, već će se to zaključiti iz konteksta. Funkcija `printNumbersBetween()` ispisuje brojeve između *firstNumber* i *lastNumber* i ne vraća nikakvu vrijednost [1].

```
fun printNumbersBetween(firstNumber: Int, lastNumber: Int) {
      for (x in firstNumber..lastNumber)
            println(x + " ")
}
```

2.5.2. Imenovani parametri funkcije

Za razliku od Java jezika, u Kotlinu se parametar može imenovati na mjestu poziva. Na taj način se jasno vidi kojem parametru se dodjeljuje koja vrijednost. Važno je napomenuti da ukoliko specificiramo naziv jednog parametra, onda je to potrebno uraditi i za svaki sljedeći kako ne bi došlo do zabune. Kompajler uopšte neće dopustiti neimenovanje ostalih parametara ako je prvi imenovan, što znači da sljedeći dio kôda neće raditi [1]:

```
setUserData(username = "user", email = "e@gmail.com", "123")
// I posljednji parametar je potrebno imenovati kao i prethodne!
```

2.5.3. Varijabilni broj argumenata funkcije

Funkcije u Kotlinu mogu imati varijablini broj argumenata. Sintaksa se malo razlikuje od Java programskog jezika, ali je suština ista. Za

deklaraciju funkcija koje imaju varijabilni broj argumenata koristi se ključna riječ `vararg` iza koje slijedi identifikator, te tip argumenta.

```
fun sumOfNumbers(vararg numbers: Int) : Int {
  var sum: Int = 0
  for (number in numbers)
       sum += number
  return sum
}
```

Primjer poziva prethodno implementirane funkcije `sumOfNumbers`:

```
println(sumOfNumbers(1, 2)) // Rezultat ispisa: 3
println(sumOfNumbers(1, 2, 3)) // Rezultat ispisa: 6
println(sumOfNumbers(1, 2, 5, 10)) // Rezultat ispisa: 18
```

2.5.4. Lambda funkcije

Lambda funkcije predstavljaju anonimne funkcije, tj. "funkcijske literale" koji se ne deklarišu, već se kao izrazi prosljeđuju običnu funkcijama višeg reda [8]. Kao i klasične funkcije, lambda funkcije mogu primati parametre i vraćati neku vrijednost. Za njihovo kreiranje je potrebno pratiti sljedeću sintaksu:

```
val rezultat: Tip = {lista_argumenata -> tijelo_funkcije}
```

Od svega navedenog, jedino tijelo funkcije ne može biti izostavljeno, ostalo je sve opcionalno. Lambda funkcije mogu, a ne moraju, imati argumente, a tip povratne vrijednosti se često može automatski odrediti na osnovu posljednjeg izraza koji se nalazi u njenom tijelu. Također, lambda funkcija može biti direktno proslijeđena nekoj funkciji višeg reda, što znači da može, a ne mora, biti dodijeljena varijabli. Bitno je napomenuti da lambda funkcija mora biti ograničena vitičastim zagradama. Na početku se nalazi lista parametara koje očekuje, a zatim se piše znak "`->`" koji označava mjesto od kojeg počinje tijelo funkcije [1].

Da bi prethodno napisani tekst bio jasniji, slijedi jednostavan primjer kreiranja lambda funkcije koja ne očekuje nikakve parametre i ne vraća nikakvu vrijednost, a ima zadatak da ispiše poruku u konzoli.

```
val sayHello: () -> Unit = {
      println("Hello")
}
```

Prazna zagrada označava da funkcija ne očekuje parametre, zatim slijedi znak "→", te tip povratne vrijednosti, koji je u ovom slučaju `Unit`, jer funkcija ne vraća ništa. `Unit` se u Kotlinu može posmatrati kao `void` u Java jeziku.

Ako izostavimo opcionalne anotacije dobit ćemo sljedeće:

```
val sayHello = {
      println("Hello")
}
```

Ovako napisana funkcija se može odmah proslijediti nekoj funkciji višeg reda ili ju je moguće pozivati na jedan od dva načina:

```
// Prvi način:
sayHello() // Rezultat ispisa: Hello
// Drugi način:
sayHello().invoke() // Rezultat ispisa: Hello
```

U nastavku će biti prikazano kako bi izgledala lambda funkcija koja prima dva argumenta tipa `String`, a kao rezultat vraća `Boolean` u zavisnosti od toga jesu li njihove dužine jednake ili ne.

```
val result: (String, String) -> Boolean = {a: String, b: String ->
       a.length == b.length
}

// Ili ako zanemarimo opcionalne anotacije dobijemo
val result = {a: String, b: String ->
      a.length == b.length
}
```

Iako se lambda funkcije mogu pozivati kao i obične funkcije, ipak nije zamišljeno da se tako koriste. Njihova svrha je da eliminišu bespotrebno kreiranje imenovanih funkcija koje bi se iskoristile samo na jednom mjestu. Funkcija višeg reda koja kao argument prima prethodno kreiranu lambda funkciju bi mogla izgledati ovako [1]:

```
fun invokeLambda(lambda: (String, String) -> Boolean): Boolean {
      val a = "Hello"
      val b = "Hello"
      return lambda(a, b)
}
```

2.5.5. Ekstenzijske funkcije

Svim ljubiteljima Java programskog jezika je dobro poznato da dodavanje novih funkcionalnosti nekoj klasi podrazumijeva vlasništvo nad tom klasom ili kreiranje potpuno nove klase koja nasljeđuje već postojeću. To ponekad izgleda kao dodatni posao koji oduzima dosta vremena. Kao rješenje ovog problema, Kotlin uvodi pojam "ekstenzijske funkcije" (eng. *extension functions*) koje predstavljaju ulazak u svijet nepoznat u Java jeziku. Ove funkcije su poznate u mnogim modernim programskim jezicima, ali su Java programeri iz nekog razloga ostali uskraćeni. Ekstenzijske funkcije su funkcije koje se mogu pozivati nad instancom klase kao da su funkcije članice, ali su deklarisane i implementirane izvan klase i imaju pristup ključnoj riječi `this` [1].

One omogućavaju dodavanje novih funkcionalnosti već postojećim klasama bez potrebe za nasljeđivanjem istith.

U procesu razvoja mobilnih aplikacija developeri se susreću sa potrebom kreiranja nekih pomoćnih funkcija koje smiještaju u folder koji se obično zove *Utils*, kako bi dijelovi kôda koji se često ponavljaju bili na jednom mjestu. Možda je jedan od najjednostavnih, a najčešće korištenih poziva prikazivanje "Toast-a" na ekranu. *Toast* predstavlja poruku koja se obično pojavi kao *feedback* nakon neke obavljene operacije. S obzirom na frekvenciju njegovog pojavljivanja u mobilnim aplikacijama, najčešće se kreira funkcija koja će primiti potrebne parametre za njegov prikaz i koja će se moći koristiti u svim dijelovima programa. Primjer takve funkcije slijedi u nastavku:

```
fun showMessage(context: Context, message: String) {
    Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
}
```

Ovako implementiranu funkciju koristimo na način da prvo pišemo ime foldera u kojem se nalazi, zatim naziv funkcije sa parametrima koje očekuje.

```
Utils.showMessage(context, "Operation completed!")
```

Koriteći ekstenzijske funkcije koje nudi Kotlin programski jezik, prethodno napisana linija kôda bi mogla izgledati ovako:

```
context.showMessage("Operation completed!")
```

Slijedi pojašnjenje kako je to moguće i kako se deklarišu ekstenzijske funkcije.

Kako bi dodali ekstenzijsku funkciju, potrebno je koristiti ključnu riječ fun, zatim navesti klasu nad kojom kreiramo funkciju, te specificirati njen naziv i eventualno parametre koje očekuje. Funkcija `showMessage()` iz prethodnog primjera bi se mogla implementirati kao ekstenzijska funkcija na sljedeći način:

```
fun Context.showMessage(message: String) {
      Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
}
```

Ekstenzijske funkcije ne mijenjaju implementaciju postojećih klasa nad kojima se kreiraju, već proširuju njihove funkcionalnosti. Obično se postavljaju u gornji sloj aplikacije direktno ispod paketa kako bi bile dostupne svim klasama. Da bi njihovo korištenje bilo omogućeno, u *build.gradle* datoteku je potrebno dodati: `apply plugin: 'kotlin-android-extensions'`.

U razvoju mobilnih aplikacija je vrlo često potrebno prikazati ili sakriti neki element u ovisnosti od situacije. Kreiranje ekstenzijske funkcije nad klasom `View` nam omogućava da to učinimo na jednostavan i elegantan način.

```
fun View.isVisible(visible: Boolean = true) {
      visibility = if (visible) VISIBILE else GONE
}
```

Ovako kreirana funkcija se može pozivati nad svim elementima klase `View`.

```
nameTextView.isVisible(false) // nameTextView će biti skriven
```

Radi boljeg razumijevanja ekstenzijskih funkcija, bit će prikazana jednostavna implementacija još dvije funkcije koje se koriste za validaciju email-a i password-a, a kreiraju se nad klasom `String`.

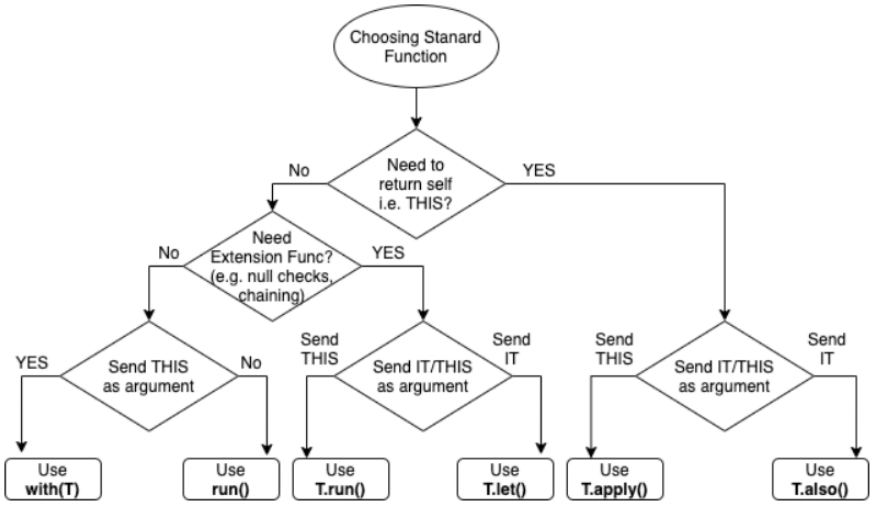
```
fun String.isValidEmail(): Boolean =
      Patterns.EMAIL_ADDRESS.matcher(this).matches()
fun String.isValidPassword(): Boolean = this.length >= 8
```

Ovako kreirane funkcije se pozivaju nad objektom tip `String` kojeg je potrebno validirati.

```
val email = "email@gmail.com"
val password = "12345678"
if (email.isValidEmail() && password.isValidPassword())
      doSomething()
```

2.5.6. Funkcije konteksta

Funkcije konteksta predstavljaju skupinu funkcija čija je svrha izvršavanje bloka kôda unutar konteksta nekog objekta. U standardnoj biblioteci Kotlina postoji pet tipova ovakvih funkcija (`apply`, `also`, `let`, `with`, `run`) koje se razlikuju prema načinu na koji se odnose na objekat konteksta i vrijednosti koju vraćaju kao rezultat. Nekada je u radu s podacima potrebno vratiti sam objekat nad kojim se operacije obavljaju, a nekada je potrebno vratiti potpuno novi objekat koji će predstavljati rezultat obavljenih akcija. To su neki od parametara koji pomažu pri odluci koju od navedenih funkcija koristiti [9]. Na slici 1. se jasnije može vidjeti o čemu treba voditi računa prilikom izbora funkcije konteksta.



Slika 1. Parametri koje treba uzeti u obzir prilikom odabira odgovarajuće funkcije konteksta2

Prije nego što sve funkcije budu detaljno objašnjene, osvrnut ćemo se ukratko na razliku između ključnih riječi `this` i `it`, te na vrste vrijednosti koje mogu biti vraćene iz funkcija konteksta.

[2] Slika preuzeta sa: https://medium.com/mobile-app-development-publication/mastering-kotlin-standard-functions-run-with-let-also-and-apply-9cd334b0ef84

2.5.6.1. THIS vs IT

Unutar tijela funkcija konteksta, objektu nad kojim se pozivaju se može pristupati na dva načina. Oba načina podrazumijevaju korištenje nekih skraćenih referenci umjesto njegovog stvarnog imena. Prvi način podrazumijeva tretiranje objekta kao prijemnika (`this`), a drugi kao argument (`it`) lambda funkcije [9].

Funkcije konteksta `run`, `with` i `apply` pristupaju objektu koristeći ključnu riječ `this`. Dakle, u njihovim lambda funkcijama atributi objekta su dostupni kao u običajenim funkcijama članicama, tj. metodama klase. U većini slučajeva ključna riječ `this` može biti i izostavljena, jer će se podrazumijevati. Navedene tri funkcije je pogodno koristiti u situacijama kada se pristupa i manipuliše atributima objekta, a ne čitavim objektom.

S druge strane, funkcije `let` i `also` objekat konteksta tretiraju kao argument lambda funkcije. Pristupanje objektu, koristeći ključnu riječ `it`, znači da atributi objekta nisu dostupni u toj mjeri kao kad im se pristupa koristeći ključnu riječ `this`. Ključna riječ `it` ne može biti zanemarena prilikom pristupa atributima. Pristup objektu koristeći ključnu riječ `it` je pogodan u situacijama kada se unutar lambda funkcije nalazi više različitih varijabli i kada je operacije potrebno vršiti nad čitavim objektom.

2.5.6.2. Funkcija `apply`

Funkcija `apply` je jedna od dvije funkcije koja vraća objekat nad kojim se poziva. Ona predstavlja i ekstenzijsku funkciju što znači da se poziva nad objektom, a unutar nje se objektu može pristupati iz konteksta `this`. Funkcija `apply` je pogodna za korištenje u situacijama kada je potrebno obaviti neke operacije nad atributima objekta [9]. Kao primjer objekta nad kojem se vrše operacije bit će korišten tekstualni element koji se često susreće u razvoju mobilnih aplikacija. U nastavku slijedi prikazan standardni način manipulacije atributima objekta

```
textView.text = “New text”
textView.color = RED
textView.visibility = View.VISIBLE
```

te primjenom `apply` funckije nad objektom.

```
textView.apply {
    this.text = “New text”
    color = RED
```

```
    visibility = View.VISIBLE
}
```

Prilikom pristupanja atributima objekta moguće je koristiti ključnu riječ `this` (npr. `this.text`), ali ako se i ne navede eksplicitno neće biti problema, jer će se jednostavno podrazumijevati. Također, bitno je napomenuti kako se u Kotlinu ne zahtijeva pozivanje određenih metoda za promjenu stanja atributa objekta, već je to moguće direktno uraditi, jer svi atributi imaju ugrađene funkcije za postavljanje i dohvatanje njihovih vrijednosti.

Iako se na prvi pogled čini da korištenje funkciju `apply` doprinosi jedino povećanju broja linija kôda, ipak se njena svrha jasnije vidi u kompleksnijim situacijama kada je potrebno lančano pozvati i niz drugih funkcija nad objektom od interesa, u kojima je vraćen kao rezultat.

2.5.6.3. Funkcija `also`

Funckija `also` je vrlo slična funkciji `apply`. Koristi se u situacijama kada želimo pristupiti već postojećem objektu ili napraviti neke promjene nad njim i na kraju kao rezultat vratiti taj objekat. Za razliku od funkcije `apply`, u ovom slučaju objektu se pristupa koristeći ključnu riječ `it`. Ova funkcija može biti korisna ukoliko je unutar konteksta funkcije potrebno pristupiti čitavom objektu, a ne samo njegovim atributima, što ne bismo mogli koristeći funkcijeu `apply`. Nad objektom, koji se vraća kao rezultat obavljenih operacija, moguće je lančano pozivati i niz drugih funkcija [9].

```
val cities = mutableListOf("Paris", "London", "Istanbul")
cities.also {
    println("The list contains $it.length cities.")
}
.add("Sarajevo")
```

2.5.6.4. Funkcija `let`

Funkcija `let` predstavlja ekstenzijsku funkciju konteksta čija povratna vrijednost odgovara rezultatu lambda izraza. Unutar konteksta ove funkcije, objektu se pristupa koristeći ključnu riječ `it`. Primjer tipičnog scenarija u kojem bi se mogla iskoristiti ova funkcija je provjera da li određena varijabla sadrži `null` vrijednost ili ne. U primjeru koji slijedi može se primijetiti i korištenje *safe-call* operatora "?." u kombinaciji sa `let` funkcijom.

```
val str: String? = “Hello world!”
val length = str?.let {

    //Unutar tijela ove lambda funkcije it sigurno nema null
    vrijednost, tako da je bez problema moguće pristupiti
    njegovim atributima

    println(“Length of the string is $it.length)
}
```

Još jedna situacija u kojoj bi se mogla iskoristiti funkcija `let` je u slučaju da želimo neku vrijednost privremeno imenovati kako bi kôd bio čitljiviji. U primjeru koji slijedi se može primijetiti da se iz liste *numbers* uzima prvi element i u tijelu lambda funkcije mu se daje naziv *firstItem*. Nakon izvršenih modifikacija nad njim, kao rezultat se vraća objekat tipa `String` nad kojim se onda primjenjuje funkcija `toUpperCase()` koja kapitalizira svako slovo unutar stringa [9].

```
val numbers = listOf(“one”, “two”, “three”, “four”)
val modifiedFirstItem = numbers.first().let { firstItem ->
    println(“The first item of the list is ‘$firstItem’”)
    if (firstItem.length >= 5) firstItem else “!” + firstItem +
    “!”
}.toUpperCase()
println(“First item after modifications: ‘$modifiedFirstItem’”)
```

2.5.6.5. Funkcija `with`

Funkcija `with` predstavlja funkciju konteksta koja kao povratnu vrijednost ima rezultat lambda izraza. Za razliku od funkcije `let`, funkcija `with` nije ekstenzijska funkcija, što znači da se ne poziva nad objektom nego se objekat prosljeđuje kao argument. Unutar tijela ove funkcije, objektu se pristupa koristeći ključnu riječ `this`. Ovu funckiju je moguće čitati kao: “Sa ovim objektom učini sljedeće” [9].

```
val numbers = mutableListOf(“one”, “two”, “three”)
with(numbers) {
    println(“’with’ is called with argument $this”)
    println(“It contains $size elements”)
}
```

Funkcija `with` se može koristiti i u svrhu kreiranja pomoćnog objekta čiji se atributi koriste za obavljanje nekih operacija i koji se vraća kao rezultat iz funkcije [9].

```
val numbers = mutableListOf("one", "two", "three")
val firstAndLast = with(numbers) {
    "The first element is ${first()}," +
    " the last element is ${last()}"
}
println(firstAndLast)
```

2.5.6.6. Funkcija `run`

Funkcija `run` za razliku od funkcije `with` spada u skupinu ekstenzijskih funkcija koje se pozivaju nad objektom od interesa. Funkcija `run` objektu pristupa koristeći ključnu riječ `this`. Korisna je u situacijama kada je potrebno neke atribute objekta inicijalizirati početnim vrijednostima, a ujedno i obaviti određene operacije, te na kraju vratiti lambda izraz kao rezultat [9].

```
val service = MultiportService("https://example.kotlinlang.org", 80)
val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

2.6. Klase

Objektno orijentisano programiranje ima za cilj približavanje modela programa načinu ljudskog razmišljanja. U osnovi objektno orijentisanog programiranja glavnu ulogu imaju objekti – instance klase koje su pohranjene na jednistvenoj adresi u memoriji, čuvaju određene podatke, sadrže metode za manipulaciju tim podacima, te imaju mogućost interakcije s drugim objektima. Za klase kažemo da su koncepti objekta, dok su objekti instance koje utjelovljuju te koncepte. U ovom poglavlju će biti objašnjeno kako deklarisati klase, zatim će biti govora o primarnom i sekundarnim konstruktorima klase, te načinima njihove implementacije u Kotlin programskom jeziku. Nakon toga slijedi poglavlje o nasljeđivanju kao vrlo važnom konceptu u objektno orijentisanom programiranju. Na kraju je opisana uloga apstraktnih klasa, te *companion* objekata koji su karakteristični za Kotlin.

2.6.1. Deklaracija klase

U programskom jeziku Kotlin, za deklaraciju klasa dovoljno je koristiti ključnu riječ `class` nakon koje slijedi naziv klase.

```
class Employee
```

Ovako kreirana klasa se instancira na sljedeći način:

```
val employee = Employe()
```

Zaglavlje i tijelo klase su opcionalni, ali se uglavnom navode. Zaglavlje dolazi nakon imena klase i ograničeno je parom običnih zagrada unutar kojih se može nalaziti lista potrebnih parametara. Sljedeći korak jeste kreiranje tijela klase koje se nalazi između vitičastih zagrada, a unutar kojeg se navode atributi i kreiraju funkcije članice, odnosno metode.

```
class Employee(val name: String) {
    // tijelo klase
}
```

Instanciranje prethodno deklarisane klase se radi na sljedeći način:

```
val employee = Employee("Ellie")

println("Employee name: " + employee.name)
// Rezultat ispisa: Employee name: Ellie
```

2.6.2. Atributi klase

Atribut, odnosno svojstvo klase (eng. *property*) se odnosi na javnu varijablu koja je definisana unutar klase zajedno sa metodama za dobavljanje i postavljanje njene vrijednosti. Potpuna sintaksa deklaracije atributa podrazumijeva pisanje ključne riječi `val` ili `var` nakon koje slijedi naziv atributa, zatim tip, te početna vrijednost. Za sve atribute se podrazumijeva da su javni, osim ako se ne navede drugačije.

```
var <propertyName>: <PropertyType> = <property_initializer>
    [<getter>]
    [<setter>]
```

Za razliku od većine programskih jezika, u Kotlinu nije potrebno posebno deklarisati i implementirati tzv. *gettere* i *settere*, odnosno funkcije za dobavljanje i postavljanje vrijednosti varijable. To je ono što je interno implementirano i o čemu ne treba posebno voditi računa osim u situaciji da je

potrebno nešto promijeniti u samoj strukturi njihove implementacije. Poljima klase se jednostavno pristupa na način da se poslije kreiranog objekta piše tačka, te naziv polja kojem želimo pristupiti.

```
class Person() {
    var name: String = "noname"
}

val person = Person()

person.name = "Ellie" // Poziva se setter
println(person.name) // Poziva se getter (Rezultat ispisa:
Ellie)
```

Dio kôda koji slijedi u nastavku ima identičnu svrhu kao prethodno napisani kôd, ali je prikazano kako ručno implementirati *gettere* i *settere*.

```
class Person {
    var name: String = "noname"
    get() = field  // getter
    set(value) { field = value } // setter
}

val person = Person()
person.name = "Ellie" // Pristupa se setteru
println(person.name) // Pristupa se getteru (Rezultat ispisa:
Ellie)
```

Činjenica da su atributi unutar klase javni narušava koncept enkapsulacije. Ukoliko to nekima predstavlja problem, uvijek je moguće atribut označiti da bude privatan, ali se onda neće automatski kreirati *getteri* i *setteri*, nego je to potrebno uraditi ručno.

Tipično, varijablama, koje imaju osobinu da nisu nulabilne, se u trenutku kreiranja klase mora dodijeliti neka početna vrijednost. Međutim, to nije uvijek moguće. Nekada ta vrijednost nije poznata odmah u procesu deklaracije klase. Kao rješenje ovog problema, u Kotlinu se može iskoristiti kasna inicijalizacija. Pisanjem ključne riječi `lateinit` (eng. *late initialization*) ispred deklaracije varijable osigurava se da će toj varijabli biti dodijeljena neka vrijednost, samo ne odmah, već u trenutku kada ta vrijednost bude poznata. Na ovaj način je kompajler siguran da varijabla neće ostati neinicijalizirana.

2.6.3. Primarni i sekundarni konstruktor

S Kotlin programskim jezikom dolaze minorne promjene kada je u pitanju rad s klasama u odnosu na ostale moderne jezike, ali ono o čemu će biti govora u ovom dijelu jeste mogućnost klase da ima više od jednog konstruktora. Podsjetimo se da konstruktori u suštini predstavljaju skupinu akcija koje se automatski izvrše nad objektom onog trenutka kada se on instancira. Kreiranje objekta zahtijeva inicijalizaciju njegovih atributa kako bi mu postali upotrebljivi, te da ne bi vraćali neočekivane vrijednosti. Konstruktor je ujedno i funkcija članica, odnosno metoda klase koja može očekivati parametre i koja nema povratnu vrijednost. U Kotlinu se javljaju pojmovi primarni i sekundarni konstruktor.

Svaka klasa može imati samo jedan primarni konstruktor koji ne može sadržavati nikakvu dodatnu logiku kao što bi bio ispis teksta na ekran ili bilo šta slično. Njegov zadatak je da primi vrijednosti kojima će biti inicijalizirani atributi klase. Atributi mogu biti *read-only* ili *read-write* u zavisnosti da li se prilikom deklaracije koristi ključna riječ `val` ili `var`. Primarni konstruktor se definiše tako što se ispred zaglavalja klase doda ključna riječ `constructor`, a zatim se unutar običnih zagrada navodi lista atributa klase koje je potrebno inicijalizirati. Ukoliko primarni konstruktor ne sadrži posebne anotacije, moguće je izbaciti korištenje ključne riječi `constructor`, jer će se ona podrazumijevati [10].

```
class Employee constructor(val name: String, var age: Int) {}

//ili

class Employee(val name: String, var age: Int) {}
```

Na ovaj način će atributi klase biti deklarisani i inicijalizirani onog trenutka kada se izvrši kreiranje objekta, odnosno instance klase. Ako je potrebno uključiti neku dodatnu logiku prilikom inicijalizacije atributa, onda se to odvija unutar `init` bloka ili direktno u tijelu funkcije. `Init` blok je prvo što se poziva nakon kreiranja objekta određene klase. U tom slučaju primljeni parametri će samo poslužiti za inicijalizaciju atributa, te se ispred njih ne pišu ključne riječi `val` ili `var`. Kada sve navedeno uzmemo u obzir dobit ćemo modifikovanu klasu koja izgleda ovako:

```
class Employee(_name: String, _age: Int) {

    val name: String
```

```
    val age: Int = _age

    init {
        name = _name.capitalize()
        println("Name: $name")
        println("Age: $age")
    }
}
```

Ovako kreiranu klasu ćemo instancirati na sljedeći način:

```
val employee = Employee("Ellie", 27)
```

Odmah nakon instanciranja u konzoli će se ispisati:

```
Name: Ellie
Age: 27
```

U Kotlinu, konstruktori mogu imati i tijelo, baš kao što je slučaj u Java programskom jeziku, ali kada se kreiraju na taj način onda se zovu sekundarnim konstruktorima i tada ih može biti više unutar jedne klase. Sekundarni konstruktori se ne upotrebljavaju često, ali ih je, radi kompletnosti rada, potrebno spomenuti. Njihova uloga se najbolje vidi u situacijama kada je potrebno prilikom inicijalizacije atributa uključiti i neku dodatnu logiku. Svaki sekundarni kontruktor mora direktno ili indirektno pozivati primarni konstruktor, u suprotnom će doći do kompajlerske greške.

Ono što je važno napomenuti jeste da parametri sekundarnog konstruktora nisu ujedno i atributi klase (kao što je slučaj s primarnim konstruktorm), već isključivo služe za njihovu inicijalizaciju. Sekundarni konstruktor se kreira jednostavno pisanjem ključne riječi `constructor` unutar tijela klase nakon čega slijedi opcionalna lista parametara. U primjeru koji slijedi vidimo da sekundarni konstruktor poziva primarni konstruktor koristeći ključnu riječ `this`, te mu prosljeđuje parametar *name*, a nakon toga se atributu *id* dodjeljuje vrijednost *id* koji se prima kao parametar u konstruktoru. Ukoliko klasa ne sadrži primarni konstruktor delegiranje parametra se vrši indirektno.

```
class Employee (val name: String) {
    var id: Int = -1
    constructor(name: String, id: Int) : this(name) {
        this.id = id
        println("Employee name: $name")
```

```
        println("Employee id: $id)
    }
}
```

Instanciranje objekta je isto kao i do sada

```
val employee = Employee("Ellie", 1)
```

Odmah nakon instanciranja će se u konzoli ispisati:

```
Name: Ellie
Age: 27
```

Ako klasa ne sadrži ni primarni ni sekundarni konstruktor onda će se automatski kreirati podrazumijevani konstruktor s javnim modifikatorom vidljivosti. Ukoliko se želi izbjeći da klasa uopšte ima bilo kakav konstruktor to se radi dodavanjem privatnog modifikatora vidljivosti ispred ključne riječi `constructor`.

```
class Employee private constructor () {}
```

2.6.4. Nasljeđivanje

Jedan od vrlo važnih koncepata u objektno orijentisanom programiranju jeste nasljeđivanje. Nasljeđivanjem klasa se stvara mogućnost za ponovno korištenje kôda, odnosno ukida se potreba za višestrukim pisanjem istih funkcionalnosti na više mjesta. Ideja nasljeđivanja jeste da se odrede slične klase i da se pri nasljeđivanju promijene samo neka specifična svojstva, dok se ostala svojstva nasljeđuju u nepromijenjenom obliku. To znači da je kod kreiranja klase, umjesto pisanja potpuno novih atributa i funkcija, moguće definisati novu klasu koja nasljeđuje dio ili sve varijable i metode već postojeće klase. Postojeća klasa se tada naziva baznom, a novokreirana izvedenom.

Međutim, za razliku od Java programskog jezika gdje se podrazumijeva da novokreirana klasa može naslijediti dio ili potpunu strukturu bilo koje već postojeće klase, u Kotlinu su sve klase označene kao "final", što znači da po *defaultu* ne mogu biti nadklase, tj. da ih nije moguće naslijediti. To znači da dio kôda, koji slijedi u nastavku, neće raditi.

```
class Person {
}
```

```
class Employee: Person () {
}
```

Da bi prethodno napisani kôd radio ono što se od njega očekuje, potrebno je klasu `Person` označiti da bude `open`, što znači da je jedna od njenih novih uloga i to da može biti naslijeđena, odnosno da može biti nadklasa nekoj drugoj klasi. Također, sve funkcije članice i atributi, koje je potrebno naslijediti unutar neke izvedene klase, moraju biti označeni koristeći ključnu riječ `open`, a onda ih je u tijelu izvedene klase potrebno označiti koristeći ključnu riječ `override` da bi kompajler znao da se radi o naslijeđenom atributu, odnosno funkciji. Unutar izvedene klase, bazna klasa se referencira koristeći ključnu riječ `super` [10].

```
open class Person {
    open val age: Int = 0
    open fun talk() {
        println(“Hello world”)
    }
}

class Employee: Person () {
    override val age: Int = 30
    override fun talk() {
        super.talk()
    }
}
```

Primijetimo da se proces nasljeđivanja sastoji od pisanja imena novokreirane klase, nakon čega slijedi dvotačka, te ime bazne klase sa parametrima koje očekuje. Ako izvedena klasa ima primarni konstruktor, onda se bazna klasa mora inicijalizirati upravo na mjestu kreiranja i moraju joj se proslijediti parametri koje očekuje (ako takvih parametara ima).

```
open class Person(name: String) {
}

class Employee(name: String): Person(name) {
}
```

U slučaju da izvedena klasa nema primarni konstruktor, onda je u svakom od sekundarnih konstruktora koji se mogu pozivati potrebno inicijalizirati baznu klasu koristeći ključnu riječ `super` sa potrebnim parametrima [10].

```
open class Person(name: String) {
}

class Employee : Person {
   constructor (name: String) : super (name)
}
```

2.6.5. *Companion* objekat

Ukoliko je potrebno neki atribut ili neku funkciju članicu, odnosno metodu vezati za čitavu klasu, a ne za njene pojedine instance, onda se takvi atributi i takve metode smiještaju unutar "*companion* objekta". *Companion* objekat predstavlja *singleton* čijim se atributima i metodama pristupa preko klase u kojoj su definisani. *Companion* objektima nije moguće pristupati preko konkretnih instanci klase. Deklarišu se na način da se prvo piše ključna riječ `companion` object nakon koje slijedi naziv objekta (koji može biti i izostavljen).

```
class Car(val horsepowers: Int) {

    companion object Factory {
        val cars = mutableListOf<Car>()

        fun makeCar(horsepowers: Int): Car {
            val car = Car(horsepowers)
            cars.add(car)
            return car
        }
    }
}
```

2.6.6. Podatkovne klase

U određenim situacijama je potrebno kreirati jednostavne klase koje neće sadržavati nikakve metode, već im je jedina svrha čuvanje podataka. Kreiranje takvih klasa bi u većini programskih jezika podrazumijevalo i pisanje svih funkcija za dobavljanje i postavljanje vrijednosti njihovih polja, što je samo po sebi proces koji bespotrebno oduzima puno vremena. Međutim, koristeći tzv. "podatkovne klase" (eng. *data classes*) koje dolaze sa Kotlin programskim jezikom, moguće je izbjeći ručnu implementaciju svih tih funkcija, jer će one biti automatski generisane. Deklaracija podatkovnih

klasa je vrlo jednostavna. Podrazumijeva pisanje ključne riječi `data` ispred ključne riječi `class`, a zatim slijedi naziv klase i lista atributa.

```
data class Person (val id: Int, val name: String, val age: Int)
```

Instanciranje se vrši na način da se u konstruktor jednostavno proslijede vrijednosti kojim će biti inicijalizirani atributi klase. Atributima se pristupa na isti način kao i kad su u pitanju obične klase.

```
val person = Person (1, "Ellie", 27)
println(person.name) // Rezultat ispisa: Ellie
```

Pored *gettera* i *settera* koji će biti kreirani automatski, kreirat će se i funkcije: `toString()`, `equals()`, `componentN()`, `copy()`, te `hashCode()`.

Posebno su korisne funkcije `copy()` i `equals()`.

Funkcija `copy()` se koristi u situacijama kada je potrebno promijeniti određene atribute unutar objekta, a ostatak ostaviti nepromijenjenim.

```
person.copy(age = 28)
println(person.toString()) // Rezultat ispisa: Person(id=1,
name=Ellie, age=28)
```

U Kotlinu su dva objekta jednaka ako imaju isti tzv. "hash code", koji se dobije koristeći `hashCode()` funkciju. Funkcija `equals()` vraća `true` ili `false` u zavisnosti od toga da li je "hash code" objekata koji se porede isti ili nije.

```
val person1 = Person(1, "Ellie", 27)
val person2 = Person(2, "Allan", 30)

if (person1.equals(person2))
     println("Person 1 and Person 2 are the same.")
else println("Person 1 and Person 2 are different.")
```

Rezultat ispisa:

```
Person 1 and Person 2 are different.
```

U radu sa podatkovnim klasama je važno obratiti pažnju na sljedeće [11]:

- Primarni konstruktor mora imati barem jedan parametar
- Svi parametri unutar primarnog konstruktora moraju biti označeni sa `val` ili `var`

– Podatkovne klase ne mogu biti apstraktne, “open”, “sealed” ili “inner”
– Podatkovne klase mogu implementirati interfejse

2.6.7. *Sealed* klase

Sealed klase u Kotlinu predstavljaju neku vrstu ekstenzija *Enum* klase. Za njih se kaže da su: “Enumi sa super moćima”. *Sealed* klase predstavljaju novi koncept koji dolazi sa Kotlin programskim jezikom i koji otvara mnoge mogućnosti koje Java developeri nemaju. Upotrebom *sealed* klasa se kreira ograničena hijerarhija u kojoj se može izabrati samo jedan tip od više ponuđenih. Za razliku od `enum` klasa kod kojih je ograničenje da sve instance moraju biti istog tipa, unutar `sealed` klase moguće je imati objekte različitih tipova [11].

Deklaracija ovakvih klasa podrazumijeva pisanje ključne riječi `sealed` nakon koje slijedi ključna riječ `class`, zatim identifikator, odnosno naziv klase, te tijelo koje sadrži klase koje predstavljaju moguće opcije.

```
sealed class VideoDownloadState {
   object Loading(): VideoDownloadState()
   data class Error(val errorMessage): VideoDownloadState()
   data class Content(val content: MediaFile):
VideoDownloadState()
}
```

Primjer korištenja `sealed` klase:

```
fun status(state: VideoDownloadState) {
  when (state) {
    is VideoDownloadState.Loading -> println(“Downloading the video.”)
    is VideoDownloadState.Error -> println(“Error has occurred.”)
    is VideoDownloadState.Content -> saveContent(state.content)
}
```

Važno je napomenuti da `sealed` klase spadaju u skupinu apstraktnih klasa, što znači da ne mogu biti pojedinačno instancirane. Također, ne mogu imati javne konstruktore. Za njihove konstruktore se podrazumijeva da su privatni.

2.6.8. Unutrašnje klase

Ugniježdene klase mogu biti korisne u slučaju da određena klasa ima samo pomoćnu ulogu za glavnu klasu i ne koristi se nigdje drugo osim na tom

mjestu. Ugniježđene klase u Kotlinu ne sadrže referencu na vanjsku klasu unutar koje se nalaze, što znači da nemaju pravo pristupa atributima i metodama vanjske klase. Upravo zbog toga, u Kotlinu se uvodi pojam tzv. "unutrašnjih klasa" (eng. *inner classes*), koje sadrže referencu na klasu unutar koje su deklarisane, tako da mogu pristupiti njenim atributima i metodama. Ovakve klase se kreiraju korištenjem ključne riječi inner [12].

```
class Person(val name: String) {
    inner class Gender(val gender: String) {
        fun getName() = name
    }
}
```

Primjer rada s unutrašnjim klasama:

```
val name = Person("Ellie").Gender("Female").getName()
println(name) // Rezultat ispisa: Ellie
```

Inner klase mogu biti i anonimne, a kreiraju se koristeći "*object expression*".

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }
    override fun mouseEntered(e: MouseEvent) { ... }
})
```

3.
ANDROID

Android je operativni sistem (OS) baziran na modificiranoj verziji Linux kernela i drugih *open source* softvera, dizajniran za pametne telefone i tablete. Android se pojavljuje 2008. godine i do danas se razvilo nekoliko verzija Android operativnog sistema, koje su poznate po kôdnim imenima, kao što su: Aestro, Blender, Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwitch, Jelly Bean, KitKat, Lollipop, Marshmallow, Nougat, Oreo, Pie, 10, 11, 12, 12L i 13.

Android aplikacije su se ranije primarno pisale u Java programskom jeziku, a danas se preferira Kotlin programski jezik. Pored njih, koriste se i drugi jezici, kao što su: C#, Python, C/C++, Lua, HTML5 + CSS + JavaScript, i drugi. Za razvoj Android aplikacija se koristi skup alata poznat pod nazivom Android *Software Development Kit* (SDK). Sav softver potreban za programiranje i razvoj Android aplikacija je besplatan i može se preuzeti s Interneta: Download Android Studio and SDK tools[3].

Android aplikacije se izvršavaju na Android OS koji je instaliran na uređaju ili virtuelnoj mašini. U Android OS svaka aplikacija je drugačiji korisnik s jedinstvenim ID brojem. Za svaku aplikaciju se postavljaju permisije, tako da sve datoteke te aplikacije može koristiti korisnik (aplikacija) sa navedenim ID brojem. Za potrebe pristupanja korisničkim datotekama ili sistemskim funkcionalnostima, potrebno je zahtijevati odgovarajuće permisije od korisnika. Više o permisijama se može pronaći na: Permissions overview[4]

Android sadrži automatski *build* sistem Gradle. Sa njim je moguće graditi (*build*), pokretati i testirati aplikacije. Rezultat *build*-anja često je APK

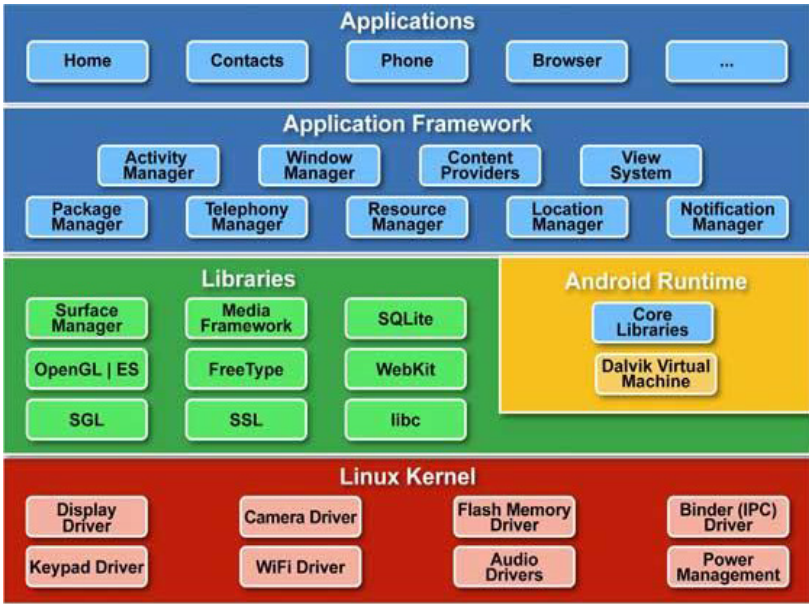
[3] http://developer.android.com/sdk/index.html

[4] https://developer.android.com/guide/topics/permissions/overview

datoteka, koja se može prebaciti na uređaj ili emulator, te pokretati. APK datoteka je softverski paket, arhiva koja sadrži Android program i sve metapodatke potrebne za njegovo pokretanje. Android Studio automatski generiše sve potrebne datoteka i konfiguracije za *build*-anje aplikacije koja se razvija. Datoteke koje su vezane za Gradle, a bitne su za ispravan rad i razvoj aplikacije su: - *build.gradle* - na nivou projekta - postavke vezane za sve module na nivou projekta; - *build.gradle* - na nivou modula, app modul - postavke vezane za dati modul, sadrži podatke o *compileSdkVersion, minSdkVersion i targetSdkVersion*. Postavka *compileSdkVersion* govori o tome koja verzija Android SDK će se koristiti prilikom *build*-anja aplikacije, *minSdkVersion* sadrži podatke o najstarijoj verziji koja je podržana u aplikaciji, *targetSdkVersion* podaci o verziji na kojoj je testirana aplikacija i za koju je aplikacija namijenjena. - *settings.gradle*- definiše listu svih modula i/ili biblioteka koji će biti uključeni u *build* procesu.

3.1. Arhitektura

Android OS se sastoji od softverskih komponenata grubo podijeljenih u pet dijelova (slika 2).



Slika 2. Arhitektura Android OS-a[5]

[5] Slika preuzeta sa: https://commons.wikimedia.org/wiki/File:Android-System-Architecture.svg

Analizirajući prema nivoima, na dnu se nalazi dio koji se odnosi na Linux kernel koji sadrži sve bitne upravljačke programe kao što su displej, audio, WiFi, tastatura, i drugi. Iznad Linux kernela je dio sa skupom biblioteka (*Libraries*) koji, pored ostalog, uključuje Libc, odnosno WebKit *engine* – softversku komponentu za predstavljanje web stranica u pretraživačima. U ovom dijelu se nalaze i SSL biblioteke zadužene za Internet sigurnost, te SQLite baza podataka, koja predstavlja repozitorij za čuvanje i dijeljenje podataka aplikacije. Android biblioteke obuhvataju biblioteke koje su specifične za Android razvoj. U nivou biblioteka se nalazi (*Runtime*) sekcija koja sadrži komponentu *Dalvik Virtual Machine* (DVM), vrsta *Java Virtual Machine* (JVM) koja je posebno dizajnirana i optimizirana za Android. Pored DVM, ova sekcija obezbjeđuje i skup biblioteka koje omogućavaju programerima Android aplikacija da razvijaju aplikacije koristeći standard Java programskog jezika. Nivo (*Application Framework*) koji se nalazi iznad biblioteka osigurava servise za aplikacije u formi Java klasa. Na vrhu arhitekture Android OS se nalazi nivo aplikacija.

3.2. Aplikacije

Za razvoj Android aplikacija koristi se Android Studio *Integrated Development Environment* (IDE), posebna verzija IntelliJ IDE-a. Android Studio sadrži vlastiti emulator sa raznim verzijama mobilnih telefona i Android verzija. Pored ovog emulatora, preporuka je korištenje vlastitih telefona za testiranje, ili Genymotion Android Emulator-a[6], radi brzine i opterećenosti.

3.2.1. Arhitektura

Prilikom razvoja mobilne aplikacije, podaci se obično dobijaju iz različitih izvora. Nakon toga slijedi njihovo preoblikovanje da bi odgovarali poslovnoj logici aplikacije, te ih je na kraju potrebno prikazati krajnjem korisniku u nekoj određenoj formi. Ukoliko svi navedeni koraci budu implementirani na jednom mjestu tada može doći do stvaranja poteškoća u daljem razvoju aplikacije. Ovaka arhitektura se naziva jednoslojnom arhitekturom i prihvatljiva je samo u situacijama kada je riječ o jednostavnom projektu koji se neće mijenjati i proširivati. U drugim situacijama potrebno je koristiti neku kompleksniju arhitekturu s ciljem da se izbjegnu ovakve vrste problema.

[6] https://www.genymotion.com/

Arhitektura aplikacije definiše kako komponente sistema trebaju biti organizovane, kako će se vršiti komunikacija među njima, te koja su to ograničenja koja posjeduje cjelokupni projekat, a koja će biti potrebno uzeti u obzir već u procesu planiranja. Ideja korištenja određene arhitekture u razvoju mobilnih aplikacija jeste da se omogući dizajniranje aplikacije na način da njen rast ne otežava proces održavanja, dodavanja novih funkcionalnosti ili mijenjanja već postojećih. Postavljanje arhitekture uključuje donošenje niza odluka, zasnovanih na različitim faktorima, uzimajući u obzir da je potrebno zadovoljiti sve tehničke i funkcionalne zahtjeve, ali u isto vrijeme i pokušati optimizirati osobine kao što su: performanse, sigurnost, mogućnost nadgrađivanja, održavanja i slično. Postavljanje dobrih osnova je najbitniji korak u razvoju bilo koje aplikacije, jer svaki propust napravljen na početku se vremenom pojavi, a često je korekcija grešaka mnogo skuplji i vremenski zahtijevniji proces od samog planiranja. Prilikom kreiranja arhitekture, potrebno je pretpostaviti da će se dizajn u skladu sa zahtjevima korisnika vjerovatno vremenom mijenjati. Nemoguće je predvidjeti sve, ali je važno na početku analizirati sistem iz šire perspektive i planirati njegovo kreiranje na način da bude spreman na promjene.

U razvoju mobilnih aplikacija postoje dva koncepta koja su od posebnog značaja i na koje je potrebno obratiti pažnju, a to su [19]:

1. Razdvajanje nadležnosti
2. Unit testiranje

Razdvajanje nadležnosti se odnosi na podjelu poslova između komponenti unutar aplikacije. Svaka komponenta je zadužena za obavljanje svog zadatka i ne bi trebala obavljati poslove drugih komponenti već samo komunicirati s njima. Za aplikaciju koja slijedi ovakav pristup kažemo da je modularna. Modularnost se postiže enkapsuliranjem informacija unutar jedne komponente koja ima dobro definisan interfejs prema ostalim komponentama.

Unit – testiranje predstavlja važan korak u razvoju bilo koje aplikacije. Testovi se ne pišu samo da bi se pokazalo da programski kôd radi, već s ciljem da se pronađu i otklone eventualne greške. Jedan od najstarijih, ali veoma efektivnih načina testiranja jeste Unit testiranje. Aplikaciju bi trebalo razvijati na način da je pisanje Unit testova jednostavno. Unit testovi se pišu za najmanju gradivnu jedinicu, klasu ili metodu. Ovi testovi su

vrlo važni, jer se testiraju pojedinačne komponente, nezavisno od ostalih dijelova sistema.

3.2.1.1. Višeslojna arhitektura

Osnovna logička arhitektura bilo koje aplikacije podrazumijeva njenu podjelu u tri sloja [19]:

1. Prezentacijski sloj
2. Sloj poslovne logike
3. Sloj za pristup podacima

Sloj se obično posmatra kao neka vrsta "crne kutije" sa interfejsom koji definiše ulaz ili izlaz i zavisnosti prema drugim slojevima. Glavna funkcija upotrebe slojeva jeste organizacija i razdvajanje nadležnosti. Pored toga, mogućnost ponovne upotrebe sloja u nekoj drugoj aplikaciji može doprinijeti skraćivanju vremena njenog razvoja. Veći stepen ponovne upotrebljivosti zajedno sa odgovarajućom izolacijom funkcija, čini održavanje sistema boljim i jednostavnijim.

Prezentacijski sloj

Ni jedna aplikacija ne bi bila posebno korisna bez korisničkog interfejsa. Bez obzira na kvalitet kôda koji se nalazi u pozadini, vrlo je važno kako prikazati podatke krajnjim korisnicima. Prezentacijski sloj se sastoji od dvije osnovne komponente: korisničkog interfejsa i prezentacijske logike, odnosno UI logike. Korisnički interfejs sadrži elemente pomoću kojih krajnji korisnik vrši interakciju sa aplikacijom. Cilj je kreirati takav interfejs da je jednostavan, upotrebljiv i intuitivan. Postoji nekoliko važnih smjernica prilikom dizajna koje je preporučljivo pratiti kako bi se poboljšao kvalitet aplikacije.

Prezentacijski sloj je u suštini posrednik između korisnika i aplikacije. On pruža alate koji omogućavaju interakciju korisnika s aplikacijom, ali sadrži i logiku koja je neophodna da bi se adekvatno reagovalo na akcije korisnika.

Neke od dobrih praksi prilikom dizajna korisničkog interfejsa su sljedeće:

– Konzistentnost – ogleda se kroz odabir boja, naziva, fontova i slično. Navigacija bi trebala biti na mjestu na kojem su korisnici mobilnih aplikacija navikli da inače bude, jer će je tu prvo i tražiti. Takođe, funkcionalnosti prikazane na ekranu trebaju odgovarati namjeni na način

da je jasno vidljivo za koju akciju su namijenjene, bez da korisnik mora pretjerano razmišljati o tome.
- Intuitivnost – najbolji interfejsi su jednostavni i intuitivni za korištenje. Korisnik bi trebao da izvede što manje akcija da bi iskoristio određenu funkcionalnost aplikacije. Veličina, boja i položaj svakog elementa rade zajedno, stvarajući jasan put za razumijevanje interfejsa.
- Povratne informacije – U svakom trenutku treba postojati interakcija između korisnika i aplikacije. Vrlo važno je obavještavati korisnika da li su njegove akcije ispravne ili pogrešno urađene. Obavještavanje o stanju greške bi se trebalo obaviti korištenjem jednostavnih poruka sa eventualnim smjernicama kako pravilno postupiti u određenoj situaciji.
- Univerzalna upotrebljivost – Aplikacija bi trebala biti upotrebljiva korisnicima bez obzira na njihovu dob, zdravstvene poteškoće i slično. Bitno je uvažiti potrebe, iskustva, ali i ograničenja korisnika.

Sloj poslovne logike

Sloj poslovne logike je zadužen za obavljanje specifičnih procesa manipulisanja objektima ovisno o poslovnoj prirodi aplikacije. Ovaj sloj ne smije sadržavati nikakav kôd koji se odnosi na funkcionalnosti korisničkog interfejsa, i treba biti neovisan o izvoru podataka. Svaki sloj predstavlja zasebnu logičku cjelinu sistema i zbog toga se zadaci pojedinačnih slojeva ne smiju miješati. Sloj poslovne logike sadrži obično vrlo složene objekte i funkcionalne algoritme koji implementiraju ključne procese unutar aplikacije nad više entiteta iz baze podataka. Za ovaj sloj se kaže da je "centar" bilo koje slojevite arhitekture, jer sadrži većinu logike aplikacije. Prilikom implementacije vrlo je važno odmah na početku ustanoviti koji su zahtjevi korisnika, kao i ograničenja koja je potrebno uzeti u obzir. Važno je napomenuti da se u ovom sloju mora posebno obratiti pažnja na sigurnost, jer se radi sa (trajnim) podacima. Obično se sigurnost bazira na privilegijama, odnosno pravima pristupa, što znači da određenim dijelovima aplikacije mogu pristupiti samo autorizovani korisnici.

Sloj za pristup podacima

U sloju za pristup podacima se obavljaju svi zadaci koji se odnose na čitanje ili upis u bazu. Ovo je jedini sloj čija implementacija zavisi od strukture baze. Za ostale slojeve, ovaj sloj bi trebao biti samo "crna kutija", kojem se šalju podaci koje je potrebno upisati u bazu ili od kojeg se zahtijeva čitanje iz baze. Sloj za pristup podacima ima zadatak da omogući trajno

smiještanje podatka i CRUD (*Create*, *Read*, *Update*, *Delete*) operacije. Nadležan je da omogući i upravljanje transakcijama.

3.2.1.2. Model-View-Controller *arhitektura*

Model–View–Controller (MVC) arhitektura je originalno razvijena za desktop aplikacije, ali se danas koristi i u pojedinim web, i rjeđe u mobilnim aplikacijama. Prvi put je javno prezentovana već 1988. godine, a do današnjeg dana je doživjela mnogo promjena. Osnovna ideja MVC arhitekture jeste podjela kôda u zasebne cjeline koje međusobno komuniciraju prilikom obavljanja različitih akcija. Svaka cjelina ima određene zadatke i odgovornosti i te odgovornosti se ne bi trebale miješati. Ovakvom podjelom kôd postaje čitljiviji, ali ono što je važnije, postaje pogodniji za održavanje i nadgradnju.

MVC arhitektura se sastoji od tri osnovne komponente [20]:

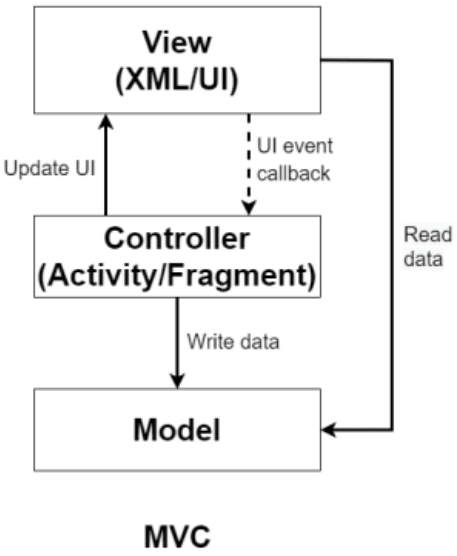
1. *Model*
2. *View*
3. *Controller*

Model predstavlja komponentu koja je odgovorna za podatke koji se koriste unutar aplikacije. Ovo je tzv. "sloj za pristup podacima" koji je korisniku skriven i udaljen, jer sadrži entitete koji upravljaju bazom podataka, mrežnim zahtjevima i bave se pretvaranjem podataka iz jednog oblika u drugi, kako bi oni bili spremi za prikaz krajnjem korisniku.

View komponenta ili tzv. "prezentacijski sloj" predstavlja vizuelnu prezentacija modela. Ima zadatak da primljene podatke prikaže korisniku na odgovarajući način, ne vodeći računa o tome odakle ti podaci dolaze.

Controller komponenta ili tzv. "sloj poslovne logike" ima zadatak da poveže korisničke interakcije sa slojem modela, koji dalje dohvata podatke i pretvara ih u korisne strukture. Ova komponenta sadrži osnovnu logiku aplikacije koja je potrebna za upravljanje zahtjevima korisnika. Ima zadatak i da provjeri da li su primljeni podaci u ispravnom stanju, i shodno tome pošalje upute prezentacijskom sloju kako prikazati korisniku određeno stanje.

Na slici 3. je prikazan način komunikacije između komponenti koje čine MVC arhitekturu.



Slika 3. Model View Controller arhitektura[7]

I *Controller* i *View* komponente zavise od *Modela*. *Controller* komponenta ažurira podatke unutar *Model* komponente, a *View* ih dohvata i prikazuje korisniku.

3.2.1.3. Model-View-Presenter *arhitektura*

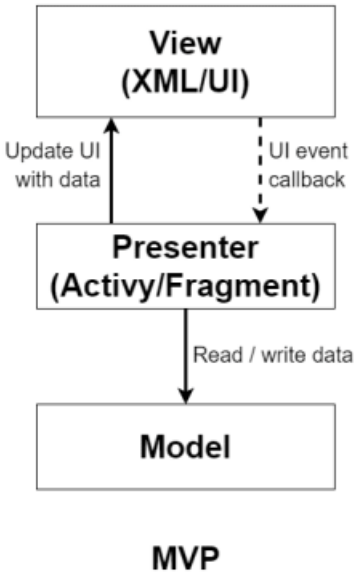
Model–View–Presenter (MVP) trenutno predstavlja najzastupljeniju arhitekturu kada su u pitanju mobilne aplikacije. To je arhitekturalni pattern korisničkog interfejsa koji omogućava automatizirano testiranje, i razdvajanje odgovornosti u prezentacijskoj logici. Korištenjem MVP arhitekture i podjele u tri sloja omogućeno je nezavisno testiranje svakog od njih. MVP ima sličnosti s MVC patternom, s tim da se u ovoj verziji MVC arhitekture nastojalo jasnije razdvojiti koja je uloga *Controller* komponente, a koja *View* komponente. Glavna motivacija za prelazak na MVP arhitekturu jeste viši stepen modularnosti kôda, bolja podjela odgovoronosti u prezentacijskom sloju, kao i bolji temelj za automatsko testiranje.

MVP arhitektura se sastoji od tri komponente [21]:

1. *Model*
2. *View*
3. *Presenter*

[7] Slika preuzeta sa: https://dev.to/vtsen/mvc-vs-mvp-vs-mvvm-design-patterns-443n

kako je predstavljeno na slici 4.



Slika 4. Model View Presenter arhitektura[8]

Model komponenta ostaje ista kao i kod MVC arhitekture. Implementirana je na isti način i ima istu ulogu.

View komponenta MVP arhitekture se razlikuje od *View* komponente MVC arhitekture. Ona podrazumijeva skup svih aktivnosti, fragmenata i ostalih elemenata u kojima se nalazi logika samo za prikaz podataka na ekranu i eventualno prosljeđivanje podataka (koje je korisnik unio) narednoj komponenti, odnosno *Presenter*-u. Aktivnosti i fragmenti ne mogu direktno komunicirati s *Model* komponentom, već sve potrebne podatke dobijaju od *Presenter*-a. Na ovaj način *View* komponenta je mnogo jednostavnija, jer se sva prezentacijska logika nalazi unutar *Presenter* komponente. Generalno, mapiranje između *View* i *Presenter* komponenti je 1-1, gdje svaki *View* uglavom sadrži instancu odgovarajućeg *Presenter*-a.

Presenter komponenta predstavlja posrednika između *View* i *Model* komponenti. On uzima podatke iz Model-a i u odgovarajućem obliku ih prosljeđuje *View* komponenti. Također, dobija podatke od *View* komponente i prosljeđuje ih *Model* komponenti.

[8] Slika preuzeta sa: https://dev.to/vtsen/mvc-vs-mvp-vs-mvvm-design-patterns-443n

3.2.1.4. Model-View-ViewModel *arhitektura*

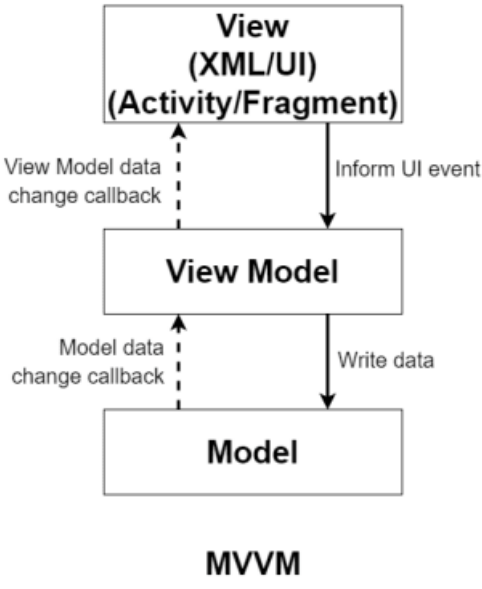
Jedan od velikih problema MVC i MVP arhitektura je taj što su *Controller* i *Presenter* komponente u uskoj vezi s *View* komponentom što otežava proces testiranja. Zbog toga je došlo do razvoja nove arhitekture pod nazivom *Model–View–ViewModel* (MVVM) koja je danas preporučena, jer je na efikasan način ispoštovan princip razdvajanja nadležnosti, pa je testiranje mnogo efikasnije i jednostavnije [19].

Korištenje MVVM arhitekture se preporučuje u slučaju kompleksnijih aplikacija, jer je tada potrebno imati više nivoa apstrakcije kako bi održavanje, nadgradnja i testiranje bilo efikasnije. Međutim, u slučaju aplikacija s jednostavnijim korisničkim interfejsom bolje je koristiti npr. MVP arhitekturu.

I ova arhitektura se sastoji od tri osnovne komponente [22]:

1. *Model*
2. *View*
3. *ViewModel*

kako je predstavljeno na sljedećoj slici 5.



Slika 5. Model View View Model arhitektura9

[9] Slika preuzeta sa: https://dev.to/vtsen/mvc-vs-mvp-vs-mvvm-design-patterns-443n

Model komponenta, kao i u prethodno objašnjenim arhitekturama, ima zadatak da prihvati određene podatke i smjesti ih u bazu ili da dobavi podatke iz baze ili sa nekog udaljenog servisa i preda ih komponenti koja ih očekuje. Ona sadrži klase koje odgovaraju podacima koji se spremaju ili dobavljaju.

View je komponenta koja ne treba sadržavati nikakvu logiku koja se tiče dobavljanja podataka ili manipulisanja njima. Njen zadatak u MVVM arhitekturi je da se "pretplati" na podatke čije promjene želi "osluškivati" i da shodno tome vrši ažuriranje korisničkog interfejsa.

ViewModel komponenta je zadužena za manipulaciju svim podacima koji će se prikazati korisniku na ekranu. *ViewModel* dohvata podatke iz *Model* komponente, priprema ih, i omogućava da se bilo koji *View* može "pretplatiti" na osluškivanje promjena koje se dešavaju nad tim podacima. To znači da jedan *ViewModel* može pripremati podatke za više različitih *View* komponenti. *View* i *Model* komponente komuniciraju isključivo preko *ViewModel* komponente i to je jedini vid komunikacije između njih. Za razliku od *Controller* i *Presenter* komponenti koje su korištene u MVC i MVP arhitekturi, kod MVVM arhitekture ni jedna *ViewModel* komponenta nema nikakvo znanje o tome koji *View* će osluškivati promjene podataka koje ona sadrži [22].

Često se između *ViewModel* i *Model* komponente kreira *Repository* komponenta koja ima instancu odgovarajuće *DAO* klase koja služi za pristup bazi podataka. Na ovaj način *ViewModel* komponenta ima zadatak da prikupi podatke od *View*-a i da ih proslijedi *Repository*-u koji preuzima svu komunikaciju sa *Model* komponentom. *ViewModel* tada ima ulogu da dobijene podatke izloži na način da ih ostale *View* komponente mogu osluškivati.

MVVM arhitektura je veoma dobar izbor u situacijama kada je potrebno krerati kompleksnije aplikacije, jer se na taj način olakšava njihovo testiranje, održavanje i nadgrađivanje. Za jednostavnije aplikacije može se koristiti MVP ili MVC, s tim da MVC polako izlazi iz upotrebe zbog nedovoljne definisanosti i razdvojenosti zadataka pojedinih komponenti.

3.2.2. Komponente

Osnovne komponente/gradivni blokovi Android aplikacije su:

Activity[10] – određuje korisnički interfejs i upravlja korisničkim aktivnostima na ekranu telefona. Aktivnost predstavlja (pojedinačan) ekran sa korisničkim interfejsom. Ova komponenta aplikacije izvršava aktivnosti na

[10] http://developer.android.com/reference/android/app/Activity.html

ekranu. Na primjer, u aplikaciji za pregled slika, jedna aktivnost može biti ekran na kojem je prikazana lista svih slika, dok druga može biti lista opcija za selektovanu sliku. Ako aplikacija ima više od jedne aktivnosti, onda bi jedna od njih trebala biti označena kao aktivnost koja će biti prikazana kada se pokrene aplikacija.

Service[11] – je komponenta koja se izvršava u pozadini i obavlja operacije koje se dugo izvšavaju, a povezane su s aplikacijom. Servis također može da radi za neke vanjske procese. Tako, na primjer, servis komponenta može obavljati dugotrajnu operaciju prenosa podataka preko mreže, a da se pri tome ne zaustavlja korisničko izvršenje akcije. Ova komponenta ne obezbjeđuje korisnički interfejs. Primjer jednog servisa je preuzimanje podataka sa web servera za aplikaciju vremenske prognoze.

Broadcast receiver[12] – je komponenta koja odgovara na *broadcast* objave (prijemnik obavijesti). Upravlja komunikacijom između Android OS i aplikacija, te odgovaraju na poruke dobijene od drugih aplikacija ili sistema. Aplikacije mogu uputiti poruku drugim aplikacijama obavještavajući ih da su neki podaci preuzeti, da se nalaze na uređaju i da su raspoloživi za korištenje. Prijemnik obavijesti će prepoznati ovu komunikaciju i pokrenuti odgovarajuću akciju. Mnogi prijemnici obavijesti potiču od samog sistema – kao što je, na primjer, obavještenje da je baterija skoro prazna. Oni nemaju svoj korisnički interfejs, ali mogu kreirati notifikaciju u statusnoj traci.

Content provider[13] – upravlja dijeljenim skupom podataka aplikacije (provajder sadržaja). Ova komponenta (na zahtjev) obezbjeđuje podatke iz jedne aplikacije drugima. Takvi podaci mogu biti sačuvani u SQLite bazi podataka, datotečnom sistemu, web-u ili nekoj drugoj lokaciji.

Fragment[14] – predstavlja dio korisničkog interfejsa u nekoj aktivnosti. Jedna aktivnost može prikazati više od jednog fragmenta na ekranu istovremeno.

View[15] – predstavlja elemente korisničkog interfejsa (*User Interface* - UI) Android aplikacije, kao što su *Button*, *Label*, *Text*, i drufi. Sve što korisnik vidi i s čime vrši interakciju predstavlja *View*. Više *View* objekata se može grupisati u *ViewGroup*.

---

[11] https://developer.android.com/reference/android/app/Service

[12] http://developer.android.com/reference/android/content/BroadcastReceiver.html

[13] https://developer.android.com/guide/topics/providers/content-providers

[14] https://developer.android.com/guide/components/fragments

[15] https://developer.android.com/reference/android/view/View

Intent[16] – predstavlja abstraktnu definiciju željene akcije (namjera). Koristi se za pozivanje komponenata, kao što je, na primjer, omogućavanje pokretanja servisa, aktivnosti, prikazivanja web stranice, liste kontakata, slanje poruka, itd.

Manifest[17] – *AndroidManifest.xml* je konfiguraciona datoteka za aplikaciju koja sadrži sve informacije o aktivnostima, provajderima sadržaja, permisijama, itd. Sve aplikacije treba da sadrže ovu datoteku.

3.2.3. UI widgets

Svi elementi korisničkog interfejsa Android aplikacije su smješteni u objekte klasa `View` i `ViewGroup`. `View` direktno sadrži elemenat UI koji će biti iscrtan na ekranu i sa kojim će korisnik imati interakciju. `ViewGroup` služi za grupisanje više `View` objekata. Struktura prikaza UI se definiše s *layout*-om. *Layout* se može napraviti na dva načina: pišući XML kôd i instancirati *layout* putem kôda koristeći `View` i `ViewGroup`.

Neki od osnovnih *layout*-a su: `LinearLayout`, `RelativeLayout`, `ListView`, `GridView`, `ConstraintLayout` i `RecyclerView` [23].

3.2.3.1. LinearLayout

`LinearLayout` grupiše *view* objekte tako da ih poravnava u jednom pravcu horizontalno ili vertikalno (slika 6). Pravac poravnavanja se određuje atributom `android:orientation`.

Slika 6. Linear Layout[18]

[16] https://developer.android.com/reference/android/content/Intent

[17] https://developer.android.com/guide/topics/manifest/manifest-intro

[18] Slika preuzeta sa: https://developer.android.com/develop/ui/views/layout/linear#:~:text=To%20create%20a%20linear%20layout,each%20view%20to%20%221%22%20.

3.2.3.2. RelativeLayout

`RelativeLayout` grupiše *view* objekte na način da ih prikazuje koristeći relativne pozicije (slika 7). Pozicija svakog *view* objekta se može zadati u odnosu na susjedne *view* objekte tako da se postavi, na primjer, lijevo od nekog *view*-a, ispod i slično. Kako svaki *view* ima svoj jedinstveni broj omogućeno je da se pomoću njega definišu međusobne relacije između *view*-ova. Neki od načina definisanja pozicija elemenata u odnosu na druge su pomoću atributa: - `android:layout_below = "@+id/elementB"` - postavlja gornju ivicu trenutnog elementa direktno ispod elementa sa id-em "elementB". - `android:layout_toRightOf = "@+id/elementB"` - postavlja lijevu ivicu trenutnog elementa pored elementa sa id-em "elementB" sa njegove desne strane.

Slika 7. Relative Layout[19]

3.2.3.3. ListView

`ListView` grupiše elemente i prikazuje ih kao listu koju je moguće skrolati. Elementi liste se automatski unose u listu koristeći `Adapter` koji dobavlja sadržaj iz izvora podataka poput niza ili upita prema bazi i konvertuje svaki element rezultata u jedan *view* koji se dodaje u listu.

3.2.3.4. GridView

`GridView` grupiše elemente i prikazuje ih na dvodimenzionalnoj mreži koju je moguće skrolati. Elementi `GridView`-a se automatski dodaju putem `ListAdaptera`.

[19] Slika preuzeta sa: https://developer.android.com/develop/ui/views/layout/relative

3.2.3.5. ConstraintLayout

`ConstraintLayout` omogućava kreiranje velikih i kompleksnih *layout*-a bez hijerarhije. Sličan je `RelativeLayout`-u gdje su definisane veze između svih `View`-ova i roditeljskih (eng. *parent*) *layout*-a, ali je daleko fleksibilniji i lakši za rad u Android Studio editor-u.

3.2.3.6. RecyclerView

`RecyclerView` predstavlja `ViewGroup` koja sadrži sve `View`-ove koji odgovaraju podacima. `RecyclerView` predstavlja komponentu koja omogućava efikasno prikazivanje velikog skupa podataka. Kreator definiše podatke i kako svaka stavka treba da izgleda i `RecyclerView` dinamički kreira elementa kada su oni potrebni.

`RecyclerView` reciklira individualne elemente. Kada se izvrši scroll van samog ekrana, ova komponenta ne uništava `View`, nego ponovno iskoristi, odnosno reciklira `View` za nove elemente. Ovo značajno poboljšava performanse aplikacije.

3.2.4. Ulazne kontrole

Ulazne kontrole su komponente koje omogućavaju interakciju korisnika sa korisničkim interfejsom. Android posjeduje mnoge ulazne kontrole, a neke on najkorištenijih su:

Button – dugme koje je moguće pritisnuti kako bi se obavila dodijeljena akcija. Klasa koja omogućava ovu kontrolu je `Button`. Atributom `android:onClick` zadajemo koja će se funkcija prilikom pritiska dugmeta pozvati.

Text field – tekstualno polje koje dozvoljava unos teksta. Klasa koja omogućava ovu kontrolu je `EditText`.

Checkbox – prekidač tipa da/ne čiju vrijednost korisnik može mijenjati. Ova kontrola se koristi kada je potrebno omogućiti izbor više opcija koje nisu međusobno isključive. Klasa u kojoj je ova kontrola implementirana je `Checkbox`.

Radio button – kontrola slična *Checkbox*-u s tim da samo jedna od opcija iz grupe može biti odabrana. Klase koje se koriste za implementaciju ove kontrole su `RadioGroup` i `RadioButton`.

Spinner – *drop-down* lista koja omogućava korisniku izbor jedne vrijednosti iz skupa ponuđenih. Klasa kojom se implementira ova kontrola se naziva `Spinner`.

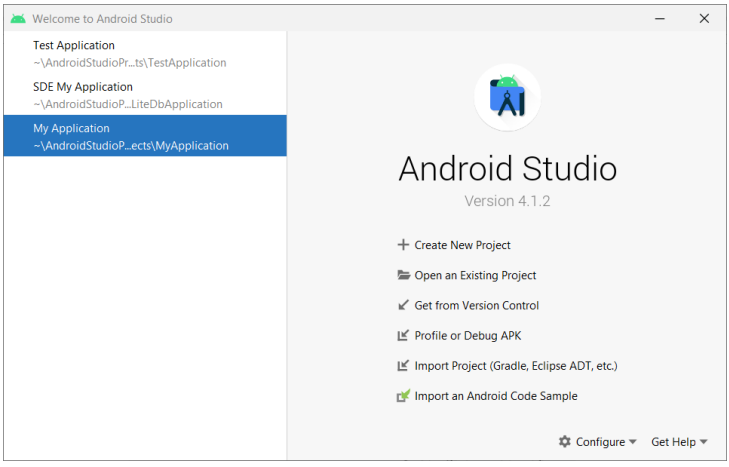
Pickers – kontrole tipa dijaloškog okvira koje omogućavaju odabir vrijednosti iz skupa koristeći dugmad gore/dole ili *swipe* pokret. Klase koje omogućavaju odabir datuma i vremena na ovaj način su `DatePicker` i `TimePicker`.

3.2.5. Ulazni događaji

Interakcija korisnika s Android aplikcijom se može detektovati na više načina. Princip je kada se dogodi neki događaj (eng. *event*) da se pozove odgovarajuća funkcija koja je dodijeljena u *event listener*-u, koji predstavlja interfejs u `View` klasi koja sadrži jednu *callback* metodu. Ova metoda se poziva kada Android detektuje odgovarajuću interakciju korisnika sa aplikacijom. Više o event listenerima se može pronaći na: Input events overview[20]

3.2.6. Razvoj u Android Studio-u

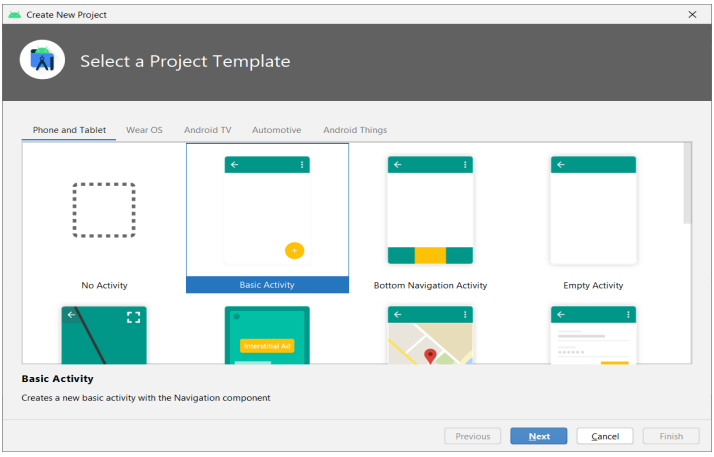
Prije početka razvoja aplikacije pomoću Android SDK potrebno je postaviti Android razvojno okruženje, nakon čega je moguće otvoriti Android Studio i kreirati aplikaciju, odnosno projekat u Android Studio-u. Nova aplikacija/projekat u Android Studio-u se može kreirati tako da se nakon pokretanja Android Studio-a izabere *Create New project*



Snimak zaslona 1. Početni ekran Android Studio

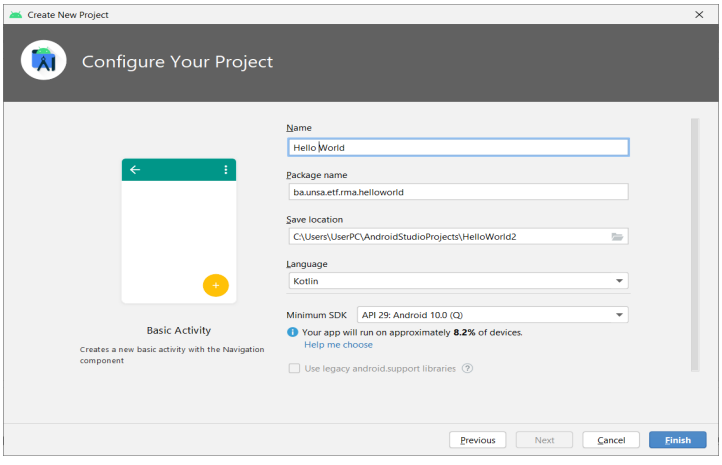
[20] https://developer.android.com/guide/topics/ui/ui-events

Nova aplikacija/projekat se može kreirati i u radnom okruženju Android Studio-a, izborom *File -> New -> New project...*. U narednom koraku (u *Select a Project Template*) je potrebno odabrati *template* projekta/aplikacije (npr. *Basic Activity*).



Snimak zaslona 2. Odabir Template-a projekta

Zatim (u *Configure Your Project*) je potrebno konfigurisati projekat/aplikaciju, odnosno unijeti podatke koji se odnose na *Name* (npr. Hello World), *Package name* (npr. ba.unsa.etf.rma.helloworld), *Save location* (npr. C:\Users\UserPC\AndroidStudioProjects\HelloWorld2), *Language* (Kotlin) i *Minimum SDK* (API 29).

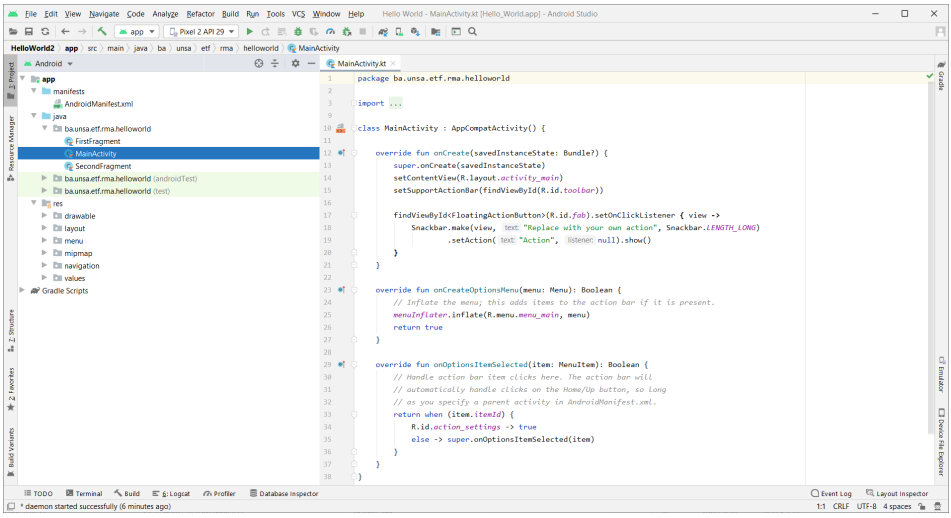


Snimak zaslona 3. Konfiguracija projekta

U nastavku, izborom *Finish*, Android Studio:

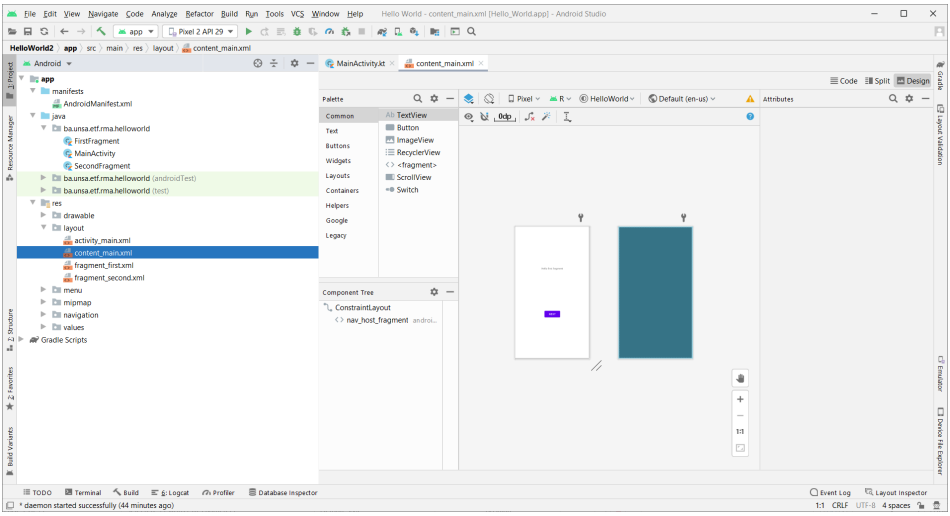
- kreira folder Android Studio projekta (koji se obično nalazi u folderu s nazivom *AndroidStudioProjects* ispod korisnikovog *home* direktorija)
- kreira projekat, što može potrajati nekoliko trenutaka. Android Studio koristi *Gradle* kao svoj sistem za gradnju, a napredak izrade projekta se može pratiti na dnu prozora Android Studio-a
- otvara editor kôda u kojem je prikazan projekat

Proces kreiranja novog projekta/aplikacije bi se trebao završiti (sličnim) izgledom prozora prikazanim u nastavku (ako je aktivna `MainActivity.kt` datoteka):



Snimak zaslona 4. Novokreirani projekat

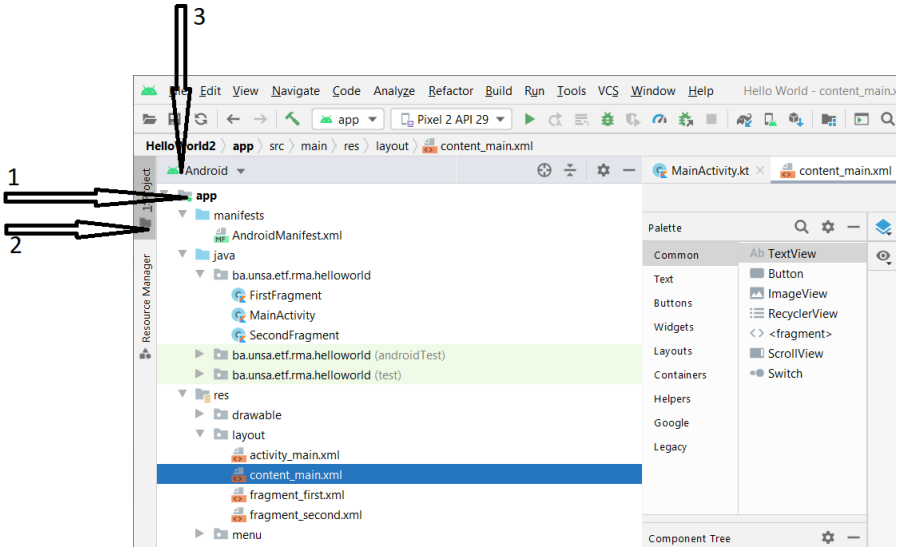
odnosno, izgledom prozora (ako je aktivna/odabrana `content_main.xml` datoteka)



Snimak zaslona 5. Layout novokreirane aplikacije

3.2.7. Struktura

Struktura Android projekta se sastoji od nekoliko foldera i datoteka. Na osnovu izabrane osnovne aktivnosti (*Basic Activity* template) za projekat, Android Studio kreira nekoliko datoteka i foldera, kako je prikazano na slici u nastavku.



Snimak zaslona 6. Struktura foldera

Hijerarhija datoteka i foldera aplikacije se može pogledati na više načina, od kojih je jedan predstavljen u *Project*-u (2). *Project* prikazuje datoteke i foldere strukturirane na način koji je pogodan za rad s Android projektom. Takođe, hijerarhija i prikaz datoteka se može predstaviti i klikom na (3). S dvostrukim klikon na `app` (1) folder moguće je prikazati hijerarhiju foldera i datoteka aplikacije, dok se klikom na *Project* (2) može sakriti ili prikazati izgled projekta.

U *Project-> Android* dijelu su prikazani folderi najvišeg nivoa ispod `app` foldera aplikacije: `manifests`, `java` i `res`.

Nakon što se proširi `manifests` folder moguće je vidjeti da se na njemu nalazi `AndroidManifest.xml` datoteka. Ova datoteka opisuje sve komponente Android aplikacije i Android *runtime* sistem je čita kada se aplikacija pokreće na izvršavanje.

U `java` folderu su smještene i organizovane sve Kotlin datoteke koje se mogu prikazati nakon što se ovaj folder proširi. Android projekat na ovom folderu čuva sve Kotlin datoteke. Za generisani primjer, folder `java` sadrži tri podfoldera:

- `ba.unsa.etf.rma.helloworld` (ili neko drugo ime domena koje je korisnik naveo u procesu kreiranja projekta/aplikacije) - sadrži Kotlin datoteke izvornog koda za aplikaciju
- *`ba.unsa.etf.rma.helloworld (androidTest)`* - predstavlja mjesto gdje bi se trebali smjestiti instrumentirani testovi, odnosno testovi koji se izvode na Android uređaju
- *`ba.unsa.etf.rma.helloworld (test)`* - mjesto gdje bi se trebali nalaziti *unit* testovi, a kojima nije potreban Android uređaj za pokretanje

Folder `res` sadrži sve resurse aplikacije kao što su XML *layout*-i, UI stringovi i slike koji su podijeljeni u odgovarajuće direktorije. Njegovi podfolderi su:

- `drawable` – na njemu se nalaze sve slike aplikacije
- *`layout`* – sadrži datoteke korisničkog interfejsa. U trenutku inicijalnog kreiranja, aplikacija ima jednu aktivnost (datoteka `MainActivity.kt`) čija *layout* datoteka ima naziv `activity_main.xml`. Takođe, ovaj folder sadrži i `content_main.xml`, *`fragment_first.xml`* i *`fragment_second.xml`* *layout* datoteke.
- *menu* – na njemu se nalaze XML datoteke koje opisuju meni korisničke aplikacije

– *mipmap* – ovaj folder sadrži ikone korisničke aplikacije
– *navigation* – sadrži graf za navigaciju koji daje informacije Androidu Studio-u u pogledu kretanja između različitih dijelova korisničke aplikacije
– *values* – sadrži resurse, kao što su stingovi i boje, koji se koriste u aplikaciji

Najvažnije datoteke aplikacije su:

MainActivity – predstavlja kôd glavne aktivnosti. Sadržaj programskog kôda ove datoteke automatski generisane wizardom za *HelloWorld* aplikaciju je:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(findViewById(R.id.toolbar))
    }
}
```

U `onCreate()` metodi koja se izvrši kada se učita aktivnost, `R.layout.activity_main` se odnosi na `activity_main.xml` datoteku koja se nalazi u `res/layout` folderu. Ovaj kôd označava da će se prilikom kreiranja aktivnosti prikazati odgovarajući *layout* `activity_main`. *Layout*-u se pristupa kroz klasu `R` koja označava sve interne resurse aplikacije. Resursi Androida su dostupni kroz `android.R`.

Manifest – svaka komponenta koja predstavlja dio aplikacije mora biti navedena u manifest datoteci sa nazivom `AndroidManifest.xml`. Ova datoteka ima ulogu interfejsa između Android OS-a i aplikacije. To znači da Android OS neće razmatrati komponentu ukoliko nije naznačena u manifest datoteci. U nastavku slijedi sadržaj `AndroidManifest.xml` datoteke za navedeni generisani primjer aplikacije:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="ba.unsa.etf.rma.helloworld">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
```

```
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.HelloWorld">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/Theme.HelloWorld.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.
                MAIN" />
                <category android:name="android.intent.category.
                LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

U navedenom sadržaju manifest datoteke, početni `<application>` i završni `</application>` tagovi obuhvataju komponente aplikacije. Atribut `android:icon` pokazuje na ikonu aplikacije koja se nalazi na `res/mipmap`, te da aplikacija koristi sliku sa nazivom *`ic_launcher`* koja se nalazi na *`mipmap`* direktoriju.

`<activity>` tag se koristi za specifikaciju aktivnosti. Upotrebom ovog taga može se navesti više (od jedne) aktivnosti. U `android:name` atributu je naznačen naziv klase aktivnosti, a `android:label` atribut predstavlja string koji se koristi za oznaku aktivnosti.

`@string` se referencira na `res/values/strings.xml` datoteku koja će biti opisana u nastavku. `@string/app_name` se odnosi na `app_name` koji je definisan u `strings.xml` datoteci, a to je HelloWorld string. Na sličan način su upotrijebljeni i drugi stringovi u aplikaciji.

Akcija za `intent-filter` ima naziv `android.intent.action.MAIN` i predstavlja ulaznu tačku za aplikaciju. Kategorija za `intent-filter` je `android.intent.category.LAUNCHER` i naznačava da se aplikacija može pokrenuti s ikonom za pokretanje uređaja.

Za specificiranje komponenata Android aplikacije u manifest datoteci se koriste sljedeći tagovi:

`<activity>` - elementi za aktivnosti
`<service>` - elementi za servise
`<receiver>` - elementi za prijemnike obavijesti
`<provider>` - elementi za provajdere sadržaja

Strings – odnosi se na `strings.xml` datoteku koja se nalazi na `res/values` folderu. Ova datoteka određuje tekstualni sadržaj, te sadrži tekstualne elemente koje koristi aplikacija, kao što su nazivi dugmadi, labele, *default*-ni tekst i drugo. Primjer:

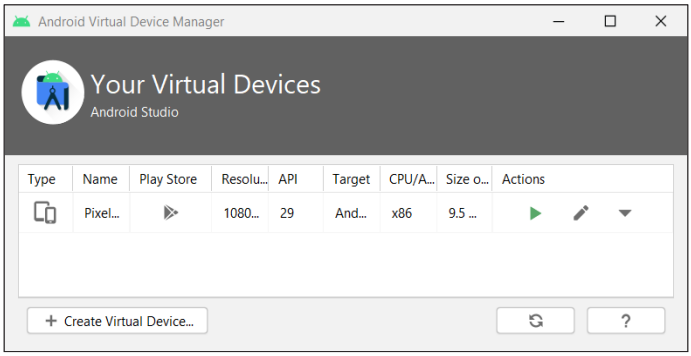
```
<resources>
    <string name=”app_name”>Hello World</string>
    <string name=”next”>Next</string>
    <string name=”previous”>Previous</string>
</resources>
```

Layout – datoteka aplikacije `activity_main.xml` se nalazi na `res/values` folderu. Nju koristi aplikacija u procesu gradnje svog interfejsa. Ova datoteka se može mijenjati po potrebi, kada se želi promijeniti izgled intefejsa aplikacije. Primjer:

```
<?xml version=”1.0” encoding=”utf-8”?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
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=”.MainActivity”>

    <include layout=”@layout/content_main” />
    <com.google.android.material.floatingactionbutton.
FloatingActionButton
        android:id=”@+id/fab”
        android:layout_width=”wrap_content”
        android:layout_height=”wrap_content”
        android:layout_gravity=”bottom|end”
        android:layout_margin=”@dimen/fab_margin”
        app:srcCompat=”@android:drawable/ic_dialog_email” />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

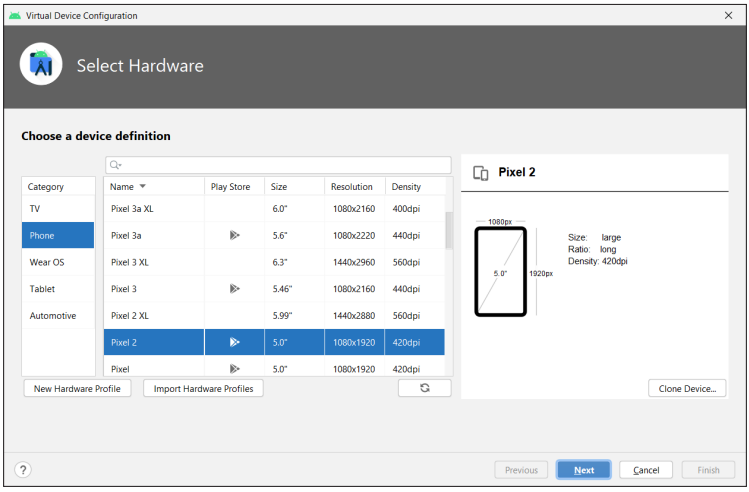
3.2.8. Kreiranje virtuelnog uređaja (emulatora)

Za kreiranje virtualnog uređaja (ili emulatora) koji simulira konfiguraciju za određenu vrstu Android uređaja koristi se *Android Virtual Device* (AVD) menadžer. Prvi korak je kreiranje konfiguracije koja opisuje virtualni uređaj na način da se u Android Studio-u- odabere *Tools-> AVD Manager* ili klikne na ikonu AVD menadžera na traci alata  nakon čega slijedi (*Your Virtual Devices*) prozor sljedećeg izgleda.



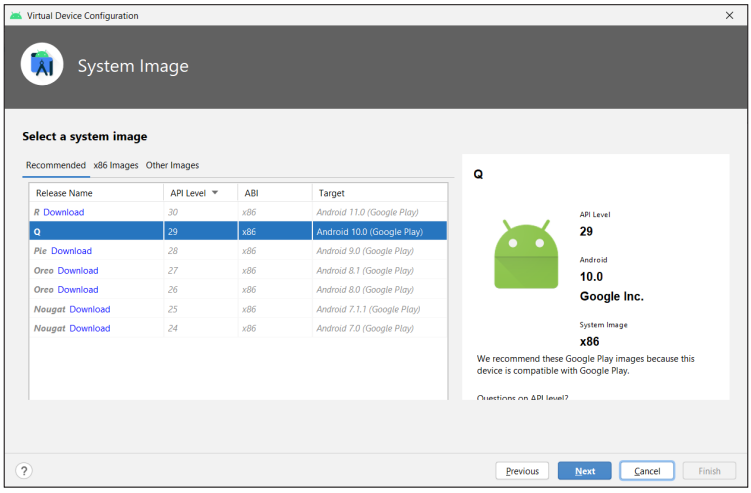
Snimak zaslona 7. Spisak virtuelnih uređaja

Ukoliko su ranije kreirani virtuelni uređaji, oni će biti prikazani u prozoru. Dalje je potrebno odabrati *+Create Virtual Device* koji se nalazi se na dnu prozora. U dijalog prozoru *Select Hardware* je prikazana lista unaprijed konfigurisanih hardverskih uređaja.



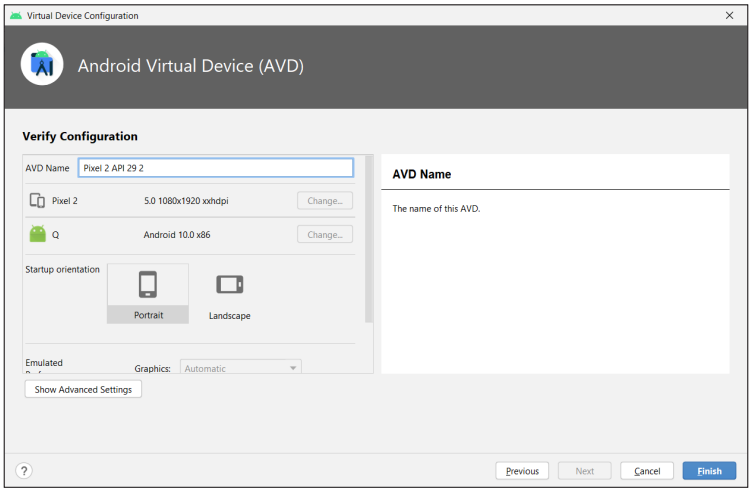
Snimak zaslona 8. Kreiranje novog virtuelnog uređaja/Odabir hardvera

U ovom koraku je potrebno odabrati željeni uređaja (npr. Pixel 2), i kliknuti *Next*. U dijalog prozoru *System Image* (u tabu *Recommended*) potrebno je odabrati željenu verziju Android operativnog sistema.



Snimak zaslona 9. Odabir operativnog sistema

Ako je pored prikazane verzije vidljiv *Download* link to znači da ta verzija nije instalirana. Za njenu instalaciju prvo je potrebno preuzeti je s Interneta klikom na link, a zatim kliknuti na *Next* kada je *download* gotov. U narednom dijalog prozoru *Android Virtual Device (AVD)*-a potrebno je prihvatiti *default*-ne vrijednosti i kliknuti *Finish*.

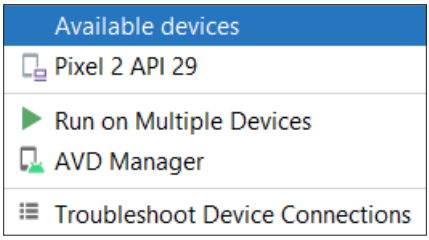


Snimak zaslona 10. Konačna konfiguracija virtuelnog uređaja

nakon čega će AVD menadžer prikazati virtualni uređaj koji je dodan. U slučaju da je *Your Virtual Devices* prozor AVD menadžera virtualnih uređaja i dalje otvoren, potrebno ga je zatvoriti.

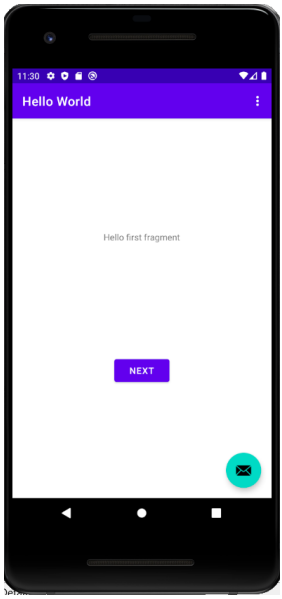
3.2.9. Pokretanje aplikacije na emulatoru

Nakon izbora *Run* > *Select Device* u Android Studio-u, u dijalog prozoru koji slijedi, potrebno je odabrati konfigurisani virtualni uređaj.



Snimak zaslona 11. Odabir uređaja za pokretanje

Za pokretanje aplikacije potrebno je u Android Studio-u odabrati *Run-> Run 'app'* ili na traci alata kliknuti na *Run* ikonu . Android Studio će instalirati aplikaciju na AVD-u i pokrenuti je. Nakon određenog vremena će se aplikacija *build*-ati i instalirati na uređaj, a emulator prozor će imati slijedeći izgled:



Snimak zaslona 12. Pokrenuta aplikacija na uređaju

3.2.10. Aktivnosti

Aktivnosti predstavljaju prezentacijski nivo razvoja aplikacija, odnosno ulaznu tačku interakcije sa korisnikom. Aktivnost olakšava interakcije između korisnika i sistema, kao što su: praćenje za šta je korisnik trenutno zainteresovan, odnosno šta se nalazi na ekranu; vođenje računa da prethodno korišteni procesi sadrže elemente na koje se korisnik može htjeti vratiti; pomaganje aplikaciji u rukovanju procesima, te omogućavanje načina da aplikacije međusobno implementiraju tokove, a sistem da koordinira te tokove. Aktivnost se implementira kao podklasa `Activity` ili `AppCompatActivity` klase [24].

Jedna od aktivnosti treba biti označena kao glavna aktivnost i odnosi se na aktivnost koja se poziva kada se otvori aplikacija. Svaka aktivnost može pozvati neku drugu aktivnost za izvršavanje raznih akcija. Da bi mogla biti korištena u aplikaciji, potrebno je aktivnost registrovati u manifest datoteci. Aktivnost se u manifest datoteci deklariše putem `<activity>` taga u kojem je atribut `android:name` obavezan.

```
class MainActivity : AppCompatActivity() {
}
```

Aktivnostima se u sistemu upravlja po principu steka (aktivnosti). Kada se pokrene nova aktivnost, ona se obično stavlja na vrh steka i postaje trenutno aktivna, dok prethodna aktivnost uvijek ostaje ispod nje u steku i neće se ponovo aktivirati sve dok se nova aktivnost ne završi.

Aktivnost ima četiri stanja:

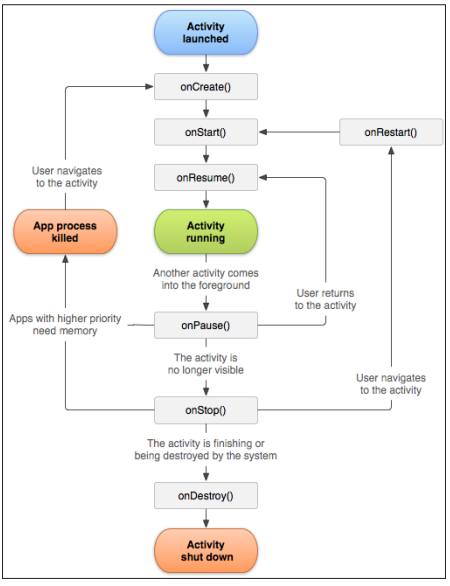
- Aktivna – ako je aktivnost prikazana na ekranu, odnosno izvršava se. To je obično aktivnost s kojom je korisnik u interakciji, odnosno s kojom trenutno komunicira.
- Vidljiva – ako nova aktivnost nije u punoj veličini, a druga aktivnost ima višu poziciju ili se aktivnost ne nalazi u fokusu trenutnog prozora. Takva aktivnost je i dalje živa.
- Zaustavljena – ako je neka aktivnost u potpunosti zaklonjena drugom aktivnosti, odnosno ako je skrivena. Aktivnost i dalje sadrži sve podatke o stanju, ali više nije vidljiva korisniku.
- Završena – sistem može izbaciti aktivnost iz memorije, nakon čega je ona uništena.

Životni ciklus aktivnosti

Android sistem pokreće svoj program u okviru aktivnosti pozivom `onCreate()` metode. Klasa aktivnosti definiše određene događaje, koji su u nastavku grafički prikazani u dijagramu životnog ciklusa aktivnosti (slika 8).

U dijagramu su prikazana najvažnije metode koje predstavljaju pojedina stanja aktivnosti, a to su:

- `onCreate()` – poziva se kada se prvi put kreira aktivnost. Ova metoda se uvijek implementira i predstavlja mjesto gdje korisnik incijalizira aktivnost. U ovoj metodi se poziva `setContentView()` metoda koja postavlja odgovarajući *layout* za aktivnost.
- `onStart()` – poziva se kada aktivnost treba postati vidljiva korisniku. Poziva se nakon `onCreate()` pri pokretanju aktivnosti.
- `onResume()` – poziva se kada korisnik počinje interakciju s aplikacijom
- `onPause()` – poziva se kada se zaustavi tekuća aktivnost, a nastavi se prethodna aktivnost
- `onStop()` – poziva se kada aktivnost nije više vidljiva
- `onDestroy()` – poziva se prije nego što sistem uništi aktivnost
- `onRestart()` – poziva se kada se aktivnost ponovo pokrene, nakon što je zaustavljena



Slika 8. Životni ciklus aktivnosti[21]

[21] Slika preuzeta sa: https://developer.android.com/guide/components/activities/activity-lifecycle

3.2.11. Resursi

Tokom razvoja aplikacije potrebno je voditi računa o raznim resursima koje će aplikacija koristiti. Android aplikacija sadrži eksterne elemente kao što su slike, definicije izgleda, bitmape, boje, stringovi korisničkog interfejsa, te ostalo što je vezano za izgled aplikacije. Korištenje resursa olakšava promjenu raznih karakteristika aplikacije, pri čemu nije potrebna promjena kôda. Svi resursi se nalaze unutar `res` foldera. Oni bi trebali biti razdvojeni i trebaju se nalaziti i održavati na različitim podfolderima na `res` folderu, kao što su [25]:

- `drawable` – slike (.png, .jpg, .gif,..)
- `layout` – XML datoteke koje definišu izgled korisničkog interfejsa
- `menu` – XML datoteke koje definišu meni aplikacije
- `mipmap` – sadrži ikone pokretača za aplikaciju
- `navigation` – XML datoteka koja definiše graf za navigaciju koji Android Studio-u govori kako se kretati između različitih dijelova aplikacije
- `values` – XML datoteke koje sadrže brojeve, stringove, boje itd.

Primjer organizacije resursa:

```
HelloWorld/
  app/
     java/
        ba.unsa.etf.rma.hw.helloworld
          MainActivity.kt
     res/
        layout/
          activity_main.xml
          content_main.xml
        menu/
          menu_main.xml
        values/
          colors.xml
          strings.xml
```

3.2.11.1. Alternativni resursi

Aplikacija bi radi bolje optimizacije trebala obezbijediti alternativne resurse, kao što su različiti jezici ili slike različitih rezolucija za različite veličine ekrana. Za svaki resurs Android SDK definiše jedinstveni id za referenciranje na određeni resurs u kôdu. Na primjer, za podršku različitim

rezolucijama ekrana potrebno je uključiti odgovarajuće slike kao alternativni resurs koje će Android učitati kada ustanovi konfiguraciju tekućeg uređaja.

Konfiguracija konkretne alternative za skup resursa se može uključiti tako da se kreira novi poddirektorij na *res/* direktoriju. Taj poddirektorij treba biti u formi `<naziv_resursa>-<konfig_kvalifikator>`, gdje `naziv_resursa` određuje resurs, slično ranije navedenim `layout`, `drawable`, itd., dok `konfig_kvalifikator` određuje konfiguraciju za koju će se taj resurs koristiti. U dokumentaciji Androida se može pronaći kompletna lista kvalifikatora za razne tipove resursa. Alternativni resursi se trebaju sačuvati na novom direktoriju i trebaju biti imenovani isto kao datoteke *default*-nih resursa, ali sadržaj tih datoteka je specifičan za alternativu.

Promjena jezika

Organizovanje tekstova labela u `res/values` omogućava da se isti tekst može pomoću jedinstvene varijable referencirati na različitim mjestima u aplikaciji. Na ovaj način su olakšane promjene teksta i omogućen je jednostavniji prevod aplikacije na različite jezike, tako što se u folder sa nazivom `values-xx`, gdje je `xx` dvoslovni kôd jezika, smiještaju stringovi na tom jeziku.

Primjer:

Engleski jezik - `/values/strings.xml`:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <string name=”title”>My Application</string>
    <string name=”hello_world”>Hello World!</string>
</resources>
```

Španski jezik - `/values-es/strings.xml`:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <string name=”title”>Mi Aplicaci&#xF3;n</string>
    <string name=”hello_world”>Hola Mundo!</string>
</resources>
```

Francuski jezik - `/values-fr/strings.xml`:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <string name=”title”>Mon Application</string>
    <string name=”hello_world”>Bonjour le monde !</string>
</resources>
```

Vrijednostima stringa unutar koda se može pristupiti s `R.string.NAZIV` (npr. `R.string.hello-world`). U zavisnosti od odabranog jezika dobit će se vrijednost stringa na navedenom jeziku ili *default*-na vrijednost ukoliko ne postoji prevod za taj jezik.

Moguće je i druge resurse podjeliti na sličan način tako da se primjenjuje jedna varijanta resursa u zavisnosti od postavke, stanja ili fizičkih karakteristika uređaja.

Promjena *layout*-a u zavisnosti od veličine *display*-a

Layout-i u `res` folderu se mogu podjeliti na način da se određeni *layout* primjenjuje na određenoj veličini *display*-a, tako da je:

- `res/layout` – *default*-ni folder za *layout*-e
- `res/layout-large` – folder za *layout*-e gdje je veći *display*

Ukoliko želimo specificirati *layout* za *landscape* orijentaciju potrebno je samo dodati `-land` u nastavku foldera:

- `res/layout-land`
- `res/layout-large-land`

Ako su dva *layout*-a sa istim imenom stavljena u dva različita foldera učitat će se onaj za kojeg je ispunjen zahtjev veličine i orijentacije *display*-a naveden u nazivu foldera, inače se učitava *layout* koji je postavljen u početni folder `res/layout`. Na primjer, ako želimo da se prikazuje jedan *layout* kada je aplikacija u *portrait* orijentaciji uređaja, a drugi ako je u *landscape* orijentaciji uređaja tada kreiramo dva *layout*-a s istim imenom i smiještamo ih u foldere `res/layout-port` i `res/layout-land` respektivno. Sadržaje navedenih *layout*-a definišemo po potrebi i oni mogu biti različiti.

Slike i gustina prikaza

Slika veličine 40px × 40px će na *display*-u s gustinom prikaza od 80ppi biti veličine 0.5”, dok će ista slika na ekranu od 200ppi biti veličine 0.02”. Kako bi omogućili vjeran prikaz slika na svim *display*-ima potrebno je obezbijediti slike različite rezolucije.

Slike možemo podijeliti u foldere koji označavaju kategorije *display*-a sa istom gustinom prikaza:

- `res/drawable` – *default*-ni folder za slike
- `res/drawable-xhdpi` – slike veoma velike rezolucije
- `res/drawable-hdpi` – velike rezolucije
- `res/drawable-mdpi` – srednje rezolucije
- `res/drawable-ldpi` – male rezolucije

Kada se računa rezolucija slike, kao referentna veličina se uzima mdpi, pa je tada:

- ldpi 0.75 puta veća od referentne
- hdpi 1.5 puta veća od referentne
- xhdpi 2 puta veća od referentne

Više detalja oko podrške uređaja sa različitim veličinama *display*-a se može pronaći na: Screen Compatibility Overview[22]

U nastavku je dat primjer u kojem su navedene datoteka slike za *default*-ni ekran i datoteka alternativne slike za ekran visoke rezolucije.

```
HelloWorld/
  app/
     java/
        ba.unsa.etf.rma.hw.helloworld
          MainActivity.kt
     res/
        drawable/
          icon.png
        drawable-hdpi/
          icon.png
        layout/
          activity_main.xml
        values/
          strings.xml
```

Tokom razvoja aplikacije potrebno je pristupiti definisanim resursima u kôdu ili u *layout*-u (XML datotekama).

Pristup resursima u kôdu

Kada se kompajlira Android aplikacija generiše se `R` klasa koja sadrži id resursa, za sve resurse koji se nalaze na `res` folderu. `R` klasa se može koristiti

[22] Screen Comphttps://developer.android.com/guide/practices/screens_support.htmlatibility Overview

za pristup resursu navođenjem pod foldera i naziva resursa, ili direktno pomoću id resursa.

Tako se, na primjer, datoteci slike `icon.png` koja se nalazi na `drawable` podfolderu `res` foldera može pristupiti s `R.drawable.icon`.

```
imageView.setImageResource(R.drawable.icon);
```

S druge stranem, u primjeru u nastavku je navedena upotreba datoteke `strings.xml` sa sljedećim sadržajem:

```
<resources>
    <string name="app_name">Hello World</string>
</resources>
```

gdje je moguće odrediti tekst za `TextView` objekat upotrebom id resursa na sljedeći način:

```
textView.setText(R.string.app_name);
```

Također, moguće je koristiti *layout* (XML datoteku) `activity_main.xml` sa sljedećim sadržajem:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World" />

    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Next" />
</LinearLayout>
```

i u glavnoj aktivnosti aplikacije učitati sadržaj u `onCreate()` metodi:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
```

```
    setContentView(R.layout.activity_main)
}
```

Pristup resursima u XML-u

U ovom slučaju će biti analizirana XML datoteka `strings.xml` sa sljedećim sadržajem:

```
<resources>
    <string name=”app_name”>Hello World</string>
    <color name=”gold”>#ffd700</color>
</resources>
```

Sada se ovi resursi mogu koristiti u *layout*-u za postavljanje stringa i boje teksta na sljedeći način:

```
<EditText xmlns:android=”http://schemas.android.com/apk/res/
android”
    android:layout_width=”fill_parent”
    android:layout_height=”fill_parent”
    android:text=”@string/app_name”
    android:textColor=”@color/gold” />
```

3.2.12. Adapter

Adapteri su komponente koje vežu podatke za `ViewGroup-e` koje su nasljeđene iz klase `AdapterView` poput `ListView-a` ili `Gallery`. Adapteri kreiraju *child* `View` elemente i pune ih sa odgovarajućim podacima [26].

Adapter *pattern* je jedan od *design pattern*-a koji ima za cilj da interfejs jedne klase pretvori u neki željeni interfejs kako bi se ona mogla koristiti u situaciji u kojoj bi inače problem predstavljali nekompatibilni interfejsi. Ovaj *pattern* postoji i u stvarnom životu. Na primjer, ako postoji kabl koji ide u utičnicu A, a na raspolaganju postoji utičnica B, da bi se uključio kabl A u utičnicu B potreban je adapter koji će dati novi interfejs kablu tako da on bude kompatibilan sa utičnicom. Adapteri navedeni u ovom poglavlju imaju sličnu svrhu. Oni daju odgovarajući interfejs podacima kako bi se oni mogli iskoristiti u `ViewGroup` objektima.

Postoje dvije često korištene izvedbe adaptera: `ArrayAdapter` i `SimpleCursorAdapter`. Opis `ArrayAdapter-a` slijedi u nastavku, dok će o `SimpleCursorAdapter-u` biti navedeno više detalja uz opis rada sa bazom podataka.

`ArrayAdapter` povezuje `AdapterView` sa nizom objekata unaprijed navedene klase. `ArrayAdapter`, ukoliko nije drugačije specificirano, poziva metodu `toString` svakog objekta iz niza i dobijenim stringom popunjava `TextView`-ove.

Ova klasa se može proširiti s vlastitom implementacijom za rad, s vlastitim klasama.

Standardno korištenje adaptera:
– Deklarisanje niza elemenata

```
ArrayList<String> myStringArray = new ArrayList<String>();
```

– Dohvaćanje ID-a *layout*-a elementa liste

```
int layoutID = android.R.layout.simple_list_item_1;
```

– Instanciranje adaptera i dodavanja odgovarajućem `ListView` objektu

```
ArrayAdapter<String> myAdapterInstance;
myAdapterInstance = new
ArrayAdapter<String>(this,layoutID,myStringArray);
myListView.setAdapter(myAdapterInstance);
```

Prilikom dodavanja novog elementa u listu potrebno je ažurirati adapter:

```
adapter.notifyDataSetChanged();
```

3.2.12.1. Proširivanje adaptera

Često je potrebno da se u listi čuvaju cijeli objekti, a ne samo njihove reprezentacije u obliku stringa. Kako bi se omogućio prikaz objekata vlastite klase potrebno je napraviti proširenje klase `ArrayAdapter`.

Proširivanje `ArrayAdapter`-a se radi na sljedeći način:
– Kreira se nova klasa koja je naslijeđena iz `ArrayAdapter` klase i odgovarajući konstruktor iste

```
class MyArrayAdapter(context: Context, @LayoutRes private val
layoutResource: Int, private val elements: List<Element>):
ArrayAdapter<Element>(context, layoutResource, elements) {
```

– *Override*-a se metoda `getView`

```
override fun getView(position: Int, newView: View?, parent:
ViewGroup):
```

```
                                                        View {
        var newView = newView
    //Vrsi se inflate layout-a odnosno kreira se odgovarajuci
    view svakog elementa na osnovu postojeceg layout-a
        newView = LayoutInflater.from(context).inflate(R.
        layout.element_list, parent, false)
    // Ovdje se moze dohvatiti reference na View
        val text = newView.findViewById(R.
        id.idTextViewElementa);
    // i popuniti ga s vrijednostima polja iz objekta
        val element = elements.get(position)
        return newView
    }
```

Metoda `getView` služi za kreiranje, *inflate* (proces kreiranja UI iz XML-a) i popunjavanje `View` objekta koji će biti dodan roditeljskom `AdapterView-u`. Ova metoda kao povratnu vrijednost ima novi `View` objekat koji je popunjen s novim vrijednostima.

Kako ova klasa kao parametar prima vrijednost pozicije elementa u listi koji se kreira/ažurira, tako je moguće dohvatiti odgovarajući objekat iz niza objekata. Dohvatanje objekta iz niza s pozicije `position` radi metoda `getItem(position)`.

Reference na elemente unutar novokreiranog/ažuriranog `View` objekta, poput `TextView-a`, se mogu dobiti pozivanjem metode `findViewById` nad `newView`.

3.2.13. Kreiranje dinamičke liste korištenjem `RecyclerView`

Za kreiranje dinamičke liste potrebne su određene klase:

- `RecyclerView` - predstavlja `ViewGroup` koja sadrži sve UI elemente koji odgovaraju podacima.
- Svaki element jedne liste je definisan korištenjem `ViewHolder` objekta. Kada se kreira `ViewHolder`, on nema povezane podatke. Nakon njegovog kreiranja, `RecyclerView` će povezati, odnosno `bind-ati` mu podatke. Definiše se korištenjem `RecyclerView.ViewHolder`.
- `RecyclerView` zahtijeva `View` i vrši povezivanje podataka s `View` pozivanjem odgovarajućih metoda iz adaptera koji proširuje `RecyclerView.Adapter`.

– `LayoutManager` definiše poziciju individualnih elemenata liste. Može se koristiti Androidov ili se može kreirati vlastiti.

Nakon kreiranja *layout*-a, potrebno je implementirati `Adapter` i `ViewHolder`. Ove dvije klase zajedno definišu kako će se podaci prikazivati. `ViewHolder` je omotač nad `View`-om koji sadrži *layout* svakog individualnog elementa liste. `Adapter` kreira `ViewHolder` objekte po potrebi i dodjeljuje vrijednosti `View`-ovima. Proces povezivanja `View`-a s podatkom se naziva `binding`.

Implementacija adaptera podrazumijeva *override* sljedećih metoda:

– `onCreateViewHolder()` - `RecyclerView` poziva ovu metodu kada god kreira novi `ViewHolder`. Ova metoda kreira i inicijalizira `ViewHolder` i njegov vezani view, ali ga ne puni podacima.
– `OnBindViewHolder() - RecyclerView` poziva ovu metodu kada veže podatke za `ViewHolder`. Ova metoda dohvati podatke i dodijeljuje ih odgovarajućem `View`-u.
– `getItemCount() - RecyclerView` poziva ovu metodu da dobije veličinu skupa podataka. Ova metoda je potrebna da se odredi kada nema više podataka.

Primjer: *layout* `text_row_item_xml` koji slijedi predstavlja izgled elementa liste:

```
<FrameLayout xmlns:android=”http://schemas.android.com/apk/res/
android”
    android:layout_width=”match_parent”
    android:layout_height=”@dimen/list_item_height”
    android:layout_marginLeft=”@dimen/margin_medium”
    android:layout_marginRight=”@dimen/margin_medium”
    android:gravity=”center_vertical”>
    <TextView
        android:id=”@+id/textView”
        android:layout_width=”wrap_content”
        android:layout_height=”wrap_content”
        android:text=”@string/element_text”/>
</FrameLayout>
```

Odgovarajući `RecyclerView` se može definisati na sljedeći način:

```
class CustomAdapter(private val dataSet: Array<String>) :
        RecyclerView.Adapter<CustomAdapter.ViewHolder>() {
    //Klasa za pruzanje referenci na sve elemente view-a
```

```
    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val textView: TextView
        init {
            // Definisanje akcija na elemente
            textView = view.findViewById(R.id.textView)
        }
    }
    // Kreiranje novog view-a
    override fun onCreateViewHolder(viewGroup: ViewGroup,
    viewType: Int): ViewHolder {
        // Kreiraj novi view koji definise UI element liste
        val view = LayoutInflater.from(viewGroup.context)
             .inflate(R.layout.text_row_item, viewGroup, false)
        return ViewHolder(view)
    }
    // Izmjena sadrzaja view-a
    override fun onBindViewHolder(viewHolder: ViewHolder,
    position: Int) {
        //Dohvatanje elementa iz skupa podataka i zamijena sadrzaja
        //View-a s odgovarajucim
        viewHolder.textView.text = dataSet[position]
    }
    // Vracanje velicine skupa
    override fun getItemCount() = dataSet.size
}
```

3.2.14. Intenti

Android ima jedinstven način komuniciranja između aplikacija (i aplikacije i operativnog sistema). To je realizovano putem mehanizma razmjene poruka koji se nazivaju intetnti (eng. *Intents*) [27]. Intetnti povezuju različite komponente u toku izvršavanja aplikacije. Jedan od najčešće korištenih primjena intetnt-a je za pokretanje novih aktivnosti. Intenti obavljaju proces pozivanja aktivnosti, ili navigacije između različitih aplikacija. Neke od primjena intent-a su:

- Slanje korisnika iz jedne aplikacije u drugu
- Dobavljanje rezultata iz neke druge aktivnosti
- Dozvoljavanje drugim aplikacijama pokretanje aktivnosti

Intent predstavlja apstraktni opis operacije koja se treba izvršiti. Može se koristiti sa:

– `startActivity` metodom za pokretanje aktivnosti
– `broadcastIntent` metodom za slanje intenta nekoj `BroadcastReceiver` komponenti
– `startService` ili `bindService` metodama za komunikaciju sa servisom

Postoje odgovarajuće metode za dostavljanje intenta svakom tipu komponente aplikacije, kao što su aktivnosti, servisi i prijemnici obavijesti. Te metode su:

– `Context.startActivity()` - intent objekat je proslijeđen ovoj metodi za pokretanje nove aktivnosti ili dobijanje postojeće aktivnosti koja će uraditi nešto novo
– `Context.startService()` - intent objekat je proslijeđen ovoj metodi za pokretanje servisa ili prenos nove instrukcije za servis koji je u toku izvršenja
– `Context.sendBroadcast()` - intent objekat je proslijeđen ovoj metodi za prenos poruke svim zainteresovanim prijemnicima obavijesti

gdje se `Context` odnosi na trenutno stanje aplikacije i okruženje. Intent objekat je kolekcija informacija koju koristi komponenta Android sistema. U ovisnosti o komunikaciji i akciji za koju se koristi, intent objekat može da sadrži sljedeće komponente:

Akcija – je obavezni dio intent objekta tipa `String` i odnosi se na naziv akcije koja će biti izvršena. Standardne akcije Android intenta su:

`ACTION_ALL_APPS` – lista sve dostupne aplikacije na uređaju

`ACTION_ANSWER` – upravlja dolaznim telefonskim pozivom

`ACTION_ATTACH_DATA` – naznačava da bi neki dio podataka trebao biti uključen na neko drugo mjesto

`ACTION_BATTERY_CHANGED` – obavijest koja sadrži informacije o bateriji

`ACTION_BATTERY_LOW` – obavijest koja se odnosi na upozorenje o niskom nivou punjenja baterije

`ACTION_BATTERY_OKAY` – obavijest da je baterija uredu

`ACTION_BOOT_COMPLETED` – obavijest da je sistem završio proces pokretanja (*boot*) i da je spreman za rad

`ACTION_BUG_REPORT` – prikazuje aktivnost za izvještavanje o kvaru

`ACTION_CALL` – izvršava poziv nekoga ko je naznačen u podacima

`ACTION_CALL_BUTTON` – aktiviranje dumića za poziv, nakon čega je omogućeno biranje poziva

`ACTION_CAMERA_BUTTON` – aktiviranje dugmića kamere

`ACTION_CHOOSER` – prikaz izbornih aktivnosti

`ACTION_CONFIGURATION_CHANGED` – promjena konfiguracije uređaja

`ACTION_DATE_CHANGED` – obavijest da je promijenjen datum

`ACTION_DEFAULT` – sinonim za `ACTION_VIEW` (prikazuje podatke korisniku)

`ACTION_DELETE` – briše određene podatke iz kontejnera

`ACTION_DEVICE_STORAGE_LOW` – obavijest koja ukazuje na slabu memoriju na uređaju

`ACTION_DEVICE_STORAGE_OK` – ukazuje da je memorija na uređaju uredu

`ACTION_DIAL` – bira navedeni broj

`ACTION_DOCK_EVENT` – obavijest o promjenama fizičkog stanja uređaja

`ACTION_EDIT` – omogućava pristup za editovanje dobijenih podataka

`ACTION_GET_CONTENT` – omogućava korisniku da odabere određenu vrstu podataka i da ih vrati

`ACTION_HEADSET_PLUG` – slušalice su uključene ili isključene

`ACTION_INPUT_METHOD_CHANGED` – ulazna metoda je promijenjena

`ACTION_INSERT` – ubacivanje prazne stavke u dati kontejner

`ACTION_INSERT_OR_EDIT` – izbor postojeće stavke ili ubacivanje nove, a zatim njeno editovanje

`ACTION_INSTALL_PACKAGE` – pokretanje *installer*-a aplikacije

`ACTION_LOCALE_CHANGED` – mjesto (*locale*) uređaja je promijenjeno

`ACTION_MEDIA_BUTTON` – aktivira dugme medija

`ACTION_MEDIA_CHECKING` – provjera vanjskog medija

`ACTION_MEDIA_EJECT` – zahtjev korisnika za uklanjanjem vanjskog medija

`ACTION_MEDIA_REMOVED` – vanjski medij je izbrisan

`ACTION_NEW_OUTGOING_CALL` – novi odlazni poziv

`ACTION_POWER_CONNECTED` – uređaj je priključen na napajanje

`ACTION_REBOOT` – sistem je ponovo pokrenut

`ACTION_RUN` – pokretanje podataka

`ACTION_SCREEN_OFF` – slanje nakon što je ekran isključen

`ACTION_SCREEN_ON` – slanje nakon što je ekran uključen

`ACTION_SEARCH` – pretraga

`ACTION_SEND` – slanje podataka

`ACTION_SENDTO` – slanje poruke naznačenom primaocu

`ACTION_SEND_MULTIPLE` – slanje više podataka

`ACTION_SET_WALLPAPER` – prikaz postavki za izbor pozadine

`ACTION_SHUTDOWN` – isključenje uređaja

`ACTION_SYNC` – sinhronizacija podatka

`ACTION_TIMEZONE_CHANGED` – vremenska zona je promijenjena

`ACTION_TIME_CHANGED` – vrijeme je promijenjeno

`ACTION_VIEW` – prikazuje podatke korisniku

`ACTION_VOICE_COMMAND` – pokretanje komande glasa

`ACTION_WALLPAPER_CHANGED` – promjena pozadine (tekućeg) sistema

`ACTION_WEB_SEARCH` – web pretraživanje

Akcija u intent objektu se može postaviti pomoću `setAction()` metode i pročitati pomoću `getAction()` metode.

Podatak – je podatak s kojim se radi, kao što je određeni slog u bazi podataka, prikazan kao *Uniform Resource Identifier* (*URI*). Na primjer, ako je akcija `ACTION_EDIT`, tada polje podatka sadrži URI dokumenta koji će biti prikazan za editovanje.

U nastavku slijede primjeri parova akcija/podatak:

- `ACTION_VIEW content://contacts/people/1` – prikazuje informacije osobe čiji je identifikator "1"
- `ACTION_EDIT content://contacts/people/1` – editovanje informacija osobe čiji je identifikator "1"
- `ACTION_VIEW content://contacts/people/` – prikazuje listu osoba koje korisnik može pregledati

Metode za rad s podacima su: `setData()` koja specificira podatak kao URI, `setType()` specificira podatak kao MIME tip, a `setDataAndType()` specificira podatak kao URI i MIME tip. URI se može dobiti s `getData()`, a tip s `getType()` metodom.

Kategorija – je opcionalni dio intent objekta tipa `String` koja daje dodatne informacije o akciji koja će biti izvršena. Ona sadrži dodatne informacije o komponenti koja bi trebala da upravlja intentom.

Neke od standardnih kategorija Android intenta su:

`CATEGORY_APP_BROWSER` – koristi se s `ACTION_MAIN` za pokretanje aplikacije pretraživača

`CATEGORY_APP_CALCULATOR` – koristi se s `ACTION_MAIN` za pokretanje aplikacije kalkulatora

`CATEGORY_APP_CALENDAR` – koristi se s `ACTION_MAIN` za pokretanje aplikacije kalendara

`CATEGORY_APP_EMAIL` – koristi se s `ACTION_MAIN` za pokretanje email aplikacije

`CATEGORY_APP_MESSAGING` – koristi se s `ACTION_MAIN` za pokretanje *messaging* aplikacije

`CATEGORY_APP_MUSIC` – koristi se s `ACTION_MAIN` za pokretanje aplikacije za muziku

`CATEGORY_CAR_MODE` – koristi se da ukaže da se aktivnost može koristiti u autu

Metode za rad s kategorijama su: `addCategory()` koja dodaje kategoriju u intent objekat, `removeCategory()` briše ranije dodanu kategoriju, dok metoda `getCategories()` dohvata sve kategorije koje su trenutno u objektu.

Tip – je opcionalni dio intent objekta kojim se navodi tip podatka intenta.

Komponenta – je opcionalno polje koje predstavlja klasu aktivnosti, servisa ili prijemnika obavijesti.

Metode za rad s komponentama su: `setComponent()` kojom se specificora komponenta, te `getComponent()` za čitanje komponente.

Dodatak – je predstavljen parovima ključ-vrijednost kao dodatna informacija koje se treba proslijediti komponenti koja upravlja intentom. Njegova

vrijednost se može zadati pomoću `putExtras()` metode i čitati pomoću `getExtras()` metode.

Neki od standardnih dodatnih podataka Android intenta su:

`EXTRA_BCC` – predstavlja `String[]` koji sadrži e-mail adresu koja bi trebala biti nevidljiva primaocu (*blind carbon copy* - Bcc)

`EXTRA_CC` – `String[]` koji sadrži e-mail adresu koja bi trebala dobiti kopiju maila (*carbon copy* – Cc)

`EXTRA_EMAIL` – `String[]` koji sadrži e-mail adresu na koju bi se trebalo poslati mail

`EXTRA_SUBJECT` – sadrži temu (*subject*) poruke

`EXTRA_SHORTCUT_ICON` – definiše ikonu *shortcut*-a

`EXTRA_SHORTCUT_INTENT` – definiše intent *shortcut*-a

`EXTRA_SHORTCUT_NAME` – definiše naziv *shortcut*-a

Android pruža dva načina za korisnike da dijele podatke između aplikacija:

- Android *Sharesheet* je prvenstveno dizajniran za slanje sadržaja aplikacije i/ili direktno drugom korisniku. Na primjer, dijeljenje URL-a sa prijateljem.
- Android *intent resolver* je najprikladniji za prosljeđivanje podataka u slijedeću fazu dobro definisanog zadatka. Na primjer, otvaranje PDF-a iz aplikacije i dopuštanje korisnicima da izaberu željeni preglednik.

Intenti se mogu koristiti da se pošalje poruka kroz cijeli sistem, a ostale aplikacije tu poruku osluškuju i analiziraju da li one mogu odgovoriti na nju. Na ovaj način se skup svih aplikacija transformiše iz kolekcije nepovezanih, nezavisnih aplikacija i komponenti u jedan sistem povezanih komponenti.

Android operativni sistem takođe može slati intente u slučajevima kada se desi neki sistemski događaj, poput: poziva, SMS poruke, prazne baterije i sl. Jedan od osnovnih principa Androida je da se intenti koriste za obavještavanje aplikacija i propagaciju akcija kroz druge aplikacije bez da se svaka aplikacija eksplicitno poziva. Ovo omogućava da se aplikacija na jednostavan način nadopunjuje i da se njene funkcionalnosti proširuju. Ovaj način razvoja forsira razdvajanje komponenti i čini ih nezavisnim od drugih, što omogućava eventualnu zamjenu nekih komponenti, bez da je potrebno mijenati druge.

`Intent`-om se može zahtijevati da se neka akcija obavi nad podacima:

– uz specificiranje tačne klase (aktivnosti) koja će se učitati - Eksplicitni intent
– bez specificiranja tačne aktivnosti - Implicitni intent

3.2.14.1. Eksplicitni intent

Eksplicitni intent specificira komponentu koja se treba pozvati. U `Intent` objektu se definiše klasa koja će biti pozvana. Upotrebom eksternog intenta moguće je jednu aktivnost povezati s drugom. Ovi intenti određuju ciljnu komponentu pomoću njenog imena. Obično se koriste za interne poruke aplikacije. Dakle, u slučaju da je poznata tačna klasa aktivnosti koja se treba pokrenuti intent-om, tada se intent kreira navođenjem trenutne aktivnosti i klase aktivnosti koja se treba pokrenuti.

Primjer:

Prvi parametar u konstruktoru je objekat tipa `Context`. `Context` govori o trenutnom stanju aplikacije i samog okruženja. Drugi parametar je specificirana klasa aktivnosti koja se treba pokrenuti. Pozivanje nove aktivnosti se radi pomoću `startActivity` metode gdje se kao parametar navodi novokreirani intent.

```
// Eksplicitni intent – navođenje naziva klase
val intent = Intent(this, SecondAcitivity::class.java) {
}

// Pokretanje ciljne aktivnosti
startActivity(intent);
```

Novoj aktivnosti se mogu poslati dodatni podaci s `putExtra(kljuc,vrijednost)` metodom `Intent` klase, kojoj se navedu parametri u formi para ključ-vrijednost. Ovo je potrebno uraditi prije startanja aktivnosti.

Primjer:

```
// Eksplicitni intent – navođenje naziva klase
val intent = Intent(this, SecondAcitivity::class.java) {
      intent.putExtra(“k1”, “abc”)
      intent.putExtra(“k2”, “123”)
}

// Pokretanje ciljne aktivnosti
startActivity(intent);
```

3.2.14.2. Implicitni intent

Ponekad je potrebno specificirati samo akciju, a druga aplikacije se treba pobrinuti za učitavanje odgovarajuće klase. Takvi intent-i se nazivaju implicitni intent-i, jer se nova aktivnost ne navodi, već druga aplikacija treba prepoznayi da se intent odnosi na nju.

Najčešće se koristi Android *Sharesheet* za prikaz liste aplikacija i potencijalnih korisnika kojima se može podijeliti intent. Android *Sharesheet* se poziva korištenjem `Intent.createChooser()` metode kojoj se proslijedi odgovarajući `Intent` objekat.

Slanje tekstualnog sadržaja

Najjednostavnija i najčešća upotreba Android *Sharesheeta* je slanje tekstualnog sadržaja iz jedne aktivnosti u drugu. Na primjer, većina pretraživača može dijeliti URL trenutno prikazane stranice kao tekst s drugom aplikacijom. Ovo je korisno za dijeljenje članka ili web stranice s prijateljima putem e-pošte ili društvenih mreža.

Primjer: Ako želimo da kroz aplikaciju podijelimo nešto s drugim korisnicima, a ne želimo implementirati proces dijeljenja, već tražimo od korisnika da odabere neku od postojećih aplikacija da obavi slanje podataka umjesto nas, tada koristimo implicitne intent-ove. Primjer kreiranja i pozivanja implicitnog intent-a je:

```
// Kreiranje tekstualne poruke
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, textMessage)
    type = “text/plain”
}
val shareIntent = Intent.createChooser(sendIntent, null)
startActivity(shareIntent)
```

Prilikom kreiranja implicitnog intent-a navodi se (neka od ranije opisanih) akcija koju je potrebno izvršiti. Nakon specificiranja akcije navode se podaci koji se prosljeđuju uz akciju, u ovom slučaju to je tekst poruke.

U slučaju da nije sigurno da li postoji aplikacija koja može odgovoriti na intent, potrebno je provjeriti da li postoji aplikacija koja može odgovoriti na njega. Ukoliko postoji jedna aplikacija koja odgovara na navedenu akciju, tada se ona automatski poziva. Ako postoji više takvih aplikacija, tada se daje korisniku na izbor putem sistemskog dijaloga. U slučaju da ne

postoji aplikacija koja odgovara na navedenu akciju, aplikacija će krahirati, pa zbog toga uvijek treba izvršavati start unutar try-catch bloka.

Primjer otvaranja web stranice:

```
// Kreiranje tekstualne poruke
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, textMessage)
type = “text/plain”
}
// Try-catch block za pozivanje aktivnosti
try {
    startActivity(sendIntent)
} catch (e: ActivityNotFoundException) {
    // Definisati naredbe ako ne postoji aplikacija za navedenu
    akciju
}
```

3.2.14.3. Intent filteri

Android koristi filtere kako bi odredio skup aktivnosti, servisa i prijemnika obavijesti koji mogu upravljati intentom, a uz pomoć skupa akcija, kategorija i podataka vezanih za intent [27]. U manifest datoteci je potrebno uključiti `<intent-filter>` tag navodeći akcije, kategorije i podatke koji su povezani sa aktivnostima, servisima i prijemnicima obavijesti.

Primjer:

U ovom primjeru manifest datoteke `AndroidManifest.xml` je navedena aktivnost `ba.unsa.etf.rma.hw.CustomActivity` koju mogu pozvati dvije akcije, kategorija i podatak. Kako je aktivnost naznačena unutar filtera, nju mogu pozivati druge aktivnosti koristeći `android.intent.action.VIEW` ili `ba.unsa.etf.rma.hw.LAUNCH` akciju obezbjeđujući kategoriju `android.intent.category.DEFAULT`. Element podatka određuje tip podatka kojeg aktivnost može pozvati, tako da u navedenom primjeru aktivnost očekuje da podatak počinje s “`http://`”.

```
<activity android:name=”.CustomActivity”
    android:label=”@string/app_name”>
    <intent-filter>
        <action android:name=”android.intent.action.VIEW” />
        <action
        android:name=”ba.unsa.etf.rma.hw.LAUNCH” />
```

```
        <category android:name=”android.intent.category.DEFAULT” />
        <data android:scheme=”http” />
    </intent-filter>
</activity>
```

Dakle, koristeći `intent-filter` tag i njegove atribute unutar manifest datoteke moguće je navesti na koje vrste akcija aplikacija odgovara. Tako putem sljedećih tagova unutar `intent-filter` tag-a se navodi:

- `action` - naziv akcije intent-a na koju aplikacija ima odgovor. Svaki intent-filter mora imati bar jedan `action` tag.
- `category` - ovim tag-om se specificira pod kojim uslovima će aplikacija reagovati na navadenu akciju. Na primjer, ako se želi da aplikacija reaguje na akcije pozvane iz pretraživača (eng. *browser*) tada se koristi kategorija BROWSABLE, i sl.
- `data` - ovim tag-om se navodi koje tipove podataka aplikacija podržava

Više detalja o atributima i tag-ovima koji se mogu navesti unutar `intent-filter` tag-a moguće je pronaći na: intent-filter[23]

Kada pokušava da nađe odgovor za implicitni intent:
- Android sastavlja listu intent filtera svih aplikacija
- Aplikacije, čiji intent filteri ne odgovaraju na akciju pozvanog intent-a ili imaju drugačije specificiranu kategoriju ili tip podataka koje intent šalje, se odbacuju
- Nakon svih završenih provjera ukoliko je ostala samo jedna podobna aplikacija, tada se ona pokreće. Ako ima više aplikacija koje odgovaraju, one se nude korisniku kako bi odabrao aplikaciju koju želi.

U slučaju da je aplikacija prošla provjeru i korisnik ju je odabrao, potrebno je pokupiti podatke iz intent-a. U većini slučajeva se može u `onCreate()` metodi početne aktivnosti aplikacije dohvatiti referenca na pozivajući intent koristeći `getIntent()` metodu. Nad dobijenom referencom se mogu dalje obraditi podaci na osnovu `action` i `data` elemenata. Za dobijanje tipa akcije intent-a i podataka koriste se `getAction()` i `getData()` metode nad dobijenom referencom.

Primjeri odgovaranja na intent:

U slučaju da aplikacija prima velike količine podataka treba omogućiti dobavljanje tih podataka na sporednim nitima.

[23] https://developer.android.com/guide/topics/manifest/intent-filter-element.html

```
override fun onCreate(savedInstanceState: Bundle?) {
    …
    when {
        intent?.action == Intent.ACTION_SEND -> {
            if (“text/plain” == intent.type) {
                handleSendText(intent) // Handle text being sent
            } else if (intent.type?.startsWith(“image/”) == true) {
                handleSendImage(intent) // Handle single image
                being sent
            }
        }
        intent?.action == Intent.ACTION_SEND_MULTIPLE &&
            intent.type?.startsWith(“image/”) == true -> {
            handleSendMultipleImages(intent) // Handle multiple
            images being sent
        } else -> {
            // Rukovati drugim intent-ima, kao sto je pokretanje
            s pocetnog ekrana
        }
    }
    …
}

private fun handleSendText(intent: Intent) {
    intent.getStringExtra(Intent.EXTRA_TEXT)?.let {
        // Azuriranje UI da odrazava tekst koji se dijeli
    }
}
private fun handleSendImage(intent: Intent) {
    (intent.getParcelableExtra(Intent.EXTRA_STREAM) as? Uri)?.let {
        // Azuriranje UI da odrazava sliku koja se dijeli
    }
}
private fun handleSendMultipleImages(intent: Intent) {
    intent.getParcelableArrayListExtra(Intent.EXTRA_STREAM)?.let {
    // Azuriranje UI da odrazava vize slika koje se dijele
    }
}
```

Ukoliko se podesi aktivnost da postane aktivna, ako je bila u pozadini, i da primi intent u tom slučaju se intent ne može obraditi samo u `onCreate()` metodi, već je potrebno implementirati metodu `onNewIntent()` i unutar nje

obraditi intent. U većini slučajeva dovoljno je samo implementirati obradu intent-a u `onCreate()` metodi.

3.2.15. Prijemnici obavijesti

Komunikacija između Android operativnog sistema i aplikacija funkcioniše tako da sistem emituje (eng. *broadcast*) razne obavijesti koje određene aplikacije mogu osluškivati i odgovoriti na njih nekom akcijom. *Broadcast* obavijesti su sistemski ili aplikacijski događaji [28]. Svaka *broadcast* poruka se emituje kao `Intent` objekat. Prijemnici obavijesti (eng. b*roadcast receiver*) se koriste za odgovor na te poruke. Oni omogućavaju registrovanje na sistemske i aplikacijske događaje. Kada se desi neki događaj, registrovani prijemnici budu obavješteni, te zatim mogu izvršiti određenu akciju vezanu za taj događaj. Tako se mogu kreirati aplikacije koje emituju obavijesti na koje neka druga aplikacija može odgovoriti.

Ukoliko je potrebno da aplikacija osluškuje prijemnike obavijesti, koje šalje sistem ili neka druga aplikacija, potrebno je implementirati vlastitu klasu *broadcast receiver*-a koja je naslijeđena iz klase `BroadcastReceiver`. Kreiranje `BroadcastReceiver`-a je prilično jednostavno. Potrebno je nakon deklarisanja klase izvršiti `override` metode `onReceive()`. Ova metoda se poziva ukoliko se neka *broadcast* obavijest desila.

```
private const val TAG = "MyBroadcastReceiver"
class MyBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        StringBuilder().apply {
            append("Action: ${intent.action}\n")
            append("URI: ${intent.toUri(Intent.URI_INTENT_
            SCHEME)}\n")
            toString().also { log →
                Log.d(TAG, log)
                Toast.makeText(context, log, Toast.LENGTH_LONG).
                show()
            }
        }
    }
}
```

Nakon implementiranja funkcionalnosti `BroadcastReceiver`-a potrebno je dodati permisije koje on zahtjeva u manifest datoteci. Primjer permisije koja se zahtijeva (stanje umreženosti) je dat u nastavku:

```
android.permission.ACCESS_NETWORK_STATE
```

Postoje dvije vrste prijemnika poruka:
- Statički prijemnici - koji su deklarisani u manifest datoteci i rade kada je aplikacija zatvorena
- Dinamički prijemnici - koji rade samo ako je aplikacija aktivna ili miniminizirana, i koji se registruju (s `registerReceiver()`) i odjavljuju (s `unregisterReceiver()`) za vrijeme izvršenja

Ako je potrebno da se obavijesti stalno osluškuju (čak i kada aplikacija nije pokrenuta) potrebno je u manifest datoteci registrovati `BroadcastReceiver`. To se postiže korištenjem `<receiver>` tag-a unutar `<application>` tag-a. U samom tag-u se navodi ime klase implementiranog *receiver*-a. Unutar *receiver* tag-a se definišu tag-ovi intent akcija na koje *receiver* odgovara.

Primjer registracije `BroadcastReceiver`-a unutar taga:

```
<receiver android:name=".MyBroadcastReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_
        COMPLETED"/>
        <action android:name="android.intent.action.INPUT_
        METHOD_CHANGED"/>
    </intent-filter>
</receiver>
```

Unutar *receiver*-a nije poželjno da se izvršavaju bilo kakve komplikovane operacije. On treba da služi kao osluškivač i da prosljeđuje poruku aplikaciji, a ona treba da zna kako da odgovori na nju.

Primjer registracije `BroadcastReceiver`-a unutar kôda:

```
val br: BroadcastReceiver = MyBroadcastReceiver()
val filter = IntentFilter(ConnectivityManager.CONNECTIVITY_
ACTION).apply {
    addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED)
}
//Registruj
registerReceiver(br, filter)
//Odjavi
unregisterReceiver(receiver);
```

Potrebno je voditi računa o lokaciji registrovanja *receiver*-a. Ako se registrujete u `onCreate()`, tada je odjavu potrebno uraditi u `onDestroy()`. U

slučaju da se registrujete u `onResume()`, onda se odjava vrši u `onPause()`. Ako se koristite `onResume()` i `onPause()`, tada `BroadcastReceiver` osluškuje obavijesti samo dok je aplikacija pokrenuta.

3.2.16. Testiranje mobilne aplikacije

Testiranje mobilne aplikacije podrazumijeva validaciju funkcionalnosti i upotrebljivosti aplikacije prije nego što ona bude isporučena krajnjem korisniku i stavljena u produkciju, a u svrhu provjere da li zadovoljava očekivane tehničke i druge postavljene zahtjeve.

3.2.16.1. Instrumented testiranje

Pored unit testova koji se mogu pisati kao za bilo koju drugu aplikaciju, kod razvoja mobilnih aplikacija bitni su i instrumented testovi. To su testovi koji se pokreću na uređaju ili emulatoru. Njima se može automatizovati korisnička interakcija. Instrumented testovi za Android aplikacije se pakuju u izvršni apk koji nije isti kao apk za aplikaciju koja se pokrećete na standardni način. Postoje posebi gradle zadaci koji prave odgovarajući apk za instrumented testove i pokreću ih na povezanim uređajima ili emulatorima. Ovi zadaci su ugrađeni i unaprijed konfigurisani kada se radi u AndroidStudio okruženju.

Instrumented testovi se pišu unutar foldera `[moduleName]/src/androidTest/java/`.

Za pisanje instrumented testova i njihov opis, u ovoj knjizi će biti korišten Espresso framework. Espresso testovima se može provjeriti stanje komponenti aplikacije, simulirati interakcije i napraviti različite testne provjere.

U nastavku slijedi primjer jednog espresso testa. Bit će kreirana jedostavna aktivnost sa `TextView` elementom i dugmetom. Klik na dugme mijenja sadržaj `TextView` elementa iz "prije" u "poslije". U testu će se provjeriti da li je tekst ispravan prije klika na drugme i nakon klika.

Prvo je potrebno dodati *dependency*-e koji nedostaju (oznaka + označava najnoviju verziju), kao na primjer:

```
androidTestImplementation 'androidx.test.ext:junit:+'
androidTestImplementation 'androidx.test:core-ktx:+'
androidTestImplementation 'androidx.test.ext:junit-ktx:+'
testImplementation("org.hamcrest:hamcrest:+")
androidTestImplementation 'androidx.test.espresso:espresso-
```

```
core:+’
androidTestImplementation ‘androidx.test.espresso:espresso-
intents:+’
```

Zatim se kreira jednostavni primjer i doda klasa `PrviTest` u `app/src/androidTest/java/com/`

```
package com.example.instrumentedprimjer1

import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.click
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.espresso.matcher.ViewMatchers.withText
import androidx.test.ext.junit.rules.ActivityScenarioRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class PrviTest {

    @get:Rule
    var activityRule: ActivityScenarioRule<MainActivity> =
ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun prijeKlika(){
        onView(withId(R.id.tekst)).
check(matches(withText(“prije”)))
    }

    @Test
    fun nakonKlika(){
        onView(withId(R.id.dugme)).perform(click())
        onView(withId(R.id.tekst)).
check(matches(withText(“poslije”)))
    }
}
```

U primjeru je navedena klasa `ActivityScenarioRule`. Ova klasa služi za pokretanje aktivnosti koja se testira na početku testa i zatvaranje kada se test

završi. U ovom slučaju se testira `MainActivity` tako da je navedena ta aktivnost u definiciji testnog pravila. Ovo pravilo iznad sebe treba imati anotaciju `@get:Rule`. Testovi se pišu kao metode testne klase sa anotacijom `@Test` iznad definicije funkcije.

Da bi se pristupilo nekom `View` elementu u Espresso testu koristi se `onView` metoda. Ova metoda prima kao parametar klasu koja implementira `Matcher<View>` interfejs, pa se ove klase zovu *view matcher*-i. Neki od najčešće korištenih *view matcher*-a su:

- `withId(id:Int)` - vraća *true* ako `View` ima proslijeđeni id
- `withText(text:String)` - vraća *true* ako `TextView` ima tekst koji odgovara proslijeđenom tekstu
- `withSubstring(text:String)` - vraća *true* ako `TextView` sadrži proslijeđeni podstring
- `isFocused()` - vraća *true* ako je `View` fokusiran

Ostali *view matcher*-i se mogu pronaći na: Espresso cheat sheet[24]

`onView` metoda nakon što pronađe `View` koji ispunjava uslov(e) *view matcher*-a vraća `ViewInteraction` objekat. Nad ovim objektom je moguće vršiti provjere stanja `View` elementa s metodom `check()` ili izvršiti simulaciju korisničke akcije sa `perform()` metodom.

Metoda `check()` kao parametar prima klase koje implementiraju `ViewAssertion` interfejs. Najčešće korištene `ViewAssertion` metode su:

- `matches(M)` - metoda koja prima *view matcher* M i provjerava da li je on ispunjen za odabrani `View` element
- `doesNotExist()` - metoda koja vraća *true* ako odabrani `View` ne postoji
- metode `isLeftOf(M),isRightOf(M),isAbove(M),isBelow(M)` - provjeravaju pozicije odabranog `View` elementa i onog koji je selektovan *view matcher*-om M

Ostale metode se mogu pronaći na: Espresso cheat sheet

Metoda `perform()` kao parametar prima klase koje implementiraju `ViewAction` interfejs. Najčešće korištene `ViewAction` metode su:

- `click()` - simulira klik nad odabranim `View` objektom
- `typeText(tekst:String)` - simulira upisivanje teksta u tekstualno polje

[24] https://developer.android.com/training/testing/espresso/cheat-sheet

– `replaceText(tekst:String)` - simulira brisanje starog teksta iz tekstualnog polja i upis novog teksta proslijeđenog kao parametar

Ostale metode se mogu pronaći na: Espresso cheat sheet

U primjeru iznad su obje vrste interakcija sa `View` objektima. Sa `check()` se provjerava tekst prije i poslije klika na dugme, a sa `perform()` simulira klik na dugme.

Za testiranje intenta koristi se espresso `Intents` klasa. Na početku testa koji koristi intente se poziva `Intents.init()`, a na kraju se poziva `Intents.release()`.

Obrada intenta u testovima se radi sa `intended()` i `intending()` metodama. Ove metode primaju intent *matcher*-e koji imaju sličnu ulogu kao *view matcher*-i, ali se odnose na `Intent` komponente. Razlika između dvije metode je što se `intending()` koristi za stub testove kada se simulira odgovor od intenta, dok se `intended()` koristi za provjeru intenta koje je aplikacija inicirala.

Primjer testa s `indended()` metodom:

U prethodni primjer se dodaje druga aktivnost `SecondActivity` i dugme na početnoj aktivnosti koje otvara drugu aktivnost. Nakon toga je potrebno dodati test koji će provjeriti da li se otvara aktivnost nakon klika dugmeta:

```
@Test
fun prebaciNaDruguAktivnost(){
    Intents.init()
    onView(withId(R.id.dugmeAktivnost)).perform(click())
    intended(hasComponent(SecondActivity::class.java.name))
    Intents.release()
}
```

Metoda `intended()` može primiti različite intent *matcher*-e, a najčešće korišteni su:

– `hasComponent(name:String)` - vraća *true* ako intent poziva klasu sa nazivom koji je proslijeđen kao parametar
– `hasExtra(key:String,val:T)` - vraća *true* ako intent ima extra podatke gdje je vrijednost ključa `key`, a vrijednost podatka tipa `T` je `val`
– `hasAction(action:String)` - vraća *true* ako intent poziva akciju sa nazivom koji odgovara proslijeđenom parametru

– `hasExtraWithKey(key:String)` - vraća *true* ako intent ima extra podatke koji sadrže proslijeđeni ključ `key`

U primjeru je iskorišten `hasComponent` *intent matcher* da bi se provjerilo da li je intent otvorio željenu aktivnost.

Intending, onData i RecyclerView testiranje

U okviru espresso instrumented testiranja je opisano na koji način je moguće testirati stanja korisničkog interfejsa, simulirati korisničke interakcije i testirati intente koje aktivnost pokrene. U nastavku će biti navedeni dodatni primjeri i scenariji koji su česti kod instrumented testiranja.

Intending testiranje

Ukoliko je potrebno simulirati odgovor nakon poziva intenta korištenjem `ActivityResultLauncher` tada se odabere taj intent u `intending()` metodi prosljeđujući joj odgovarajući intent *matcher* i nad rezultatom se pozove `respondWith()` metoda koja prima kao parametar simulirani odgovor od aktivnosti.

Na primjer, ako unutar aplikacije postoji intent koji otvara galeriju i traži od korisnika da odabere jednu sliku koja se poslije obrađuje u aktivnosti, tada se u testu može iskoristiti `intending(hasAction(Intent.ACTION_GET_CONTENT)).respondWith(result)`. Ovdje je `result` pripremljeni odgovor kao da je korisnik odabrao neku sliku, nakon čega se može provjeriti da li je aplikacija ispravno obradila odgovor.

Primjer – aplikacija otvara sliku iz galerije i prikazuje je u `ImageView` elementu:

```
@RunWith(AndroidJUnit4::class)
class VjezbeTest {

    @get:Rule
    val rule:ActivityScenarioRule<MainActivity> =
ActivityScenarioRule<MainActivity>(MainActivity::class.java)

    @Test
    fun intendingTest() {
        Intents.init()
        //Stub intenta Intent.ACTION_PICK, dodavanje slike u
folder Download na uredjaju i vracanje njenog URI-a
        val folder = ApplicationProvider.
```

```
getApplicationContext<Context>().
getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS)
        val testFile: File = File(folder, “testSlika.jpg”)
        val istr: InputStream = ApplicationProvider.
getApplicationContext<Context>().resources.openRawResource(R.
drawable.robot)
        istr.copyTo(testFile.outputStream())
        val uri = Uri.fromFile(testFile)
        val intent = Intent()
        intent.data = uri
        val result = Instrumentation.ActivityResult(Activity.
RESULT_OK, intent)
        intending(hasAction(Intent.ACTION_GET_CONTENT)).
respondWith(result)

        onView(withId(R.id.odaberiDugme)).perform(click())
        onView(withId(R.id.odabranaSlika)).
check(matches(withImage(R.drawable.robot)))
        Intents.release()
    }
}
```

Navedeni test prvo priprema simulaciju korisničkog odabira slike i, kada se u aktivnosti pozove intent za odabir slike, automatski se u testu vraća URI slike koju smo dodali u testu. Slika se nalazi na uređaju u folderu `Download`. Da bi se mogle dodavati slike iz kôda potrebno je u `AndroidManifest.xml` datoteci dodati sljedeće permisije:

```
<uses-permission android:name=”android.permission.WRITE_
EXTERNAL_STORAGE” />
<uses-permission android:name=”android.permission.READ_EXTERNAL_
STORAGE” />
```

Ovim permisijama je omogućeno aplikaciji da upisuje i čita iz externe memorije uređaja. Nakon što je slika prebačena na uređaj i pripremljen automatski odgovor na intent, simulira se klik na dugme. Nakon ovoga aplikacija inicira intent na koji odmah ima odgovor. U posljednjoj liniji testa se vrši provjera da li je ispravna slika dodana u `ImageView` element. Kako prilikom učitavanja i snimanja slike u datoteku može doći do različitih artefakata zbog kompresije, poređenje se treba uraditi procentualno, piksel po piksel:

```
class BitmapUtil {
    companion object {
        fun pixelDiff(rgb1: Int, rgb2: Int): Int {
            val r1 = rgb1 shr 16 and 0xff
            val r2 = rgb2 shr 16 and 0xff
            return Math.abs(r1 - r2)
        }

        fun razlikaBitmapa(b1: Bitmap, b2: Bitmap): Float {
            val pikseli1 = IntArray(b1.width * b1.height)
            val pikseli2 = IntArray(b2.width * b2.height)
            b1.getPixels(pikseli1, 0, b1.width, 0, 0, b1.width,
            b1.height)
            b2.getPixels(pikseli2, 0, b2.width, 0, 0, b2.width,
            b2.height)
            if (pikseli1.size != pikseli2.size) return 1.0f
            var razlika: Long = 0
            for (i in pikseli1.indices) {
                razlika += pixelDiff(pikseli1[i], pikseli2[i]).
                toLong()
            }
            return razlika.toFloat() / (255 * pikseli1.size).
            toFloat()
        }
    }
}
```

Navedene funkcije se mogu iskoristiti kako bi se napravio odgovarajući *matcher* za `ImageView` element:

```
fun withImage(@DrawableRes id: Int) = object :
TypeSafeMatcher<View>() {
    override fun describeTo(description: Description) {
        description.appendText("Drawable does not contain image
        with id: $id")
    }

    override fun matchesSafely(item: View): Boolean {
        val context: Context = item.context
        var bitmap: Bitmap? = context.getDrawable(id)?.toBitmap()
        if (item !is ImageView) return false
        val origBitmap = item.drawable.toBitmap()
        bitmap = bitmap!!.scale(origBitmap.width, origBitmap.
```

```
        height)
        return BitmapUtil.razlikaBitmapa(origBitmap, bitmap) < 0.01
    }
}
```

Ukoliko je razlika slika manja od 1% tada se smatra da su slike iste.

Kôd aktivnosti koja se testira je:

```
    val dugme = findViewById<Button>(R.id.odaberiDugme)

    val launcher: ActivityResultLauncher<Intent> =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) {
        if (it.resultCode == Activity.RESULT_OK && it.data !=
        null) {
            val data: Intent? = it.data
            val inputStream: InputStream? =
                data?.data?.let { it1 ->
                    applicationContext.contentResolver.
                    openInputStream(
                        it1
                    )
                }
            val bmp: Bitmap = BitmapFactory.
            decodeStream(inputStream)
            val image = findViewById<ImageView>(R.
            id.odabranaSlika)
            image.setImageBitmap(bmp)
        }
    }
    dugme.setOnClickListener {
        val intent: Intent = Intent()
        intent.action = Intent.ACTION_GET_CONTENT
        intent.type = "image/*"
        launcher.launch(intent)
    }
```

onData() – testiranje dinamičkih podataka

Kada se radi s `View` elementima koji u nekom trenutku mogu biti van ekrana ili koji se učitavaju nakon scrollanja poput listi koje koriste `AdapterView`, tada se ne može koristiti `onView()` metoda već se treba koristiti `onData()`.

Ova metoda koristi `Adapter` da pronađe element koji se traži i pozicionira listu na odgovarajuću poziciju.

Na primjer, ako se u spinneru želi odabrati neka vrijednost tada se simulira klik na spinner i onda iskoristi `onData()` da bi se došlo do odgovarajuće pozicije.

U nastavku slijedu primjer testa u kojem spinner sa id-em `R.id.tipovi` sadrži vrijednosti "novi", "polovni" i "iznajmljivanje", te se želi odabrati novi i provjeriti da li se u `TextView` elementu `R.id.rezultat` nalazi tekst "Novi automobil".

```
@Test
fun spinnerTest(){
    onView(withId(R.id.tipovi)).perform(click())
    onData(allOf(Is(instanceOf(String::class.java)),
Is("novi"))).perform(click())
    onView(withId(R.id.rezultat)).check(matches(withText("Novi
    automobil")))
}
```

Testiranje RecyclerView elementa

Za razliku od spinnera, `RecyclerView` ne podržava testiranje putem `onData()` metode. U ovoj vrsti testova potrebno je koristiti `espresso-contrib` paket i `RecyclerViewActions` pomoću kojih se može prolaziti kroz `RecyclerView`. Često korištene akcije su:

- `scrollTo()` - skrolanje do `View`-a koji ispunjava uslov *view matcher*-a
- `scrollToPosition()` - skrolanje do pozicije
- `actionOnItem()` i `actionOnItemAtPosition()` - izvršavanje akcije na `View` elementu koji zadovoljava *matcher* ili poziciju

Da bi se koristile navedene akcije potrebno je u `app/build.gradle` dodati *dependency*:

```
androidTestImplementation 'androidx.test.espresso:espresso-
contrib:+'
```

Primjer: neka postoji lista studenata u `RecyclerView` elementu koji ima atribute ime i index, i u testu se želi provjeriti da li ima student "Neki Student" sa indeksom 12345:

```
onView(withId(R.id.listaStudenata)).perform(
    RecyclerViewActions.scrollTo<RecyclerView.ViewHolder>(
        CoreMatchers.allOf(
            hasDescendant(withText(“Neki Student”)),
            hasDescendant(withText(“12345”))
        )
    )
)
```

Ako se želi provjeriti da li ovaj `RecyclerView` sadrži određeni broj elemenata, tada se može definisati nova `ViewAssertion` metoda:

```
fun hasItemCount(n: Int) = object : ViewAssertion {
    override fun check(view: View?, noViewFoundException:
    NoMatchingViewException?) {
        if (noViewFoundException != null) {
            throw noViewFoundException
        }
        assertTrue(“View nije tipa RecyclerView”, view is
        RecyclerView)
        var rv: RecyclerView = view as RecyclerView
        assertThat(
                “GetItemCount RecyclerView broj elementa: “,
                rv.adapter?.itemCount,
                Is(n)
        )
    }
}
```

U ovoj metodi se provjerava da adapter u `RecyclerView`-u ima broj elemenata koji odgovara proslijeđenom parametru `n`. Navedena metoda se može u testu pozvati na sljedeći način:

```
onView(withId(R.id.listaStudenata)).check(hasItemCount(5))
```

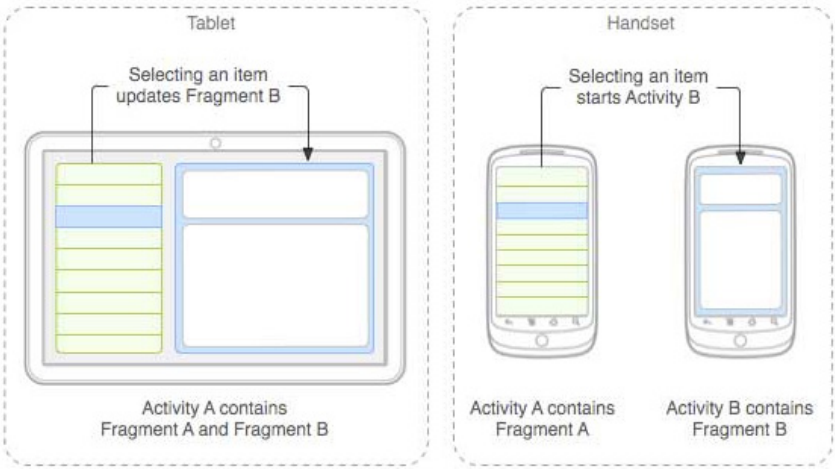
3.2.17. Fragmenti i navigacijska komponenta

Kod dizajniranja korisničkog interfejsa za mobilne aplikacije treba imati na umu da postoji veliki broj uređaja sa različitom veličinom displeja i gustinom prikaza. U zavisnosti od ove dvije veličine, različit raspored komponenti ili druge komponente korisničkog interfejsa mogu biti pogodnije. Dakle, može se zaključiti da bi bilo pogodno jednu aplikaciju prikazati na različite načine na različitim uređajima, kako bi se iskoristile sve pogodnosti veličine display-a i gustine prikaza slike.

Raspoređivanje komponenti korisničkog interfejsa u fleksibilne *layout*-e pomaže prilikom prikaza aplikacije na različitim rezolucijama. Međutim, postoje situacije kada to nije dovoljno, kao što je slučaj kada se dvije aktivnosti prikazuju kao jedna, odnosno obje se prikazuju na jednom ekranu. Da bi se to postiglo sadržaj aktivnosti se enkapsulira u `Fragment`.

`Fragment` je modularni dio aktivnosti koji se može u toku izvršavanja aplikacije skinuti sa aktivnosti ili dodati na aktivnost [29]. Fragment je po svojoj prirodi poput podaktivnosti sa svojim vlastitim životnim ciklusom, ulaznim eventima itd. Fragmenti su ponovo iskoristivi dijelovi aktivnosti koji se mogu koristiti u različitim aktivnostima i na taj način smanjuju količinu ponovno napisanog istog koda.

Jedan od načina kako iskoristiti veći prostor *display*-a jeste da se aktivnosti koje su se prije samostalno prikazivale (npr. lista kontakata i detalji o kontaktu) izdvoje u fragmente tako da se ti fragmenti mogu dodati jednoj aktivnosti i prikazati jedan do drugog. U slučaju manjih display-a i dalje se mogu prikazivati dvije odvojene aktivnosti, čiji je sada sadržaj izdvojeni fragment (slika 9).



Slika 9. Fragmenti[25]

Da bi interfejs aplikacije bio dinamičan fragmenti se često dodaju i skidaju sa aktivnosti. Kako bi se najbolje shvatio ovaj proces bitno je poznavati životni ciklus fragmenta. Kada se dodaje fragment nekoj aktivnosti tada:

[25] Slika preuzeta sa: https://www.tutorialspoint.com/android/android_fragments.htm

– Aktivnost uzima referencu na fragment
– Aktivnost uzima referencu na `ViewGroup` unutar kojeg će se fragment prikazivati
– Fragment kreira svoj `View` koji je definisan u odgovarajućem *layout*-u fragmenta
– Fragment vraća kreirani `View` aktivnosti
– Aktivnost dodaje fragmentov kreirani `View` i dodaje ga u `ViewGroup`

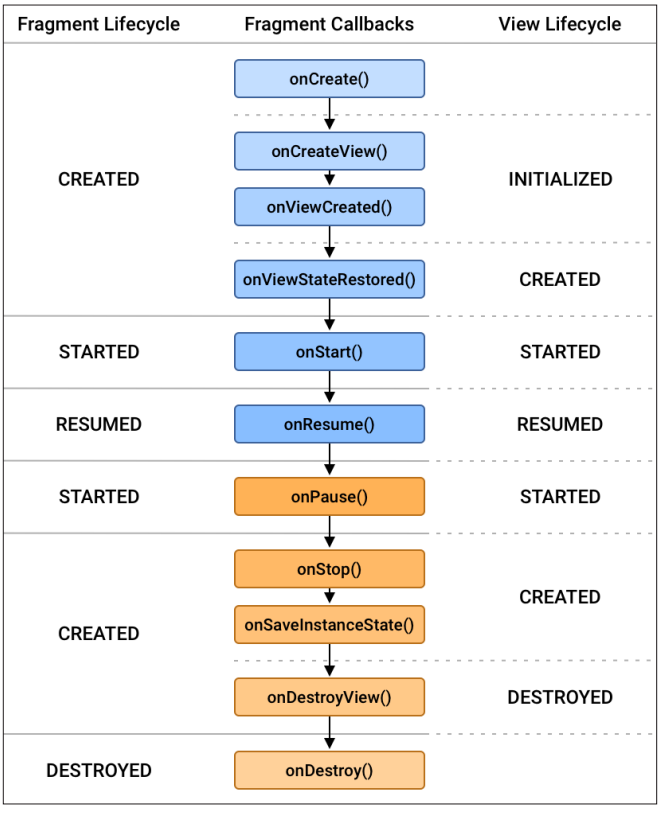
Događaji kroz koje fragment prolazi propraćeni su metodama koje se implementiraju unutar klase fragmenta. Svaki fragment prolazi određeni niz koraka od kreiranja do uništenja.

Događaji u nastavku su navedeni u poretku kako se javljaju od početka do kraja života jednog fragmenta:

– `onAttach()` - ova metoda se poziva kada se desi događaj dodavanja fragmenta. Ona je prva u nizu metoda i poziva se čak i prije `onCreate()` metode. Unutar ove metode se može vidjeti referenca na aktivnost koja će sadržavati fragment;
– `onCreate()` - ova metoda se poziva kada se fragment kreira. Unutar ove metode je potrebno postaviti sve što fragment treba da koristi u slučaju kada se fragment vraća iz *pause* ili *stop* stanja. Ova metoda se često implementira;
– `onCreateView()` - ova metoda se poziva kada sistem treba da iscrta korisnički interfejs fragmenta po prvi put. Ona vraća kao rezultat `View` objekat koji će se prikazati. U slučajevima kada fragment nema svoj korisnički interfejs tada ova metoda može vratiti *null*. Ova metoda se treba implementirati;
– `onViewCreated()` - ova metoda se poziva kada je završeno kreiranje `View` objekta fragmenta. Ona je korisna u slučajevima kada je potrebno dodatno izmijeniti `View` objekat. On se dobija kao parametar metode. Treba napomenuti da u ovom trenutku `View` nije dodjeljen `ViewGroup`-u unutar aktivnosti. Ovdje se mogu podesiti adapteri ako unutar fragmenta ima npr. `ListView`;
– `onActivityCreated()` - poziva se kada je aktivnost unutar koje će se fragment prikazivati kreirana i kada je fragmentov `View` spreman za prikazivanje. Unutar ove metode je poznato da je `onCreate()` metoda aktivnosti pozvana i da je sva inicializacija aktivnosti gotova;

– `onStart()` - u ovom koraku se fragment prikazuje na odgovarajućem mjestu unutar aktivnosti. Ovaj događaj je povezan s `onStart()` događajem aktivnosti;
– `onPause()` - ovaj događaj se dešava kada sistem registruje da korisnik odlazi sa fragmenta (prebacuje na drugi fragment, drugu aktivnost, aplikaciju, itd). Ovdje je potrebno spasiti trenutno stanje, ukoliko fragment nešto trenutno radi, kako bi se moglo nastaviti kada fragment postane ponovo vidljiv korisniku;
– `onStop()` - poslije ove tačke fragment će biti zaustavljen;
– `onDestroyView()` - u ovom koraku je potrebno osloboditi određene resurse koje fragment koristi ukoliko postoji potreba za tim;
– `onDestroy()` - posljednja faza uklanjanja fragmenta;
– `onDetach()` - obavijest da je uklanjanje fragmenta završeno.

Životni ciklus fragmenta se može prikazati grafički (slika 10):



Slika 10. Životni ciklus fragmenta[26]

[26] Slika preuzeta sa: https://developer.android.com/guide/fragments/lifecycle

3.2.17.1. Dodavanje fragmenta u aktivnost

Postoje dva načina kako možete ubaciti fragment u aktivnost, a to su statički i dinamički. Statičko dodavanje fragmenta se radi putem fragment taga. Tada se fragment tag dodaje u *layout* aktivnosti. Dinamičko dodavanje fragmenta radi se na način da se unutar *layout*-a aktivnosti rezerviše mjesto za fragment, a onda putem kôda, kada je potrebno, fragment dinamički ubacije u unaprijed rezervisani prostor. Mjesto za dinamičko dodavanje fragmenta se može rezervisati putem `FrameLayout`-a ili nekog sličnog `ViewGroup` elementa. Dodavanje, brisanje i zamjena fragmenata je moguće i ono se radi putem `FragmentManager`-a ili `SupportFragmentManager`-a, ako je aktivnost izvedena iz *support* biblioteke. Ako se dodaj, briše i zamjenjuje fragment unutar fragmenta onda trebate koristiti `ChildFragmentManager`.

Proces dinamičkog dodavanja fragmenta počinje sa dohvatanjem fragment managera sa `getFragmentManager()` (ili `getSupportFragmentManager()`, ako se koristi *support* biblioteka ili `getChildFragmentManager()` za rad sa fragmentom unutar fragmenta.

Napomena: Fragment klasa je deprecated od API 28. Obzirom da je najveći broj uređaja <28 API, koristi se *support* biblioteka ili *Jetpack Fragment*.

Prije dodavanja, zamjene ili brisanja fragmenta potrebno je početi fragment transakciju. Ova transakcija ima sličnu ulogu kao transakcija u bazama podataka. Transakcija osigurava da se zna kada promjene počinju, šta je učinjeno sa fragmentima i kada su promjene završene.

Zamjena fragmenta se radi s `replace`:

```
supportFragmentManager.commit {
   replace<ExampleFragment>(R.id.fragment_container)
   setReorderingAllowed(true)
   addToBackStack("name")
}
```

Brisanje fragmenta se radi s `remove`, a dodavanje s `add`:

```
 fragmentManager.commit {
    // Instancirati prije dodavanja
    val myFragment = ExampleFragment()
    add(R.id.fragment_view_container, myFragment)
    setReorderingAllowed(true)
}
```

Dohvatanje reference na fragment se može raditi preko ID i tag-a.

Primjer preko ID-a:

```
supportFragmentManager.commit {
   replace<ExampleFragment>(R.id.fragment_container)
   setReorderingAllowed(true)
   addToBackStack(null)
}
//Dohvatanje reference
val fragment: ExampleFragment =
        supportFragmentManager.findFragmentById(R.id.fragment_
        container) as ExampleFragment
```

Primjer preko tag-a:

```
supportFragmentManager.commit {
   replace<ExampleFragment>(R.id.fragment_container, "tag")
   setReorderingAllowed(true)
   addToBackStack(null)
}
//Dohvatanje reference
val fragment: ExampleFragment =
        supportFragmentManager.findFragmentByTag("tag") as
        ExampleFragment
```

Na kraju svake fragment transakcije se treba izvršiti `commit`. `Commit` fragment transakcije se rade pomoću istoimene metode, koja ne prima parametar.

Ukoliko se želi omogućiti da se fragment transakcije vraćaju kada korisnik klikne *back* dugme, onda se treba poslije zamjene/brisanja/dodavanja fragmenta, a prije `commit`-a, dodati transakciju na *backstack*. To se radi s metodom `addToBackStack` gdje se kao parametar proslijeđuje string koji označava ime transakcije. Ovo ime može biti korisno ukoliko se želi razlikovati razne tipove transakcija (ako ih ima više), u većini slučajeva dovoljno je proslijediti `null`.

3.2.18. Deklarisanje fragmenta

Da bi se kreirao jedan fragment potrebno je implementirati njegovu klasu. Klase fragmenata koje se implementiraju potrebno je izvoditi iz bazne

klase `Fragment`. Kako je već ranije napisano potrebno je implementirati neke od njegovih metoda poput, `onCreate()` i `onCreateView()`.

Kada se implementira metoda `onCreateView()`, potrebno je kreirati `View` objekat fragmenta. Da bi se kreirao `View` poziva se `inflate()` metoda `Inflater` objekta kojeg se dobija kao parametar `onCreateView()` metode.

`inflate()` metoda kao parametre prima redom:

- ID *layout*-a koji definiše izgled fragmenta
- referencu na `ViewGroup` u kojeg se ubacuje novokreirani `View`
- treći parametar je *boolean* koji označava da li se želi `View` povezati na `ViewGroup`. Pošto će sistem poslije odraditi taj posao, ovdje je potrebno proslijediti vrijednost `false`.

Kao rezultat metode `onCreateView()` potrebno je vratiti `View` kojeg je kreirala `inflate` metoda.

Layout fragmenta se definiše na isti način kako se definiše i *layout* za aktivnost i element `ListView`-a opisan ranije. Više o fragmentima se može pronaći na: Fragments[27]

Danas se rijetko kada fragmenti mijenjaju programski, već se koristi navigacijska komponenta, o kojoj će više detalja biti navedeno u poglavlju 4.

3.2.19. Web servisi i korutine

Većina Android aplikacija ne bi imala smisla bez mogućnosti da dijeli i prikuplja podatke putem Interneta. Za razmjenu podataka sa drugim aplikacijama, koje se mogu izvršavati na raznim okruženjima i udaljenim lokacijama koriste se web servisi. Web servisi su dijelovi softvera koji omogućavaju razmjenu podataka između heterogenih sistema putem Interneta [30].

3.2.19.1. Korištenje web servisa u Android aplikaciji

Web servisi otvaraju mogućnost nekoj drugoj aplikaciji da pozove neku od metoda udaljenog sistema kako bi dobila podatke ili obavila neku akciju nad tim sistemom. Primjer jednog web servisa jeste servis za vremensku prognozu *Open Weather Map*. Ovaj sistem za vremensku prognozu omogućava da se za određenu geografsku lokaciju, putem metoda web servisa, dobije trenutna vremenska prognoza.

[27] https://developer.android.com/guide/fragments

Poput svake metode i metode web servisa primaju neke parametre (u ovom slučaju geografsku širinu i dužinu) i vraćaju rezultat određenog tipa (u ovom slučaju niz vrijednosti vremenske prognoze kodirane u JSON formatu).

`JSON` je format koji objekte predstavlja u obliku objekata iz JavaScript programskog jezika. Svi atributi jednog objekta se nalaze unutar vitičastih zagrada, atributi su pobrojani kao `"kjuc":vrijednost`, vrijednost može biti i broj i string, ali i neki drugi objekat.

Ukoliko se pozove jedna metoda web servisa *Open Weather Map* za vrijednosti geografske širine i dužine koje odgovaraju Sarajevu dobit će se rezultat u JSON formatu koji odgovara sljedećoj slici.

http://api.openweathermap.org/data/2.5/weather?lat=43.87&lon=18.42&appid=44db6a862fba0b067b1930da0d769e98

{"coord":{"lon":18.39,"lat":43.88},"weather":[{"id":701,"main":"Mist","description":"mist","icon":"50d"}],"base":"cmc stations","main":{"temp":274.15,"pressure":1031,"humidity":80,"temp_min":274.15,"temp_max":274.15},"wind":{"speed":0.5},"clouds":{"all":75},"dt":1453723200,"sys":{"type":1,"id":5967,"message":0.0031,"country":"BA","sunrise":1453702241,"sunset":1453736836},"id":3197964,"name":"Kobilja Glava","cod":200}

Snimak zaslona 13. Rezultat poziva web servisa

Web servis za dobijanje podataka o vremenskoj prognozi je pozvan putem `http` protokola. Gore navedeni link se može ukucati i u web pretraživač kako bi se dobila vremenska prognoza.

Parametri metode za dohvatanje vremenske prognoze se navode nakon adrese web servisa http://api.openweathermap.org/data/2.5/weather i oni se nalaze u obliku `naziv_parametra=vrijednost` odvojeni sa simbolom `&` jedni od drugih i simbolom `?` odvojeni od adrese web servisa. Ovako upućen zahtjev za web servisom koristi `GET` metodu `HTTP` protokola gdje se svi parametri web servisa šalju u URL-u. `HTTP` je protokol namijenjen za interakciju sa sistemima koji se nalaze na mreži. On ne pamti stanja i zasnovan je na razmjeni poruka putem zahtjeva i odgovora. U komunikaciji koja koristi `HTTP` protokol ističu se klijent i server. Klijent je identitet koji uspostavlja komunikaciju s ciljem da pošalje jedan ili više zahtjeva. Server prihvata konekciju. obrađuje zahtjeve od klijenta i odgovara sa odgovarajućom porukom. Više o `HTTP` protokolu se može naći u specifikaciji na IETF.[28]

[28] https://datatracker.ietf.org/doc/html/rfc7230

Pored parametara geografske širine i dužine (`lat` i `lon`) naveden je i parametar `appid`. Ovaj parametar je identifikacijski broj putem kojeg web servis zna koja aplikacija zahtijeva njegove podatke. Često se ovaj identifikacijski broj dobija putem neke vrste registracije na web stranici web servisa gdje se u slučaju komercijalnih web servisa mora izvršiti pretplata čija cijena često zavisi od broja poziva metoda web servisa u toku jednog obračunskog perioda (npr. mjeseca ili godine). Postoji veliki broj web servisa koji ne naplaćuju svoje usluge za nekomercijalne aplikacije i za aplikacije čiji broj poziva je ispod određene granice, a postoje i web servisi koji su u potpunosti besplatni.

Kao što se može vidjeti, rezultat poziva web servisa može da sadrži brojne atribute poput objekta s nazivom `main` koji sadrži osnovne atmosferske podatke poput pritiska, temperature, vlažnosti i sl. Ove podatke je dalje moguće iskoristiti u nekoj aplikaciji. Ukoliko bi se razvila mobilna aplikacija za prikaz vremenske prognoze, sa svakim pozivanjem ove metode bi se dobijali podaci o vremenskoj prognozi relevantni u trenutku pozivanja i s takvim podacima bi i aplikacija bila relevantna i upotrebljiva.

Ono što se može primijetiti u vezi web servisa jeste da nije poznato kako radi sistem web servisa u pozadini, koji je programski jezik korišten za njegov razvoj, koje okruženje. Web servis je načešće "crna kutija" sa dobro definisanim interfejsima. U slučaju ispravne izmjene web servisa i zamjene s drugim web servisom pisanim u drugom okruženju i/ili programskom jeziku s identičnim interfejsom (isti nazivi metoda, parametara, atributa rezultata i strukture rezultata), aplikacija bi i dalje trebala raditi bez problema.

3.2.19.2. Poziv web servisa

Da bi uopšte mogli pristupiti resursima na Internetu potrebno je dodati permisije u manifest datoteci kako bi Android aplikacija mogla koristiti Internet. Permisije za pristup Internetu se postavljaju s XML elementom `uses-permission` i atributom `android:name="android.permission.INTERNET`. Kompletan XML element za permisije za pristup internetu je:

```
<uses-permission android:name=" android.permission.INTERNET" />
```

Ovaj element je potrebno postaviti u `Android manifest` datoteku aplikacije prije `application` tag-a. Nakon dodavanja potrebne permisije aplikacija može putem HTTP protokola pristupiti resursima na Internetu.

Svaki HTTP resurs ima svoju odgovarajuću adresu koja se zadaje putem URL klase. URL klasa se inicializuje putem konstruktora koji prima parametar tipa string, a koji predstavlja stringovnu reprezentaciju URL adrese resursa.

```
val url = new URL(adresaResura)
```

Nakon što se kreira `url` objekat potrebno je otvoriti konekciju prema resursu. Podaci o konekciji se čuvaju u klasi `URLConnection`, koja se može prilagoditi (kastimizirati) u klasu `HttpURLConnection` koja će obezbjediti atribute i metode za rad s konekcijom preko HTTP protokola.

```
(url.openConnection() as? HttpURLConnection)?.run {
    val responseCode = this.getResponseCode()
    val result = this.inputStream.bufferedReader().use {
it.readText() }
 }
```

Kada se putem `HttpConnection` objekta pošalje zahtjev prema lokaciji navedenoj u `url`, očekuje se odgovor koji će sadržavati tražene podatke ili kôd koji označava da je došlo do greške, ako nije moguće dostaviti podatke. Ukoliko je moguće dobiti podatke sa lokacije navedene u `url` objektu, tada `responseCode` ima vrijednost koja odgovara enum vrijednosti `HTTP_OK` i tada je moguće preuzeti podatke u obliku `InputStream`-a, izvršiti procesiranje nad dobijenim podacima i iskoristiti ih na način kako zahtijeva poslovni dio aplikacije.

3.2.19.3. Blokiranje glavne niti

Ukoliko bi se ovaj kôd ubacio u metodu `onCreate` ili neku drugu metodu aktivnosti (npr. da se ovaj kôd pozove klikom na dugme), desio bi se izuzetak tipa `NetworkOnMainThreadException`. Ovaj izuzetak se javlja jer kôd, koji dohvata podatke sa Interneta, blokira izvršavanje glavne niti (eng. *main thread*) koja ima ulogu da ažurira korisnički interfejs aplikacije. Neke manje složene akcije u glavnoj niti se mogu izvršavati bez problema. Međutim, dohvatanje podataka s Interneta može trajati dugo (preko par sekundi, pa čak i minuta u slučaju spore konekcije ili velikih podataka). Zbog svega toga, dohvatanje podataka s Interneta ne bi trebalo obavljati u niti koja ima tako važnu ulogu, koja osigurava da aplikacija reaguje na korisničke ulaze i čini aplikaciju upotrebljivom.

Izdvajanjem akcije dohvatanja podataka u posebnu nit, ili izvršavanje akcija konkurentno s glavnom niti, osigurava se da, dok se podaci učitavaju, korisnik može obavljati druge akcije nad aplikacijom i čini takvu aplikaciju ugodnom za korisnike. Da bi se spriječilo blokiranje glavni niti mogu se koristit `Coroutine`.

`Coroutine` nije nit u Kotlin-u. Moguće je pokrenuti neograničen broj `coroutine`-a u nekoj niti i to bez njenog blokiranja, te se na ovaj način postiže asinhrono ponašanje. Međutim, u Androidu je ograničen broj niti po aplikaciji.

`Coroutine` je blok kôda koji se izvršava bez blokiranja niti na kojoj se izvršava, odnosno ne čeka da se blok kôda izvši. Iako ne čeka da se kôd završi, ima definisane interne `callback` metode nakon završetka istog. U nastavku je dat pregled osnovnih stavki `coroutine`-a.

3.2.19.4. CoroutineScope

`CoroutineScope` su podklase `CoroutineContext`. One su ograničavajući faktor `coroutine`-a, odnosno svaka `coroutine` se izvršava u kontekstu prostora (eng. *scope*). U Androidu postoji `GlobalScope` ili razni *scope* za `ViewModel`, `Activity` i `Fragment`.

3.2.19.5. Builders

`Builders` definišu kako će se ponašati `coroutine`-a kada se pozove. Ovo su obične metode koje se vrše van ili unutar `coroutine`-e i koje rezultiraju s `coroutine` instancom. Postoje više načina na koje se može pokrenuti `coroutine`:

– `runBlocking` - sve unutar ovog *builder*-a blokira nit na kojoj se izvršava. Ova metoda se rijetko kad koristi.

```
runBlocking {
    // blokirajuci kod
}
```

– `launch` - ovaj *builder* koristi `dispatcher` koji definiše lokaciju izvršavanja. `Coroutine` se može izvršavati u pozadini, na trenutnoj niti ili gdje god je definisana. Primjer:

```
GlobalScope().launch(Dispatchers.IO) {
    // Asinhroni kod
}
```

– `async` - izvršava sve asinhrono. Razlika između `launch` i `async` je što `async` vraća `Deffered` objekat, a `launch` `Job` objekat. `Async` se najčešće koristi kada se želi paralelno izvršavati zadatke. Na primjer, ako ima 10 poziva i svaki traje 3 sekunde. Ukupno vrijeme izvršavanja svih ovih poziva će biti 3 sekunde, odnosno svi pozivi se izvršavaju paralelno. Primjer:

```
GlobalScope().launch(Dispatchers.IO) {
    // k&#xF4;d
    val result = async { myApiCall() }
    // linija se odmah izvrsava nakon prethodne
    // nece cekati 10 sekundi
    result.await() // rezultat - string ce se dobiti ovdje
}
suspend fun myApiCall() {
    delay(10_000) // 10 second delay koji imitira mrezni poziv
    return “Works!”
}
```

3.2.19.6. Dispatchers

`Dispatcher` definiše gdje će se `coroutine` izvršavati. Postoje sljedeće varijante `Dispatcher`-a:

– `Dispatchers.IO` - izvršava se na pozadinskoj niti. Ne izvršava kompleksna izračunavanja, nego I/O operacije
– `Dispatchers.Main` - izvršava se na glavnoj niti. Služi za ažuriranje korisničkog interfejsa na osnovu podataka dobijenih iz pozadinskih niti
– `Dispatchers.Default` - izvršava se na odvojenoj niti. Preporučuje se izvršavanje CPU intenzivnih zadataka

3.2.19.7. Izmjena konteksta

U `coroutine`-i je dozvoljena izmjena `Dispatchers`-a. Na primjer, `coroutine`-a je mogla započeti u IO. Međutim, može postojati potreba da se izmijeni UI kada se zadatak završi. Ovo je moguće uraditi samo ako se nalazi u glavnoj niti. Zato će se izvršiti izmjena *context*-a korištenjem `withContext` metode. Primjer:

```
GlobalScope().launch(Dispatchers.IO) {
    // dugi zadatak
```

```
    withContext(Dispatchers.Main) { // UI update }
}
```

Više o `Coroutines` se može pronaći na: Coroutines[29]

3.2.20. Servisi

Servisi su komponente koje omogućavaju izvršavanje dugih zadataka u pozadini [31]. Oni ostaju aktivni i u slučajevima kada njihove aplikacije nisu vidljive. Mogu raditi neko vrijeme i nakon što se korisnik prebaci na korištenje druge aplikacije. Servisi imaju veći prioritet od aktivnosti i ukoliko se desi da Android nema dovoljno resursa servis će imati manje šanse da bude ugašen od aktivnosti. Aplikacije koje imaju neki posao koji se izvršava u pozadini imaju veći prioritet od aplikacija koje to nemaju.

Razlika između servisa i aktivnosti je što servisi nemaju svoj odgovarajući korisnički interfejs. Aktivnosti su dizajnirane da se pokreću, restartuju i zatvaraju tokom svog životnog ciklusa, dok su servisi dizajnirani da duže traju. Servisi zavise od aktivnosti, prijemnika obavijesti i drugih servisa. Oni nadziru njihovo kreiranje, kontrolisanje i zaustavljanje.

Napomena: Servis se izvršava u glavnoj niti. On ne kreira vlastitu nit i ne izvršava proces u njoj, osim u situacijama kada se to eksplicitno specificira. U slučaju izvršavanja blokirajućih operacija potrebno je smjestiti sve u posebnu nit.

3.2.20.1. Kreiranje

Da bi kreirali servis potrebno je implementirati klasu koja je nasljeđena iz `Service` osnovne klase. Ova klasa definiše razne metodekoje se mogu implementirati za potrebe aplikacije, tako da se aplikacija za korisnika ponaša na očekivan način. Nakon kreiranja klase potrebno je implementirati dvije metode `onCreate()` i `onBind()`.

Kako je servis komponenta aplikacije, potrebno ju je registrovati u manifest datoteci. Za registrovanje servisa koristi se `<service>` tag i smješta unutar `<application>` tag-a. Unutar `<service>` taga potrebno je navesti dva atributa: `android:enabled=”true”` i `android:name` kojem se dodjeljuje naziv klase servisa.

[29] https://developer.android.com/kotlin/coroutines

```
<service android:enabled=”true” android:name=”.MyService”/>
```

Ukoliko se želi osigurati servis, da mu druge aplikacije ne mogu pristupati bez dobijanja odgovarajući permisija, potrebno je koristiti još jedan atribut unutar `<service>` tag-a. Atribut `android:permission` se postavlja na vrijednost naziva permisije koja se zahtijeva da druga aplikacija ima kako bi koristila servis aplikacije.

3.2.20.2. Pokretanje

Da bi se mogao pokrenuti servis potrebno je dodati još jednu metodu u klasu servisa. Metoda `onStartCommand` priprema servis za početak njegovog izvršavanja. Kako se sve što se nalazi u `onStartCommand` metodi pokreće u glavnoj niti, praksa je da se na početku `onStartCommand` metode pokrene nova nit, a da se prilikom zatvaranja servisa zatvori i ta nit. Na ovaj način se osigura da servis ne blokira glavnu nit aplikacije.

```
override fun onStartCommand(intent: Intent?, flags: Int,
startId: Int): Int {
    return Service.START_STICKY;
}
```

Ukoliko se pogleda jedna tipična implementacija metode `onStartCommand` može se vidjeti da ta metoda vraća cjelobrojnu vrijednost koja govori o tome kako će se servis ponašati u slučaju ako se restartuje. Načini kako je moguće upravljati s restartanjem servisa su:

- `START_STICKY` – metoda `onStartCommand` će se pokretati svaki put kada se servis restartuje. Tada će vrijednost intent parametra biti *null*.
- `START_NO_STICKY` – servisi specificirani na ovaj način se restartuju samo u slučajevima ako postoji neki aktuelni zahtjev za njihovim startanjem. Ukoliko ni jednom nije pozvana metoda `serviceStart` nakon što je servis ugašen, tada se neće pozivati `onStartCommand` metoda. Ovakvi servisi se koriste za akcije koje ažuriraju aplikaciju u određenim vremenskim intervalima. U slučajevima ako aplikacija nije ažurirana u jednom intervalu, bit će ažurirana u sljedećem kada se servis pozove.
- `START_REDELIVER_INTENT` – ovaj režim se koristi ukoliko se želi biti siguran da su komande u servisu izvršene. Ovakva specifikacija servisa predstavlja kombinaciju prethodne dvije. Ukoliko je servis zaustavljen, servis će se restartati samo ako postoji aktuelni zahtjev za startanjem ili

ako je servis zaustavljen prije pozivanja `stopSelf` metode. U drugom slučaju pozvat će se `onStartCommand` metoda i njoj će biti proslijeđen orginalni intent čije procesiranje nije završeno.

3.2.21. Pokretanje i zaustavljanje

Pokretanje servisa se radi slično startanju intenta. Postoje dva načina: implicitni i eksplicitni.

U slučaju eksplicitnog pokretanja servisa kreira se eksplicitni intent i njemu prosljeđuje klasa servisa kao drugi parametar, nakon čega se poziva `startService` metoda kojoj se prosljeđuje novokreirani intent. Primjer:

```
private fun explicitStart() {
    val intent : Intent =  Intent(this, MyService.class);
    startService(intent);
}
```

Implicitno kreiranje servisa je veoma slično implicitnom kreiranju intenta. Potrebno je kreirati implicitni intent i kao akciju poslati akciju specificiranu u servisu. Nakon toga je potrebno pozvati `startService` metodu. U oba slučaja moguće je dodati dodatne podatke u intent koji će biti prosljeđeni servisu putem `putExtra` metode. Primjer:

```
private fun implicitStart() {
    val intent =  Intent(MyMusicService.PUSTI_PJESMU);
    intent.putExtra(MyMusicService.NAZIV_EXTRA, “Naziv pjesme”);
    startService(intent);
}
```

Način zaustavljanja servisa zavisi od toga kako je on pokrenut. Ako se radi o eksplicitnom pokretanju, potrebno je samo pozvati metodu `stopService` i kao parametar prosljediti novi intent koji ima klasu servisa kao parametar. U slučaju servisa koji je pokrenut implicitno, potrebno je kreirati novi impcitni intent u kojem je navedena akcija koja je bila i prilikom pokretanja servisa, nakon čega je potrebno pozvati metodu `stopService` i proslijediti novokreirani intent.

```
val intent = Intent(MyMusicService.PLAY_ALBUM);
stopService(intent);
```

Jedno pozivanje `stopService` metode će zatvoriti odgovarajući servis bez obzira koliko puta je pozvan `startService` nad tim servisom.

3.2.22. Tipovi

Postoje tri različite tipa servisa:

- *Foreground* – servis u prvom planu: servisi ove vrste izvode neku operaciju koja je uočljiva korisniku. Na primjer, audio aplikacija bi koristila *foreground* servis za reprodukciju audio zapisa. Ovi moraju prikazati `Notification`. Kada se koristi ovaj servis, potrebno je prikazati obavijest tako da korisnici budu svjesni da je usluga pokrenuta. Ovo se obavještenje ne može odbaciti sve dok se servis ne zaustavi ili ukloni iz prvog plana. Ova vrsta servisa se nastavlja izvršavati i kada korisnik nije u interakciji s aplikacijom.
- *Background* – servis u pozadini: ovaj servis izvršava operaciju koju korisnik ne zapaža direktno. Na primjer, ako je aplikacija koristila uslugu za sažimanje pohrane, to bi obično bila pozadinska usluga. Od API 26 sistem nameće ograničenja na pokretanje pozadinskih servisa kada sama aplikacija nije u prvom planu.
- *Bound* – vezani servis: servis je vezan kada se komponenta aplikacije veže za njega pozivanjem `bindService()` metode. Vezanai servis pruža klijent-server interfejs koji omogućava komponentama da komuniciraju sa servisima, šalju zahtjeve, primaju rezultate, i sl. Vezani servis radi samo dok je druga komponenta aplikacije vezana za njega. Više komponenti se odjednom može vezati za servis, ali kada se sve odvoje, servis se uništava.

3.2.23. Retrofit

Retrofit je HTTP klijent za Android aplikacije. S Retrofitom se može jednostavno kreirati HTTP konekcija putem jednostavnog interfejsa koji je sličan API dokumentu. Njegova sintaksa je veoma jednostavna za korištenje.

3.2.23.1. Anotacije

U Retrofit-u se koriste anotacije za opisivanje HTTP zahtjeva, kao što su:

- definisanje parametara URL-a i upita
- pretvaranje objekata u tijelo zahtjeva (pretvaranje u JSON)
- prenos tijela zahtjeva i datoteka s više dijelova (*multipart*-a)

Retrofit klijent podržava sljedeće anotacije za metode i relativni URL HTTP zahtjeva: HTTP, GET, POST, PUT, PATCH, DELETE, OPTIONS i HEAD. Relativni URL se navodi unutar anotacije.

```
@GET(“users/list”)
```

Parametri se mogu proslijeđivati u anotaciji na sljedeći način:

```
@GET(“users/list?sort=desc”)
```

3.2.23.2. URL zahtjevi

URL zahtjev se može dinamički ažurirati korištenjem zamjenskih blokova i parametara. Zamjenski blok se nalazi unutar vitičastih zagrada `{}`. Odgovarajući parametar metode se treba označiti s `@PATH`.

```
@GET(“group/{id}/users”)
fun groupList(@Path(“id”) groupId:Long):Call<List<User>>;
```

Omogućeno je dodavanje i `query` parametara.

```
@GET(“group/{id}/users”)
fun groupList(@Path(“id”) groupId:Long, @Query(“sort”)
sort:String):Call<List<User>>;
```

3.2.23.3. Tijelo zahtjeva

Objekat se može iskoristiti i slati kao tijelo zahtjeva korištenjem `@Body` anotacije.

```
@POST(“users/new”)
fun createUser(@Body user:User):Call<User>;
```

Objekat se pretvara u `RequestBody` korištenjem `converter-a` koji se specificira prilikom instanciranja Retrofita.

3.2.23.4. Form encoded i multipart

Metode se mogu deklarisati da šalju *form-encoded* i *multipart* podatke. *Form-encoded* podaci se šalju kada se koristi `@FormUrlEncoded` notacija na metodi. Svaki par ključ-vrijednost se obilježi sa `@Field` koji sadrži ime i objekat koji pruža vrijednost.

```
@FormUrlEncoded
@POST(“user/edit”)
fun updateUser(@Field(“first_name”)  first:String, @Field(“last_
name”)  last:String) : Call<User>;
```

Multipart zahtjevi se šalju s `@Multipart` anotacijom. Dijelovi se definišu korištenjem `@Parts`.

```
@Multipart
@PUT(“user/photo”)
fun updateUser(@Part(“photo”)  photo:RequestBody, @
Part(“description”)  description:RequestBody):Call<User>;
```

Dijelovi *multipart*-a koriste ili Retrofit konverter ili implementiranu serijalizaciju unutar `RequestBody`.

3.2.23.5. Upravljanje zaglavljem

Zaglavlja se mogu definisati korištenjem `@Headers` anotacije.

```
@Headers(“Cache-Control: max-age=640000”)
@GET(“widget/list”)
fun widgetList():Call<List<Widget>>
@Headers({
    “Accept: application/vnd.github.v3.full+json”,
    “User-Agent: Retrofit-Sample-App”
})
@GET(“users/{username}”)
fun getUser(@Path(“username”) String username):Call<User>
```

Zaglavlja se ne prepisuju jedna preko drugih. Sva zaglavlja s istim imenom će biti uključena u zahtjev.

3.2.23.6. Converters

Retrofit vrši deserijalizaciju HTTP tijela u `OkHTTP` `ResponseBody` tip i može prihvatiti samo vlastiti `RequestBody` tip za `@Body` anotaciju.

Konverteri se koriste kada se žele podržati i druge tipove. Postoji nekoliko sličnih modula koji vrše serijalizaciju objekata za `Retrofit:`

- Gson: `com.squareup.retrofit2:converter-gson`
- Jackson: `com.squareup.retrofit2:converter-jackson`
- Moshi: `com.squareup.retrofit2:converter-moshi`
- Protobuf: `com.squareup.retrofit2:converter-protobuf`
- Wire: `com.squareup.retrofit2:converter-wire`
- Simple XML: `com.squareup.retrofit2:converter-simplexml`
- JAXB: `com.squareup.retrofit2:converter-jaxb`

– Scalars: `com.squareup.retrofit2:converter-scalars`

Moguće je kreirati vlastitu klasu koja će vršiti serijalizaciju i deserijalizaciju i koja treba biti naslijeđena iz `Converter.Factory` klase.

U nastavku slijedi primjer korištenja `GsonConverterFactory` klase unutar `RetrofitBuilder-a`

```
val retrofit :Retrofit  = Retrofit.Builder()
    .baseUrl(“https://api.github.com/”)
    .addConverterFactory(GsonConverterFactory.create())
    .build();
```

3.2.23.7. GET i POST zahtjevi

U nastavku slijedi primjer `GET` i `POST` zahtjeva korištenjem Retrofit klijenta. U prvom koraku će biti kreiran singleton objekat za *build*-anje instance Retrofita.

```
object ApiService {
    private val TAG = “--ApiService”
    fun loginApiCall() = Retrofit.Builder()
            .baseUrl(Constants.API_BASE_PATH)
            .addConverterFactory(ApiWorker.gsonConverter)
            .client(ApiWorker.client)
            .build()
            .create(LoginApiService::class.java)!!
}
```

`GET` zahtjev:

```
interface UserApiService {
    @Headers(“Content-Type: application/json”)
    @GET(“user”)
    fun getUser(
            @Query(“Authorization”) authorizationKey: String, //
authentication header
            @Query(“UserID”) userID: String
    )
}
```

`POST` zahtjev:

```
interface LoginApiService {
    @Headers(“Content-Type: application/json”)
    @POST(“login”)
    fun doLogin(
            //@Query(“Authorization”) authorizationKey: String,
// authentication header
            @Body loginPostData: LoginPostData):
Observable<LoginResponse> // body data
}
```

Obzirom da je proslijeđeno tijelo u `POST` metodi, potrebno je prvo izvršiti serijalizaciju objekta. Korištenjem `GSONConverter-a` i anotacijom `@SerializedName` u definisanju klase objekta postiže se automatska serijalizacija i eventualna deserijalizacija.

```
data class LoginPostData(
        @SerializedName(“UserId”) var userID: String,
        @SerializedName(“Password”) var userPassword: String
)
```

Više informacija o Retrofitu se može pronaći na ovom linku[30].

3.2.24. Perzistencija podataka

Kao i druge aplikacije kod kojih je potrebno da se podaci koji su prikupljeni tokom rada ostanu dostupni i prilikom sljedećeg pokretanja aplikacije i Android aplikacije imaju svoj mehanizam za perzistenciju podataka, a to je SQLite baza podataka. Ona omogućava da se na struktuiran način sačuvaju podaci, na isti način kako se radi sa bilo kojom drugom SQL bazom.

SQLite baza podataka je relaciona baza podataka čije su datoteke privatne i dostupne samo za aplikaciju koja ih je kreirala.

Nakon što se kreira baza moguće je kreirati provajder sadržaja (eng. *Content Provider*) kako bi se aplikacijska logika i struktura podataka odvojila. Ovo je dobra praksa, jer tada aplikacija ne mora brinuti o tome kakav je izvor podataka, već koristi standardizovani interfejs koji `ContentProvider`-i pružaju. Pomoću njih je moguće podatke iz aplikacije učiniti dostupnim drugim aplikacijama. Postoje `ContentProvider`-i za različite izvore podataka poput: kalendara, imenika, media *store*-a i sl. Moguće je također implementirati i vlastiti `ContentProvider` kako bi se uključili i drugi podaci u aplikaciju.

[30] https://square.github.io/retrofit/

3.2.25. SQLite baza podataka

SQLite je relaciona baza podataka koja zadovoljava sve standarde, koja je otvorenog kôda i čije izvršavanje nije zahtjevno [32]. Ona je sastavni dio aplikacije i sastavni dio softverskog steka zbog čega je: olakšana sinhronizacija i zaljučavanje transakcija, minimizovano kašnjenje i eliminisan problem vanjskih zavisnosti. Razlika između standardnih SQL baza je što tip vrijednosti u jednoj koloni ne mora biti isti, što znači da svaki red može imati drugačiji tip podataka.

SQLite je osigurana u obliku biblioteke koja je povezana s aplikacijom. Sve operacije baze podataka obrađuju se interno u aplikaciji putem poziva funkcija koje se nalaze u SQLite biblioteci.

Android SQLite je *open-source* sistem za upravljanje bazama podataka (eng. *Database Management System* - DBMS) koja se koristi za izvršavanje operacija baze podataka na Android uređajima, kao što su smiještanje, ažuriranje, dobavljanje podataka iz baze podataka. SQLite je baza podataka koja dolazi s Android OS, tako da svaka Android aplikacija može kreirati svoju vlastitu SQLite bazu podataka. Android ima ugrađenu podršku za SQLite bazu podataka, tako da nije potrebna nikakva konfiguracija. SQLite ne zahtijeva aktivnosti vezane za određena podešavanja ili administraciju. Android smiješta bazu podataka na privatni prostor na disku koji je povezan s aplikacijom. Podaci su sigurni, jer taj prostor nije dostpan drugim aplikacijama. U SQLite bazi podacima se pristupa upotrebom *Structured Query Language*-a (SQL). Podaci su privatni i njima može pristupiti samo aplikacija koja ih je kreirala.

Upravljati bazom je moguće putem direktnih *low-level* API-ja. Iako su ovi API-ji moćni, za njihovo korištenje je potrebno puno vremena i truda, tako da:

- Ne postoji provjera neobrađenih SQL upita u vrijeme prevođenja. Kako se skup podataka mijenja, potrebno je ručno ažurirati odgovarajuće SQL upite. Ovaj postupak može biti dugotrajan i podložan greškama.
- Za pretvaranje između SQL upita i podatkovnih objekata treba se koristiti puno *boilerplate* kôda.

Iz ovih razloga se preporučuje upotreba `Room Persistence Library` kao apstraktnog sloja nad bazom podataka.

3.2.26. Room Persistence Library

Aplikacije koje obrađuju netrivijalne količine struktuiranih podataka mogu imati velike koristi od perzistencije podataka. Najčešći je slučaj keširanje relevantnih dijelova podataka, tako da, kada uređaj ne može pristupiti mreži, korisnik i dalje može pregledati taj sadržaj dok je izvan mreže.

Room persistence library je dio Android Jetpack komponenti od 2016. godine i predstavlja neku vrstu apstrakcije između klijenta i baze podataka [33]. `Room` biblioteka za perzistenciju podataka pruža sloj apstrakcije preko SQLite-a kako bi omogućila jednostavan pristup bazi podataka, istodobno iskorištavajući punu snagu SQLite-a. `Room` pruža sljedeće pogodnosti:

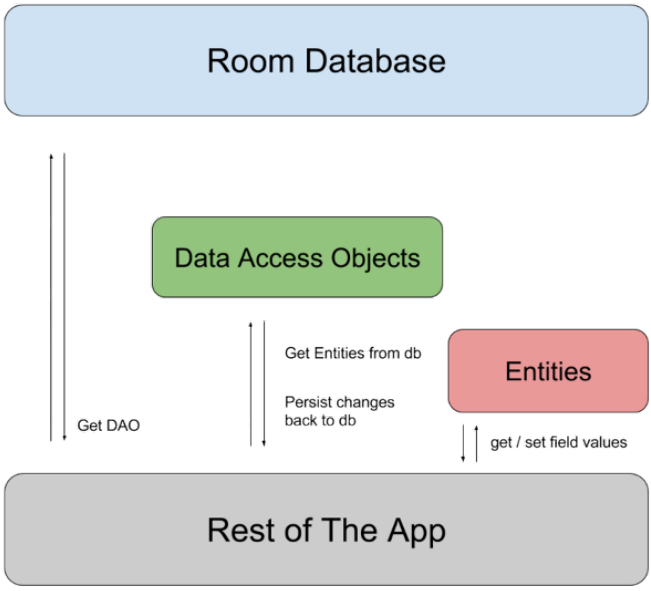
- Provjera SQL-upita tijekom kompajliranja
- Praktične anotacije koje minimiziraju ponavljajući i pogreškama podložan kôd
- Pojednostavljeni putevi migracije baze podataka

3.2.26.1. Primarne komponente

Glavne komponente biblioteke `Room` su:

- `Database` – Klasa baze podataka koja sadrži bazu podataka i služi kao glavna pristupna tačka s trajnim podacima aplikacije. Predstavlja skup tabela u kojima se čuvaju podaci. Služi kao vlasnik baze podataka. Treba biti označena s `@Database` i naslijeđena od `RoomDatabase`. Sadrži i vraća *Data Access Object* (`DAO`).
- `Enity` – Entitet podataka koji predstavlja tabelu u bazi podataka aplikacije. Treba biti označen sa `@Enity`. Broj kolona odgovara broju atributa klase koja se smiješta u bazu. Svaki se entitet sastoji od najmanje jednog polja koje se definiše kao primarni ključ.
- *Data Access Object* (`DAO`) – Objekti za pristup podacima koji sadrže svu logiku i pružaju metode koje aplikacija može koristiti za postavljanje upita, ažuriranje, umetanje i brisanje podataka u bazi podataka. Predstavlja neku vrstu poveznice između programske logike i modela, odnosno podataka koji su smješteni u bazi. Često se za svaki `Enity` kreira jedan `DAO` objekat. Treba biti označena s `@DAO` anotacijom.

Na slici 11. su prikazani odnosi između različitih komponenata `Room`-a.



Slika 11. Room arhitekturalni dijagram[31]

Dodavanje potrebnih zavisnosti

Korištenje `Room` arhitekturalne komponente zahtijeva dodavanje određenih zavisnosti u `build.gradle` datoteku projekta:

```
apply plugin: “kotlin-kapt”

dependencies {
    def room_version = “2.3.0”

    implementation(“androidx.room:room-runtime:$room_version”)
    annotationProcessor “androidx.room:room-compiler:$room_
    version”

    // To use Kotlin annotation processing tool (kapt)
    kapt(“androidx.room:room-compiler:$room_version”)

    // To use Kotlin Symbolic Processing (KSP)
    ksp(“androidx.room:room-compiler:$room_version”)

    // optional - Kotlin Extensions and Coroutines support for Room
    implementation(“androidx.room:room-ktx:$room_version”)
```

[31] Slika preuzeta sa: https://developer.android.com/training/data-storage/room

```
    ....
}
```

Entity

Nakon importa zavisnosti slijedi kreiranje `Entity` objekta koji će predstavljati tabelu u bazi.

U gornjem primjeru je prikazana implementacija klase `ArticleEntity` iznad koje je dodana anotacija `@Entity`. U zagradi se može specificirati ime tabele kojoj *entity* odgovara u bazi podataka. Nakon toga se deklarišu polja klase s tim da se iznad/pored svakog polja stavlja anotacija `@ColumnInfo` i u zagradi se piše naziv kolone koja odgovara tom polju.

```
@Entity
data class ArticleEntity(
        @PrimaryKey var title: String,
        @ColumnInfo(name = “content”)
        var content: String
)
```

Svaka klasa bi trebala imati barem jedno polje označeno kao `PrimaryKey` kako bi se instance tabele mogle uniformno identifikovati. Primarni ključ se može automatski generisati, te promijeniti naziv tabele, kako je navedeno u nastavku:

```
@Entity(tableName = “article_items”)
data class ArticleEntity(
        @PrimaryKey(autoGenerate = true)
        var id: Int,

        @ColumnInfo(name = “title”)
        var title: String,
        @ColumnInfo(name = “content”)
        var content: String
)
```

DAO

Nakon što je kreiran `Entity` objekat, može se početi s implementacijom `Dao`-a koji će biti zadužen za svu logiku pristupa bazi podataka. Ovaj objekat je interfejs kojeg je potrebno označiti anotacijom `@Dao` i unutar njega se pišu SQL upiti za spremanje, dobavljanje, brisanje podataka i slično.

Ukoliko je potrebno dodati novi objekat u bazu, koristi se anotacija `@Insert` iznad funkcije koja prima objekat. U slučaju brisanja određenog objekta koristi se anotacija `@Delete`. Za druge SQL upite se može koristiti anotacija `@Query`, i u zagradi navesti odgovarajući upit.

Ovdje su definisane osnovne funkcionalnosti SQL baze podataka, kao što je dodavanje i brisanje. Anotacija `@Query` se može koristi za označavanje funkcija koje koriste upite. Moguće je koristiti i parametre u upitima koristeći `:nazivparametra`, kao što je navedeno (`:title`) u funkciji `findByTitle`.

```
@Dao
interface ArticleDao {
    @Query(“SELECT * FROM articleEntity”)
    fun getAll(): List<ArticleEntity>

    @Query(“SELECT * FROM articleEntity WHERE title LIKE :title”)
    fun findByTitle(title: String): ArticleEntity

    @Insert
    fun insertAll(vararg article: ArticleEntity)

    @Delete
    fun delete(article: ArticleEntity)

    @Update
    fun updateArticle(vararg articles: ArticleEntity)
}
```

Kreiranje baze podataka

Kreiranje baze počinje deklaracijom apstraktne klase koja nasljeđuje `RoomDatabase()` klasu i predstavlja glavnu pristupnu tačku ka podacima. Iznad deklaracije ove klase potrebno je dodati anotaciju `@Database`, te navesti listu `Entity`-a, odnosno tabela koje se nalaze u bazi i trenutnu verziju baze. Nakon toga se u tijelu klase navode kao apstraktne metode svi `DAO` objekti koji će se koristiti.

```
@Database(entities = arrayOf(ArticleEntity::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun articleDao(): ArticleDao
}
```

Pristup bazi podataka

Nakon definisanja baze podataka moguće je dobiti instancu u aktivnosti pomoću metode `Room.databaseBuilder()`.

```
val db = Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java, “article-list.db”
).build()
```

Sada se instanca može iskoristiti za pristup `DAO` objektu.

```
GlobalScope.launch {
            db.articleDao().insertAll(ArticleEntry(“Title”,
            “Content”))
            data = db.articleDao().getAll()
}
```

4.
ANDROID *JETPACK* KOMPONENTE

Dugo vremena su developeri Android mobilnih aplikacija imali različite poteškoće u radu s bazama podataka, upravljanjem životnim ciklusom komponenata i slično. To je razlog što je Google u novembru 2017. godine predstavio skup biblioteka pod nazivom "Android *Jetpack* komponente", koje će pomoći developerima u rješavanju navedenih problema [34]. One su kreirane na način da maksimalno iskoriste sav potencijal Kotlin programskog jezika i pomognu u razvoju kvalitetnih aplikacija koje je jednostavno održavati, nadgrađivati i testirati.

4.1. Navigacijska komponenta

Korisničke akcije često uzrokuju promjenu nekog dijela korisničkog interfejsa. Korisnički interfejsi koji sadrže veliki dio dinamičkih komponenata vrlo brzo postaju teški za održavanje. Android navigacijska komponenta je nastala s ciljem da olakša posao dizajniranja i implementiranja korisničkih akcija i njihovih efekata na korisničke interfejse. Navigacijska komponenta omogućava da se putem stanja, akcija i navigacijskog grafa definiše ponašanje korisničkog interfejsa u svakom trenutku. U nastavku će biti opisano korišenje navigation komponente.

Kako bi se koristila Navigation komponenta potrebno je imati minimalnu verziju Android Studio-a 3.3. Obzirom da komponenta ovisi od AndroidX, potrebno je uključiti njegove artefakte prilikom kreiranja projekta. U `build.gradle` potrebno je dodati:

```
// Kotlin
implementation("androidx.navigation:navigation-fragment-ktx:+")
implementation("androidx.navigation:navigation-ui-ktx:+")
```

```
// Feature module Support
implementation(“androidx.navigation:navigation-dynamic-features-
fragment:+””)

// Testing Navigation
androidTestImplementation(“androidx.navigation:navigation-
testing:+”)

// Jetpack Compose Integration
implementation(“androidx.navigation:navigation-compose:+”)
```

Navigacija se odvija unutar svih destinacija aplikacije, odnosno omogućen je prelaz između različitih destinacija korištenjem akcija. Navigacijski graf predstavlja XML resurs u kojem se nalaze sve destinacije i akcije. Graf predstavlja sve navigacijske putanje.

Navigacijasku komponentu čine tri dijela:
– Navigacijski graf
– `NavHost`
– `NavController`

Navigacijski graf predstavlja XML resurs koji na jednom mjestu čuva sve informacije koje su povezane sa navigacijom. Unutar navigacijskog grafa su smještene tzv. destinacije, kao i logičke veze između njih (tzv. akcije) koje označavaju smjer prelaska iz jedne destinacije u drugu. Pomenute destinacije mogu biti fragmenti, aktivnosti, pa čak i drugi navigacijski grafovi.

`NavHost` je prazan kontejner koji prikazuje odredišta s navigacijskog grafa. Navigacijska komponenta sadrži *default*-nu `NavHost` implementaciju, `NavHostFragment`, koja prikazuje odredišta fragmenata.

`NavController` je objekat koji upravlja navigacijom aplikacija unutar `NavHost`-a. `NavController` upravlja zamjenom odredišnog sadržaja u `NavHost`-u dok se korisnici kreću kroz aplikaciju.

Dok se korisnik kreće kroz aplikaciju, `NavController` od korisnika treba da dobije informaciju da želi da se kreće duž određene staze u grafu za navigaciju ili direktno do određenog odredišta, nakon čega `NavController` prikazuje odgovarajuće odredište u `NavHost`-u.

Navigacijska komponenta pruža niz pogodnosti, kao što su:

– Upravljanje transakcijama fragmenata

– Upravljanje *Up* i *Back* akcijama
– Pružanje standardiziranih resursa za animacije
– UI *patterne* za navigaciju, kao što su *Navigation Drawer* i *Bottom* navigacija
– *Safe Args* - gradle dodatak koji pruža sigurnost tipa prilikom navigacije i prosljeđivanja podataka između odredišta
– `ViewModel` podrška – moguće je proširiti `ViewModel` na navigacijski graf kako bi se dijelili podaci povezani s korisničkim interfejsom između odredišta grafa

Za pregled i uređivanje grafova za navigaciju može se koristiti *Navigation editor* Android Studio-a.

Prvi korak koji je potrebno obaviti da bi korištenje Navigacijske komponente bilo moguće jeste dodavanje zavisnosti unutar `build.gradle` datoteke aplikacije.

```
def nav_version = “2.3.5”
implementation “androidx.navigation:navigation-fragment-
ktx:$nav_version”
implementation “androidx.navigation:navigation-ui-ktx:$nav_
version”
```

Sljedeći korak je kreiranje `NavHost` kontejnera unutar *layout*-a aktivnosti. `NavHost` predstavlja prazni kontejner u koji će se dinamički ubacivati fragmenti koji se budu smjenjivali unutar aplikacije. Ukoliko aplikacija prati *Single Activity* pristup, to znači da sadrži samo jednu aktivnost i više fragmenata koji se smjenjuju i od kojih svaki obavlja određeni zadatak. `NavHost` kontejner je smješten unutar *layout*-a aktivnosti.

Primjer:

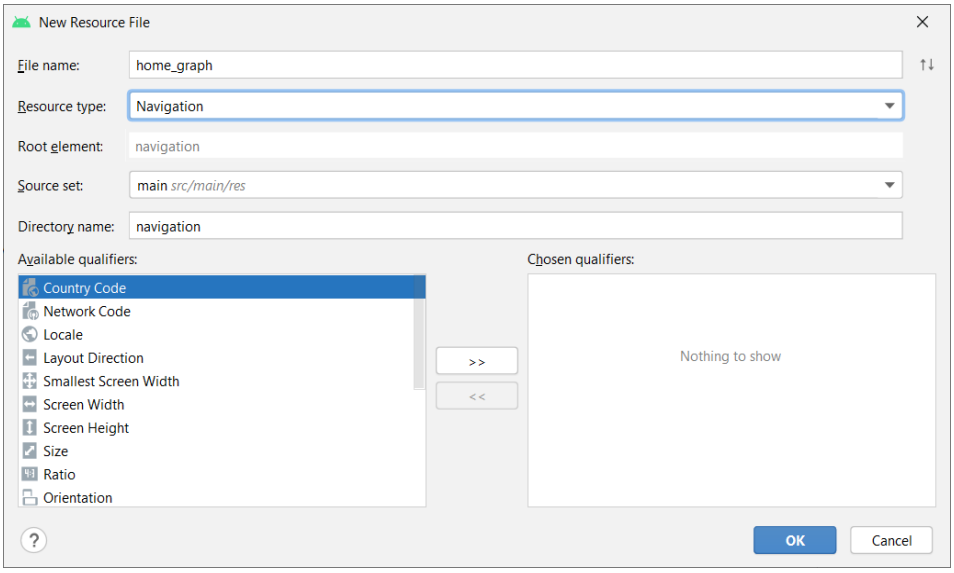
```
<androidx.fragment.app.FragmentContainerView
       android:id=”@+id/nav_host_fragment”
       android:layout_width=”match_parent”
       android:layout_height=”0dp”
       app:navGraph=”@navigation/home_graph”
       app:defaultNavHost=”true”
       android:layout_weight=”1”
       android:name=”androidx.navigation.fragment.
NavHostFragment”/>
```

`app:navGraph="@navigation/home_graph"` – povezuje `NavHost` kontejner sa odgovarajućim navigacijskim grafom unutar kojeg se nalazi informacija o tome koji se prvi fragment prikazuje korisniku na ekranu.

`app:defaultNavHost="true"` – označava da će kontejner imati ulogu `NavHost`-a koji će voditi računa o tome šta se desi kada korisnik klikne *back* dugme na svom uređaju.

`android:name="androidx.navigation.fragment.NavHostFragment"` – deklariše područje u kojem će se smjenjivati fragmenti.

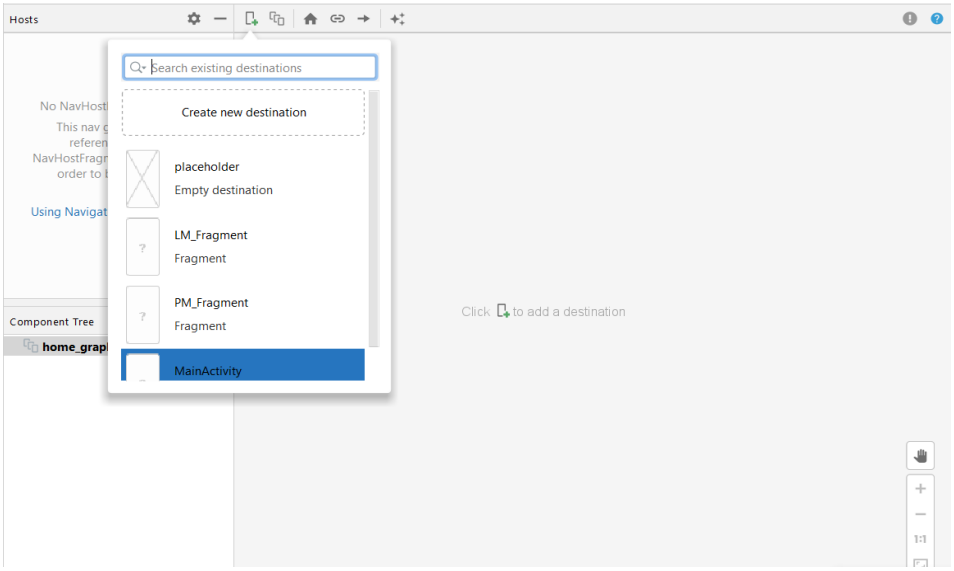
Kreiranje navigacijskog grafa podrazumijeva dodavanje XML resursa unutar *res* foldera projekta. Tip XML resursa je potrebno označiti da bude *Navigation*, a zatim mu dati ime koje će biti smisleno i koje će označavati da se radi o navigacijskom grafu (npr. home_graph, kako je navedeno u gornjem primjeru), što je predstavljeno na slici 12.



Slika 12. Kreiranje navigacijskog grafa

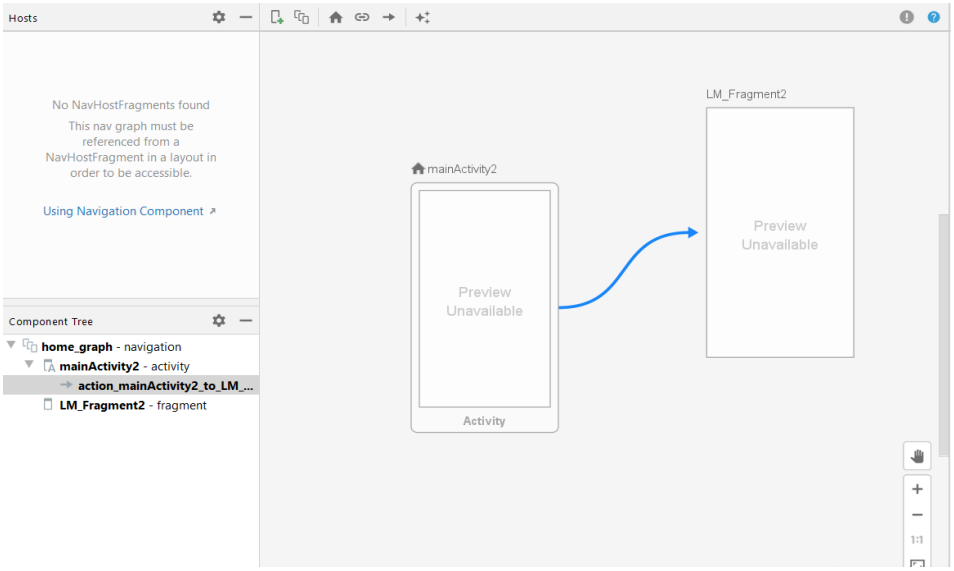
Nakon što je navigacijski graf kreiran, unutar Android Studio-a postoje dva načina dodavanja destinacija. Prvi način je standardno dodavanje elemanata u XML datoteku, a drugi način je pomoću *Navigation editor*-a koji ima intuitivan korisnički interfejs i koji omogućava dodavanje fragmenata na mnogo jednostavniji način, a sav potreban kôd unutar XML datoteke će biti automatski generisan. Klikom na ikonu sa znakom "plus" koja se

nalazi u vrhu ekrana, pojavljuje se lista svih fragmenata i aktivnosti unutar aplikacije. Prvi fragment koji se izabere ujedno postaje i početna destinacija posmatranog grafa, odnosno prvi ekran koji će se pojaviti kada korisnik pokrene aplikaciju (slika 13).



Slika 13. Proces dodavanja destinacije unutar navigacijskog grafa

Sljedeći korak je povezivanje fragmenata, odnosno kreiranje akcija. U *Navigation editor*-u se to obavlja na način da se klikne na fragment od kojeg veza počinje, a zatim se povuče mišem do odredišnog fragmenta, ili klikom na ikonu sa znakom "strelica desno" koja se nalazi u vrhu ekrana. Svaka akcija označava smjer iz kojeg fragmenta se prelazi u koji fragment (slika 14).



Slika 14. Kreiranje akcija koristeći Navigation editor

Sadržaj XML resursa/datoteke se automatski generiše, kao što je npr.:

```
<?xml version=”1.0” encoding=”utf-8”?>
<navigation xmlns:android=”http://schemas.android.com/apk/res/
android”
    xmlns:app=”http://schemas.android.com/apk/res-auto”
    android:id=”@+id/home_graph”
    app:startDestination=”@id/mainActivity2”>
    <activity
        android:id=”@+id/mainActivity2”
        android:name=”ba.unsa.etf.rma.helloworld.MainActivity”
        android:label=”MainActivity” >
        <action
            android:id=”@+id/action_mainActivity2_to_LM_Fragment2”
            app:destination=”@id/LM_Fragment2” />
    </activity>
    <fragment
        android:id=”@+id/LM_Fragment2”
        android:name=”ba.unsa.etf.rma.helloworld.LM_Fragment”
        android:label=”LM_Fragment” />
</navigation>
```

`NavController` je objekat koji je zadužen za zamjenu fragmenata unutar `NavHost` kontejnera. Svaki `NavHost` kontejner je povezan s jednim

`NavController` objektom koji predstavlja sponu između Navigacijskog grafa i `NavHost` kontejnera u kojem se fragmenti smjenjuju. Funkcija `findNavController()` vraća odgovarajući `NavController` objekat nad kojim se poziva funkcija `navigate()` kojoj se prosljeđuje `id` akcije koja označava prelazak iz jednog fragmenta u drugi.

Single Activity pristup u razvoju mobilnih aplikacija

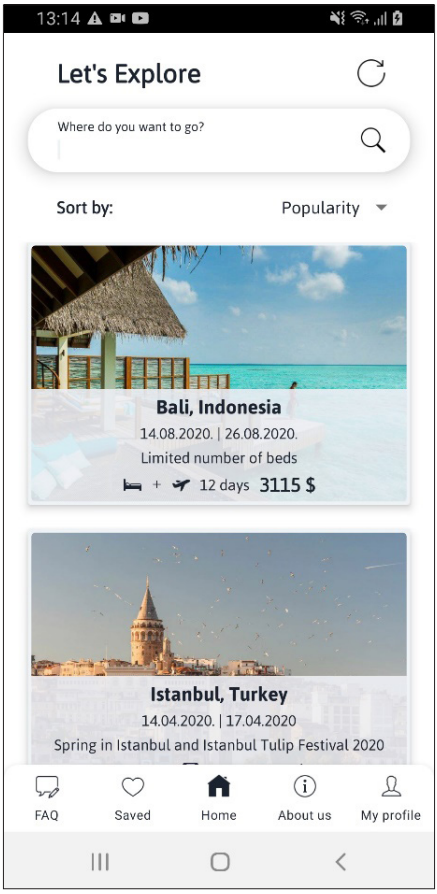
Aplikacija implementirana koristeći *Single Activity* pristup znači da aplikacija sadrži samo jednu aktivnost koja ima ulogu *host*-a i koja je zadužena za upravljanje fragmentima. Fragmente je poželjno kreirati na način da budu pogodni za višestruku upotrebu tako da bi trebalo izbjegavati da fragmenti direktno manipulišu drugim fragmentima. Korištenje jedne aktivnosti i više fragmenata u kombinaciji sa Android navigacijskom komponentom je praksa koju od 2019. godine preporučuje Google.

Neke od prednosti primjene ovakvog pristupa su sljedeće:

- **Rješava se problem nekonzistentnosti akvitnosti** – Aktivnost predstavlja dio Android *framework*-a što znači da su njene mogućnosti i njeno ponašanje povezani sa verzijama Androida. Upravo zbog toga se može desiti da neke nove funkcionalnosti neće biti podržane na starijim verzijama Androida. Kao rješenje tog problema često je potrebno koristiti nekoliko biblioteka kao što su `ActivityCompat`, `ActivityOptionsCompat` i slično, što uveliko povećava vrijeme razvoja i testiranja aplikacije. Ponašanje aktivnosti se može razlikovati i u zavisnosti od toga koji je operativni sistem instaliran na uređaju na kojem je aplikacija pokrenuta.
- **Bolje korisničko iskustvo** – Tranzicija između dvije aktivnosti zahtijeva promjenu čitavog ekrana, uključujući i *Toolbar*, što ne izgleda baš profesionalno. Bilo bi poželjno da *Toolbar* uvijek bude na istom mjestu, a da se ostali sadržaj dinamički mijenja. Također, nedosljedno ponašanje aktivnosti kada su u pitanju različiti uređaji i različite verzije operativnog sistema još je jedan od razloga da se razmisli o *Single Activity* principu.
- **Olakšava posao developerima** – Ukoliko je unutar aplikacije potrebno koristiti komponente kao što su donja ili bočna navigacija, a aplikacija sadrži više aktivnosti, onda je posao developera izuzetno otežan. Međutim, uz primjenu samo jedne aktivnosti u kombinaciji sa Android navigacijskom komponentom, ovaj proces je moguće znatno pojednostaviti.

Bottom navigation menu

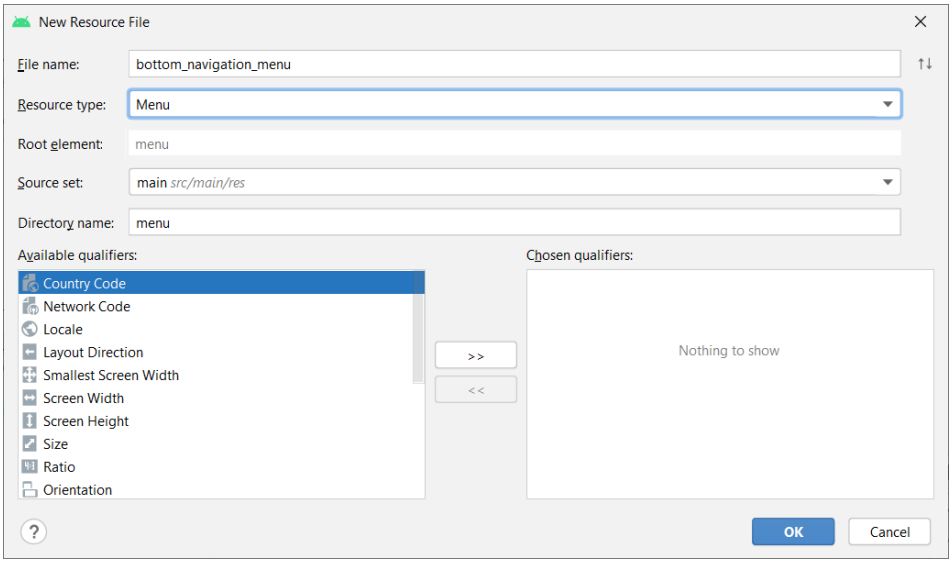
Moguće je implementirati da aplikacija sadrži i meni koji se nalazi na dnu ekrana i koji omogućava brz pristup najvažnijim ekranima aplikacije (npr. *FAQ*, *Saved*, *Home*, *About us*, *My profile*), kako je prikazano na slici 15.



Slika 15. Izgled početnog ekrana aplikacije

Donja navigacija je implementirana koristeći Navigacijsku komponentu. Čitav proces se sastoji od nekoliko jednostavnih koraka koje je potrebno pratiti.

Prvi korak jeste dodavanje XML resursa u *res* folder projekta. Tip resursa koji se dodaje treba biti *Menu (slika 16)*. Imenovanje je proizvoljno, ali je poželjno da bude intuitivno i da se na prvi pogled može zaključiti čemu je resurs namijenjen.



Slika 16. Dodavanje XML resursa za donju navigaciju

Sljedeći korak je dodavanje fragmenata koji će činiti donju navigaciju. Važno je napomenuti da `id` – evi fragmenata koji se dodaju u `bottom_navigation_menu.xml` resurs moraju odgovarati `id` – evima koji su postavljeni u navigacijskom grafu. Postavljanje ikone i naslova za svaki od fragmenata je opcionalno.

```
<?xml version=”1.0” encoding=”utf-8”?>
<menu xmlns:android=”http://schemas.android.com/apk/res/android”>
      <item android:id=”@+id/faq”
            android:icon=”@drawable/menu_faq_selector”
            android:title=”@string/menu_faq” />
      <item android:id=”@+id/saved_trips”
            android:icon=”@drawable/menu_favourites_selector”
            android:title=”@string/menu_saved_trips” />
      <item android:id=”@+id/home”
            android:icon=”@drawable/menu_home_selector”
            android:title=”@string/menu_home” />
      <item android:id=”@+id/about_us”
            android:icon=”@drawable/menu_about_us_selector”
            android:title=”@string/menu_about_us” />
      <item android:id=”@+id/my_profile”
            android:icon=”@drawable/menu_user_selector”
            android:title=”@string/menu_my_profile” />
</menu>
```

Posljednji korak je dodavanje `BottomNavigationView` elementa, unutar *layout*-a glavne aktivnosti, u koji će se dinamički ubacivati Fragmenti.

```
<com.google.android.material.bottomnavigation.BottomNavigationView
       android:id=”@+id/bottom_navigation”
       android:layout_width=”match_parent”
       android:layout_height=”wrap_content”
       android:background=”@drawable/rounded”
       android:elevation=”2dp”
       app:itemIconSize=”20dp”
       app:itemIconTint=”@color/bottom_nav_color”
       app:itemTextColor=”@color/bottom_nav_color”
       app:labelVisibilityMode=”labeled”
       app:menu=”@menu/bottom_navigation_menu” />
```

Da bi prethodno kreirani element bio spojen sa navigacijskim grafom, posljednje što je potrebno uraditi jeste povezati meni sa navigacijskim grafom:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
menuInflater.inflate(R.menu.bottom_navigation_menu, menu)
       return true
}
```

Funkcija onCreateOptionsMenu(menu: Menu?) označava koji resurs iz *menu* foldera će se koristiti u ovom slučaju.

Safe – args plugin

Prelazak sa jednog ekrana na drugi, ponekad može zahtijevati i slanje određenih podataka. Navigacijska komponenta podržava slanje podataka koristeći *Bundle*, koji predstavlja mehanizam poznati svim Android developerima. Međutim, pojavljuje se i novi sistem, a to je korištenje tzv. *SafeArgs*. *SafeArgs* je gradle plugin koji omogućava da se unutar navigacijskog grafa unesu sve potrebne informacije o argumentu kojeg je potrebno proslijediti. Ovaj plugin omogućava i *type – safety* za argumente, što znači da će se prije slanja provjeriti da li je tip agumenta ispravan, te ako nije, greška će se pojaviti već u procesu kompajliranja.

Prvo što je potrebno uraditi da se omogući korištenje *SafeArgs* plugin-a je dodavanje zavisnosti u “dependencies” blok `build.gradle` datoteke projekta.

```
def nav_version = “2.3.5”
```

```
classpath “androidx.navigation:navigation-safe-args-gradle-
plugin:$nav_version”
```

Zatim se u `build.gradle` datoteku aplikacije dodaje sljedeća linija kôda:

```
apply plugin: “androidx.navigation.safeargs.kotlin”
```

Na ovaj način je omogućeno korištenje *SafeArgs* plugin-a za slanje potrebnih podataka iz jednog fragmenta u drugi.

Nakon što se omogući *Safe Args*, plugin generiše kôd koji sadrži klase i metode za svaku akciju koja je definisana. *Safe Args* takođe generiše klasu za svako odredište, a to je odredište iz kojeg akcija potiče. Generisano ime klase je kombinacija naziva klase odredišta i riječi “*Directions*”. Na primjer, ako je odredište imenovano s `Fragment1`, generisana klasa se naziva `Fragment1Directions`. Generisana klasa sadrži statičku metodu za svaku akciju definisanu u odredištu. Ova metoda preuzima sve parametre definisane akcije kao argumente i vraća `NavDirections` objekt koji se može proslijediti `navigate()` metodi.

4.2. Lifecycle aware komponente

Skoro svaka komponenta koja se koristi u Androidu ima odgovarajući životni ciklus. Do sada su developeri morali voditi računa da prilikom obavljanja različitih zadataka na adekvatan način reaguju na promjene stanja životnog ciklusa kako ne bi došlo do curenja memorije u aplikaciji ili do potpunog prestanka njenog rada. To ponekad može biti komplikovan proces, jer dosta različitih scenarija treba uzeti u razmatranje. Primjer jednog takvog scenarija slijedi u nastavku:

Pretpostavimo da je potrebno obaviti neki zadatak koji ima duže vrijeme izvršavanja. Funkcija za obavljanje tog zadatka se pozove u `onStart()` metodi aktivnosti ili fragmenta. U međuvremenu, korisnik zatvori aplikaciju, tako da ona ode u pozadinu. Ali prije nego što se aplikacija zatvori, poziva se `onStop()` metoda. Međutim, šta ako obavljanje pomenutog zadatka još uvijek traje? Doći će do tzv. *race – condition* stanja u kojem `onStop()` metoda može završiti prije `onStart()` metode. To naravno uglavnom uzrokuje različite probleme ukoliko se ne predvidi na vrijeme [13]. Upravo zbog toga, Google se zalaže za korištenje *Lifecycle* Android *Jetpack* komponenti.

Lifecycle aware komponente su komponente koje su svjesne životnog ciklusa nekih drugih komponenti, kao što su aktivnosti i fragmenti, i poduzimaju akcije kao odgovor na promjenu stanja životnog ciklusa tih komponenti. Svrha njihovog korištenja je kreiranje aplikacija koje su bolje organizovane, lakše predvidljive, jednostavnije za održavanje i nadgrađivanje [14].

Lifecycle aware komponente koje će biti objašnjene u nastavku su `ViewModel` i `LiveData`.

4.2.1. ViewModel

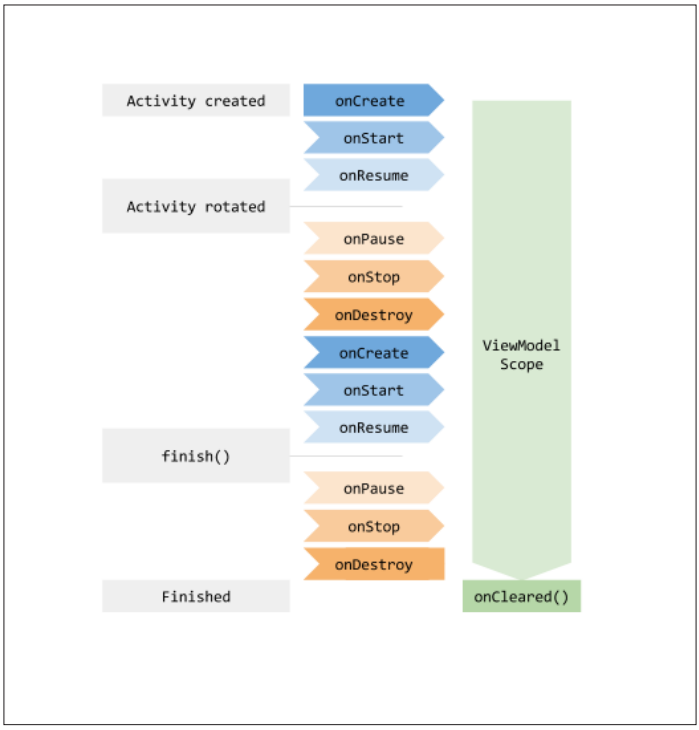
Android *framework* je zadužen za upravljanje životnim ciklusom komponenti korisničkog interfejsa (*User Interface* - UI). *Framework* ima zadatak da kreira komponentu i da je uništi kada za to dođe vrijeme. Kada dođe do završetka životnog ciklusa određene komponente, svi podaci koji su povezani s njom će biti izgubljeni. Na primjer, ukoliko se na ekranu prikazuje lista korisnika dohvaćena koristeći neki izvor i desi se određena konfiguracijska promjena (kao što je rotiranje ekrana) komponenta će se morati rekreirati i bit će potrebno ponovo dohvatiti podatke za formiranje liste. Naravno, cilj je izbjeći ovo ponovno dohvatanje podataka. Jedno od mogućih rješenja je korištenje `onSaveInstanceState()` metode. Međutim, ona je pogodna samo za spremanje manje količine podataka, tako da u slučaju velikih listi i bitmapa neće biti efikasna [15].

Još jedan scenarij koji developerima često pravi različite vrste problema je asinhrono pozivanje funkcija koje se izvršavaju poprilično dugo vremena prije nego što vrate očekivani rezultat. UI komponenta bi trebala voditi računa o tome da svi pozivi, koji su napravljeni asinhrono, budu prekinuti onog trenutka kada dođe do kraja životnog ciklusa komponente, kako ne bi došlo do curenja memorije ili drugih vrsta problema. UI komponente su prvenstveno namijenjene da prikažu podatke korisniku, da reaguju na korisnikove akcije i da komuniciraju sa operativnim sistemom u slučaju zahtjeva za odobravanjem određene funkcionalnosti i slično.

Ako je cilj razviti aplikaciju koju je jednostavno održavati, testirati i nadgrađivati, UI komponente ne bi trebale sadržavati nikakvu logiku koja se odnosi na komunikaciju sa izvorima za dobavljanje podataka. Upravo zbog toga je kreirana *lifecycle aware* Android arhitekturalna komponenta `ViewModel` koja je odgovorna za pripremu i formatiranje podataka koji će se prikazati korisniku. Objekti `ViewModel` klase su otporni na konfiguracijske

promjene, što znači da se podaci unutar njih neće izgubiti ako se desi neka konfiguracijska promjena kao što je rotacija ekrana.

Životni ciklus objekta bilo koje `ViewModel` klase je ograničen trajanjem životnog ciklusa UI komponente unutar koje je instanciran. Na slici 17. je prikazan životni ciklus `ViewModel` objekta u odnosu na životni ciklus aktivnosti.



Slika 17. Životni ciklus `ViewModel` objekta[32]

Još jedna važna uloga `ViewModel` klase ogleda se u činjenici da se vrlo često koristi za komunikaciju između aktivnosti ili fragmenata. Zamislimo scenarij u kojem se na jednom ekranu nalaze dva fragmenta. Prvi fragment sadrži listu npr. kontakata, a drugi sadrži detaljne informacije o kontaktu kojeg korisnik izabere. Ostvarivanje komunikacije između ova dva fragmenta nije trivijalan zadatak. Inače bi se mogao riješiti upotrebnom različitih interfejsa. Međutim, koristeći `ViewModel` komponentu ovaj problem može biti riješen na vrlo jednostavan i elegantan način.

[32] Slika preuzeta sa: https://developer.android.com/topic/libraries/architecture/viewmodel

4.2.2. LiveData

`LiveData` je klasa namijenjena za čuvanje podataka koju je moguće osluškivati (eng. *observe*) i koja ima osobinu da je svjesna životnog ciklusa drugih komponenti kao što su aktivnosti, fragmenti, servisi i slično. Ova svjesnost joj omogućava da obavještava samo aktivne osluškivače o promjenama svog stanja. `LiveData` klasa podrazumijeva da je osluškivač u aktivnom stanju samo ako mu je trenutno stanje: STARTED ili RESUMED. Sva ostala stanja se smatraju neaktivnim [16].

Prednosti korištenja `LiveData` klase su [16]:

- **Osigurava da stanje UI komponenti odgovara stanju podataka** – `LiveData` prati tzv. "Observer pattern". Svaki put kada dođe do promjene stanja `LiveData` objekta, `Observer` objekat primijeti tu promjenu i izvrši se ažuriranje UI komponenti koristeći nove podatke.
- **Nema padanja aplikacije zbog zaustavljanja aktivnosti** – Ako je osluškivač u neaktivnom stanju, jednostavno neće primati `LiveData` događaje.
- **Nema curenja memorije** – Objekti `Observer` klase su vezani za `Lifecycle` objekte i automatski se uništavaju nakon što se životni ciklus tih objekata završi.
- **Nema više ručnog upravljanja životnim ciklusom komponenti** – `LiveData` komponenta je svjesna životnog ciklusa komponente koja je osluškuje. Zbog toga neaktivni osluškivači neće biti ažurirani. Developer o tome ne mora voditi računa.
- **Ispravno rukovanje konfiguracijskim promjenama** – Ukoliko dođe do ponovnog kreiranja aktivnosti ili fragmenta usljed konfiguracijske promjene (kao što je rotiranje uređaja) podaci neće biti izgubljeni, jer se dobijaju iz `LiveData` objekta koji se nalazi unutar instance određene `ViewModel` klase.
- **Jednostavno dijeljenje resursa** – Ukoliko postoji neki servis koji se koristi na više mijesta unutar aplikacije, moguće ga je jednostavno smjestiti untar `LiveData` objekta i na taj način ga više različitih komponenti može osluškivati.

Da bi određena komponenta (aktivnost, fragment, servis, itd.) bila u mogućnost reagovati na promjene stanja `LiveData` objekta, prvo što je potrebno uraditi jeste kreirati objekat `Observer` klase koji će se povezati sa `LiveData` objektom i koji će obavijestiti UI komponentu ukoliko dođe do

promjene stanja posmatranog objekta. Unutar `Observer` klase kreirana je `onChanged()` metoda koja će se aktivirati u dva slučaja:

1. Kada se desi promjena podataka koje čuva `LiveData` objekat
2. Ako je proces osluškivanja započet, a unutar `LiveData` objekta se već nalaze neki podaci

Povezivanje objekta `Observer` klase sa objektom `LiveData` klase se obavlja koristeći funkciju `observe()` nad objektom `LiveData` klase koji se želi osluškivati. Metoda `observe()` očekuje da joj se proslijedi `LifecycleOwner` objekat kao i kreirani objekat `Observer` klase kako bi komponenta mogla na adekvatan način odgovoriti na promjene stanja [13].

Također, postoji i metoda `observeForever(Observer)` koja se poziva nad instancom `LiveData` klase i kojoj nije potrebno prosljeđivati `LifecycleOwner` objekat, ali će se onda podrazumijevati da je komponenta uvijek aktivna i slat će joj se obavijesti o promjeni stanja sve dok se to ručno ne zaustavi pozivom metode `removeObserver(Observer)` [16].

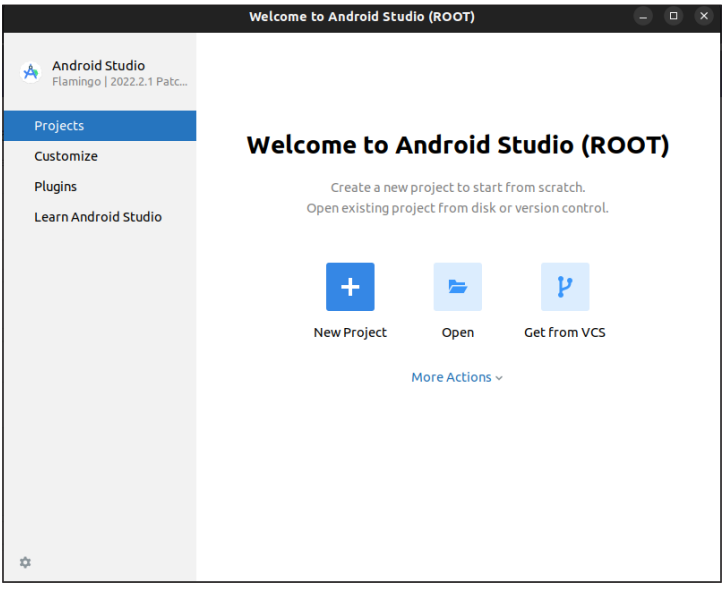
Važno je napomenuti da više različitih aktivnosti, fragmenta, servisa i sl. mogu osluškivati promjene istog `LiveData` objekta. Svaki put kada do promjene dođe, svi osluškivači koji su pretplaćeni i koji su aktivni će moći primijetiti novonastalu situaciju.

`LiveData` je omotač koji može sadržavati razne vrste podataka, uključujući i liste. Međutim, jedna važna karakteristika ove klase je da podaci unutar nje ne mogu biti promijenjeni naknadno zato što ne sadrži javne metode `setValue(T)` i `postValue(T)`. U slučaju da je potrebno naknadno promijeniti podatke, koristi se drugi tip klase – `MutableLiveData` koji se jedino razlikuje po tome što su mu metode `setValue(T)` i `postValue(T)` javne. Ove arhitekturalne komponente se obično koriste u kombinaciji sa `ViewModel` arhitekturalnom komponentom [16].

5.
DODATAK – PRAKTIČNI PRIMJER

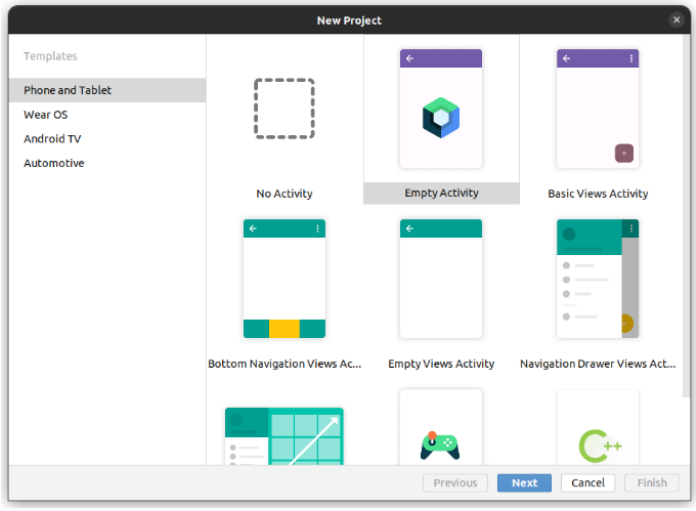
5.1. Kreiranje prve aplikacije

Kreirati novi Android projekat koristeći Android Studio:



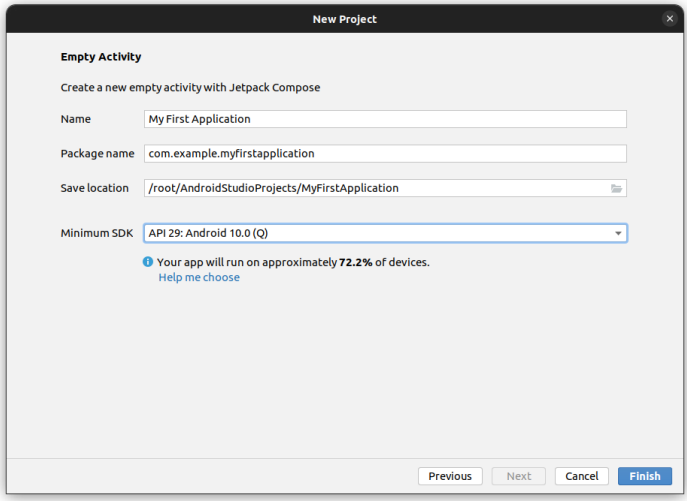
Snimak zaslona 14. Početni ekran Android Studio

Izabrati `Empty Activity`:



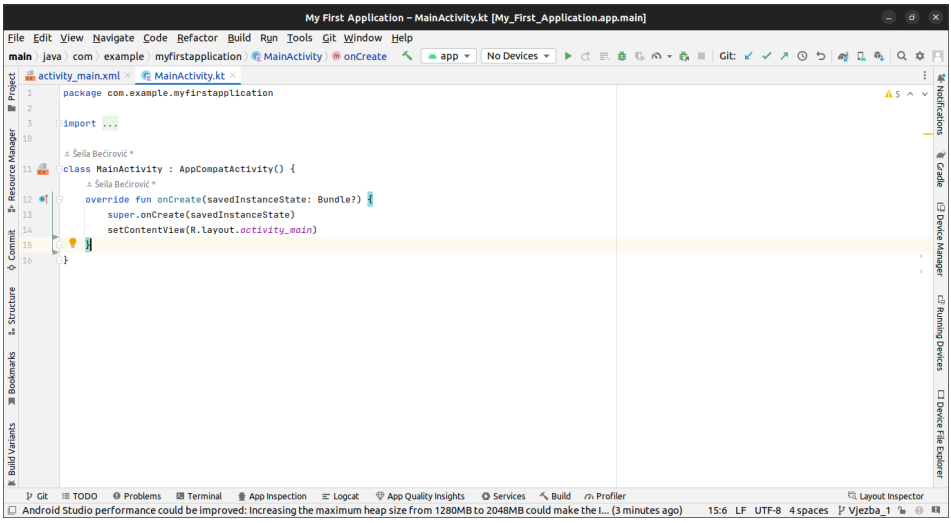
Snimak zaslona 15. Odabir template-a projekta

Konfigurisati projekat. Minimalni SDK postaviti na noviji najzastupljeniji API ili niže na osnovu vlastitog uređaja:



Snimak zaslona 16. Konfiguracija projekta

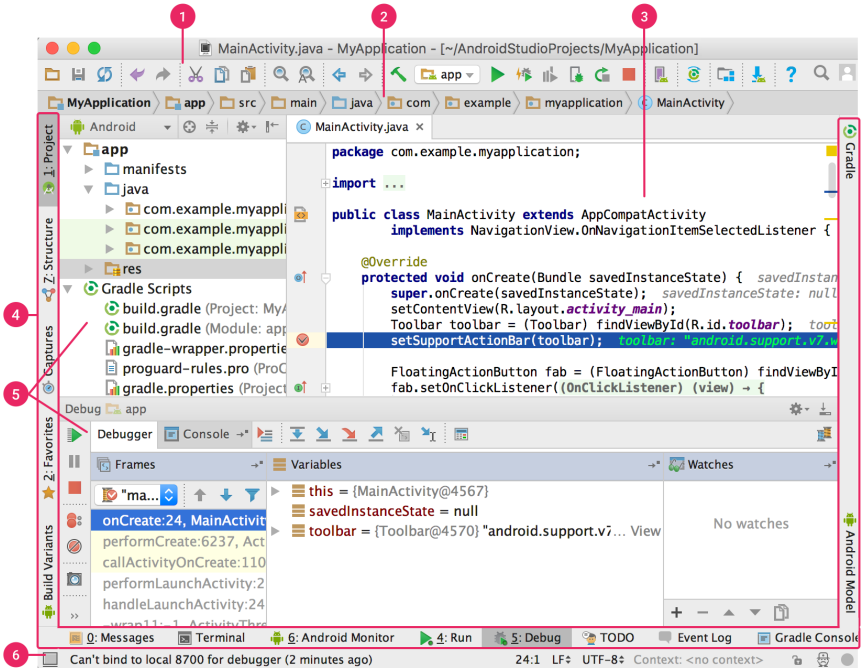
Sačekati da se projekat kreira. Nakon kreiranja, projekat će imati kreirane osnovne stavke: odgovarajuću strukturu s manifest-om, glavnom aktivnoti i *layout*-om.



Snimak zaslona 17. Početni ekran novokreiranog projekta

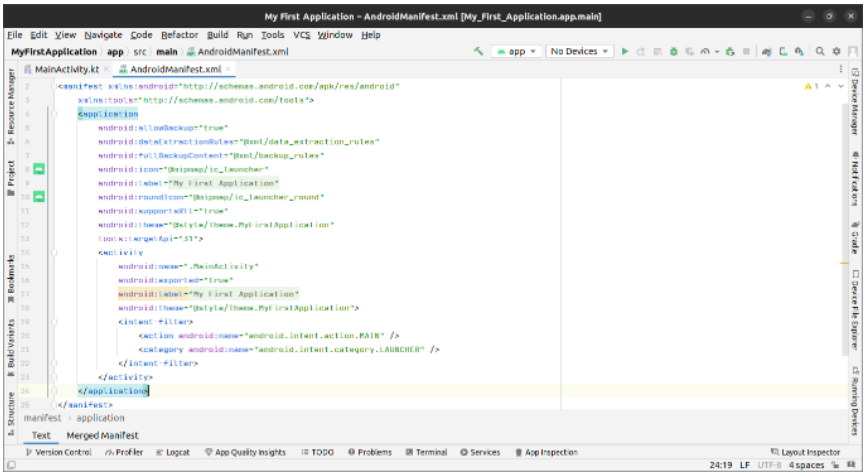
Osnovni elementi prozora Android Studio-a su:

1. Toolbar - omogućava izvršavanje raznih tipova akcija, kao što su pokretanje aplikacije i Android alata.
2. Navigacijski bar - omogućava prolaz kroz projekat i otvorene datoteke za uređivanje. Kompaktni pregled je dostupan u *Project* prozoru.
3. Editor - služi za kreiranje i uređivanje kôda. Postoje dva tipa editor-a za kôd i za *layout*.
4. Toolbar - omogućava prikazivanje određenih alata.
5. Alati projekta - podrazumijevaju: menadžment projekta, pretraga, verzija, terminal, debug, itd.
6. Statusna traka - prikazuje trenutni status projekta i IDE.



Slika 18. Android Studio IDE[33]

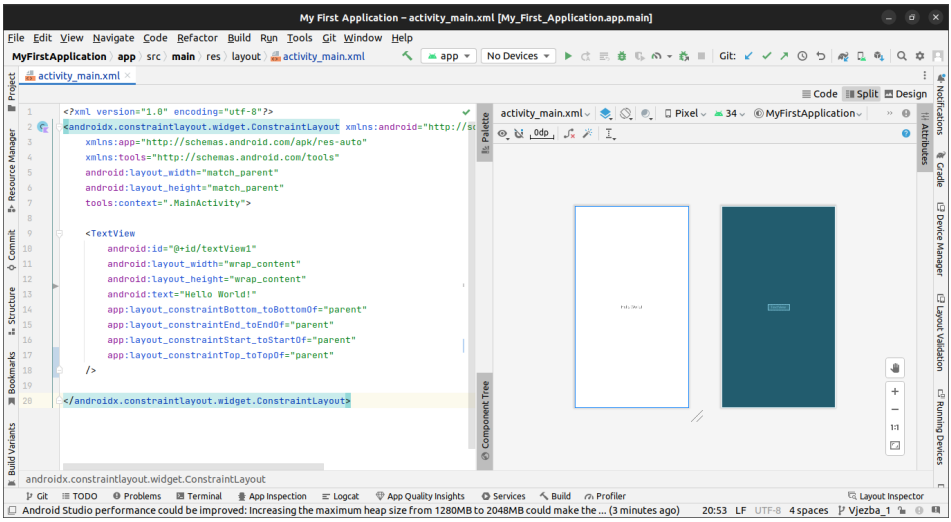
Struktura `AndroidManifest.xml` dokumenta u kojem je definisana aplikacija i njene osnovne komponente je:



Snimak zaslona 18. AndroidManifest.xml

[33] Slika preuzeta sa: https://www.teknotut.com/android-studio-tutorial-create-the-first-android-application/

Izgled *LayoutEditor*-a koji ima dva vida prikaza koji se odnose na uređivanje XML dokumenta i direktno uređivanje korisničkog interfejsa:



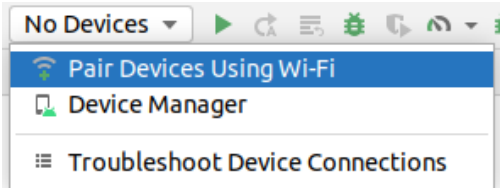
Snimak zaslona 19. Layout početne aktivnosti

5.2. Pokretanje aplikacije

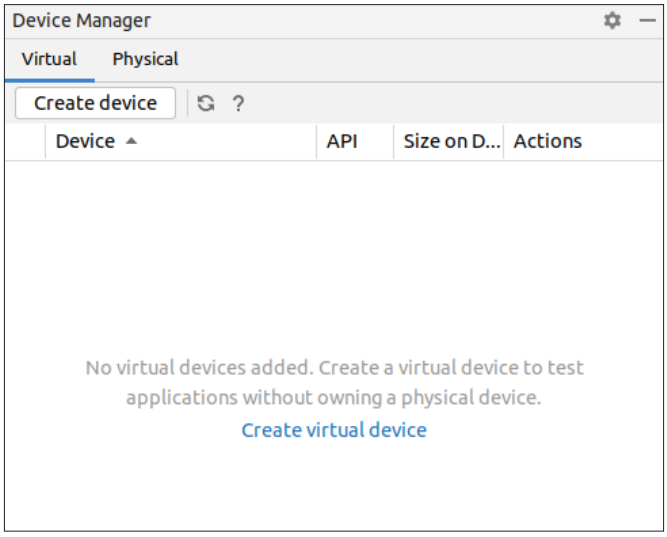
Aplikacija se može pokrenuti na vlastitom uređaju pri čemu je potrebno uključiti *usb debugging*, za čije uključenje treba omogućiti razvojne opcije (*Developer options*) klikom 7 puta na *version number* u informacijama o telefonu. Pored ovog načina, aplikacija se može pokretati na emulatoru. U slučaju pokretanja na emulatoru, preporuka je da se isti ne gasi, te da se samo aplikacija iznova pokreće na njemu.

Koraci instalacije jednog virtuelnog uređaja, pri čemu u BIOS-u opcije virtualizacije moraju biti uključene, su:

1. U *Devices* opciji pokrenuti *Device manager* i odabrati opciju za kreiranje novog uređaja.

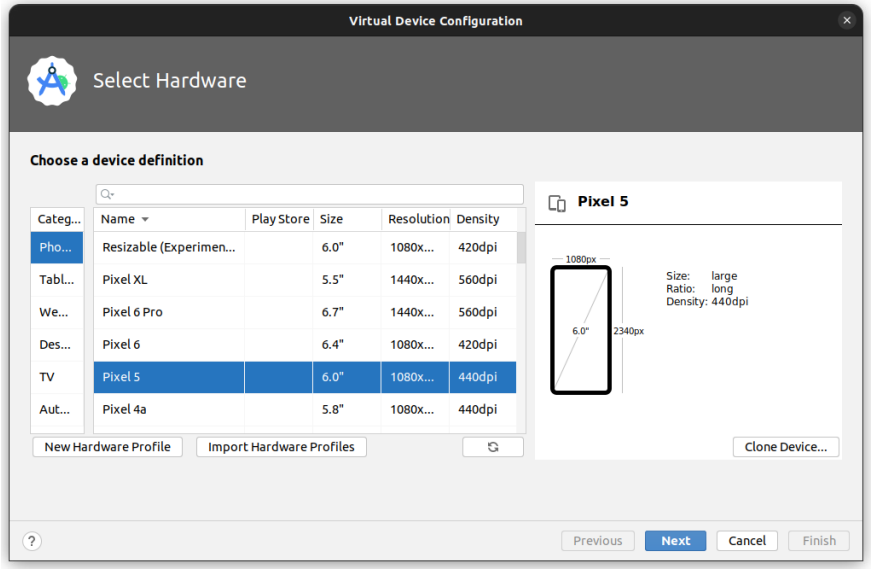


Snimak zaslona 20. Devices opcija



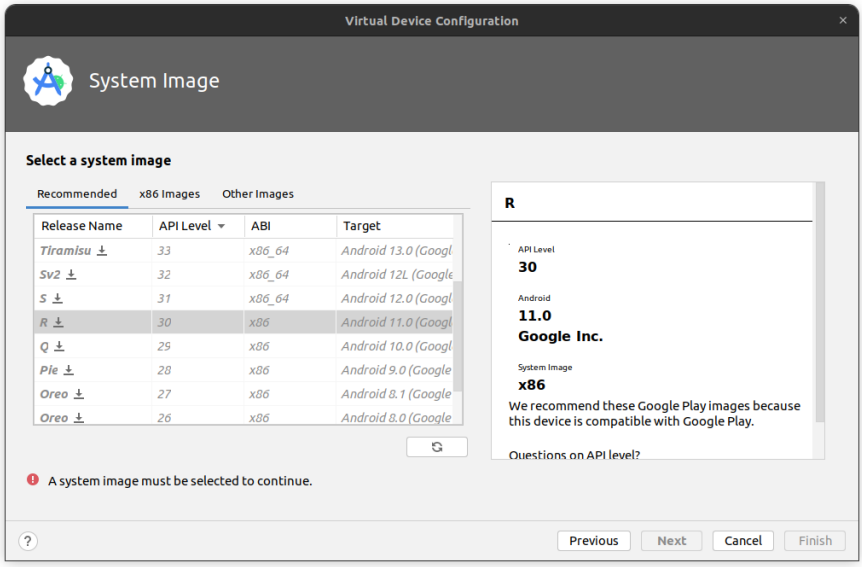
Snimak zaslona 21. Device Manager

2. Odabrati odgovarajući model uređaja



Snimak zaslona 22. Odabir hardvera novog virtuelnog uređaja

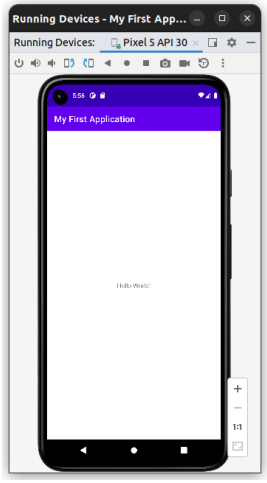
3. Odabrati operativni sistem



Snimak zaslona 23. Odabir OS-a virtuelnog uređaja

4. Sačekati da se instalacija završi

Nakon što se instalacija završi, pokretanje aplikacije treba biti omogućeno. Pokrenuti je na odgovarajućem uređaju. Emulator će se pokrenuti i nakon određenog vremena će se aplikacija *build*-ati i instalirati na uređaj. Emulator se trenutno nalazi kao dio Toolbar-a.



Snimak zaslona 24. Pokrenuta aplikacija na emulatoru

Ako se želi odvojiti uređaj radi lakšeg pregleda, potrebno je izabrati odgovarajuću opciju u *Settings/View Mode*.

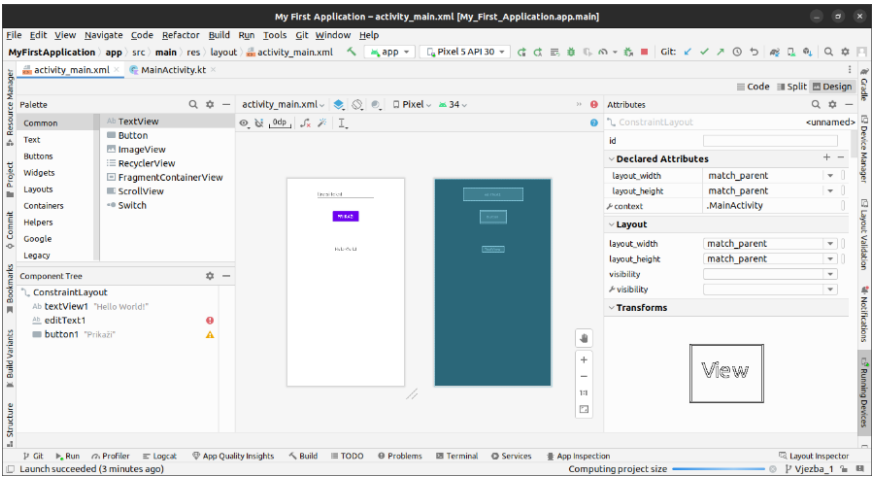
Prva aplikacija je uspješno pokrenuta.

5.3. Korištenje ulaznih kontrola

Potrebno je omogućiti unos teksta koji će s klikom na dugme biti prikazan u odgovarajućem `TextView`-u. Korištenjem editora kreirati dodatne dvije kontrole:

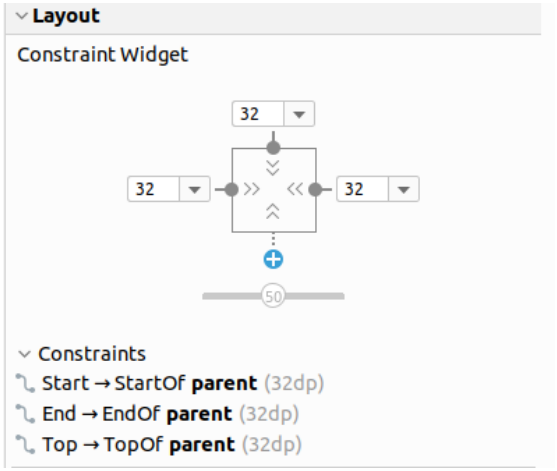
- `EditText` s ID: `editText1`
- `Button` s ID: `button1`

Postojeći `TextView` će imati ID: `textView1`. Izgled odgovarajućeg korisničkog interfejsa je:



Snimak zaslona 25. Kreiranje korisničkog interfejsa

Napomena: Obzirom da je *default*-ni *layout* `ConstraintLayout`, za sve stavke treba definisati odgovarajuća ograničenja kako je prikazano na:



Snimak zaslona 26. Opcije constraint layout-a

Nakon što su komponente kreirane, klasa glavne aktivnosti trenutno glasi:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Ovaj kôd znači da će se prilikom kreiranja aktivnosti prikazati odgovarajući *layout* `activity_main`. *Layout*-u se pristupa kroz klasu `R` koja označava sve interne resurse aplikacije. Resursi Androida su dostupni kroz `android.R`.

U `onCreate` funkciji dohvatiti referencu na dugme i dodijeliti mu osluškivač na klik.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //Dohvatit cemo referencu na view button-a preko id-a
        val button = findViewById<Button>(R.id.button1);
        //Definisat cemo akciju u slucaju klik akcije
        button.setOnClickListener {
           //akcije nakon klik-a
```

```
        }
    }
```

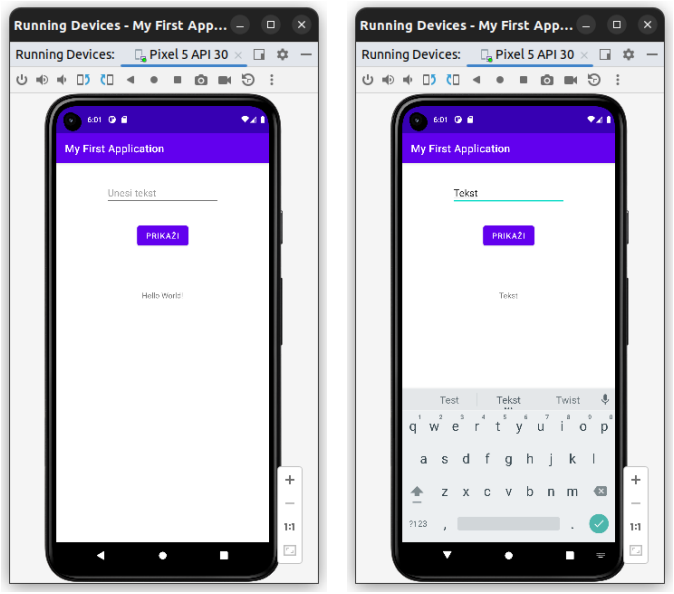
Kreirati privatnu funkciju unutar klase koja će dohvatiti tekst iz `EditText`-a i postaviti na `TextView`.

```
private fun showMessage() {
    // Pronaci cemo nas edit text i text view na osnovu id-a
    val editText = findViewById<EditText>(R.id.editText1)
    val textView = findViewById<TextView>(R.id.textView1)
    // Tekst cemo prebaciti u varijablu
    val message = editText.text.toString()
    // Postavimo tekst
    textView.text = message
}
```

Cjelokupni kôd sada je:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //Dohvatit cemo referencu na view button-a preko id-a
        val button = findViewById<Button>(R.id.button1);
        //Definisat cemo akciju u slucaju klik akcije
        button.setOnClickListener {
            showMessage()
        }
    }

    private fun showMessage() {
        // Pronaci cemo nas edit text i text view na osnovu id-a
        val editText = findViewById<EditText>(R.id.editText1)
        val textView = findViewById<TextView>(R.id.textView1)
        // Tekst cemo prebaciti u varijablu
        val message = editText.text.toString()
        // Postavimo tekst
        textView.text = message
    }
}
```

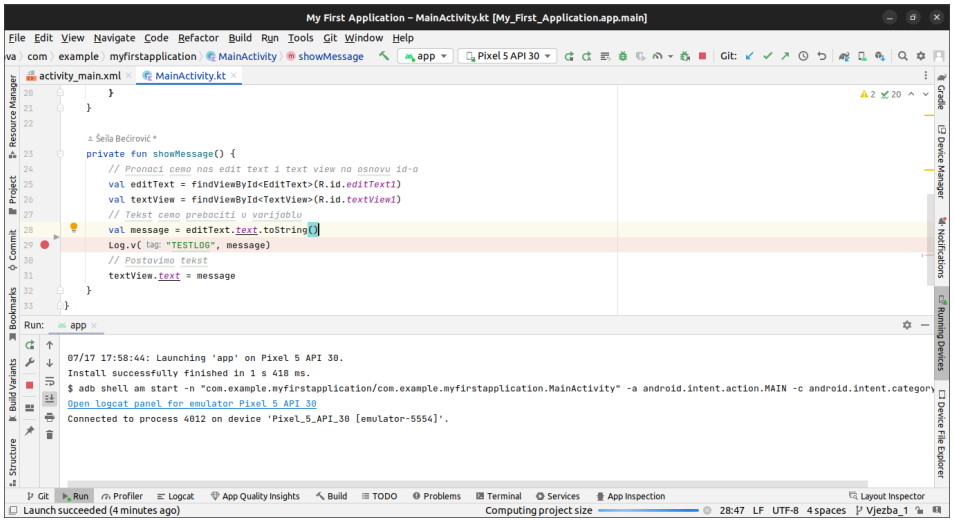
Nakon pokretanja i testiranja, aplikacija će uraditi traženo, kao što je prikazano na slijedećoj slici:



Snimak zaslona 27. Rad aplikacije

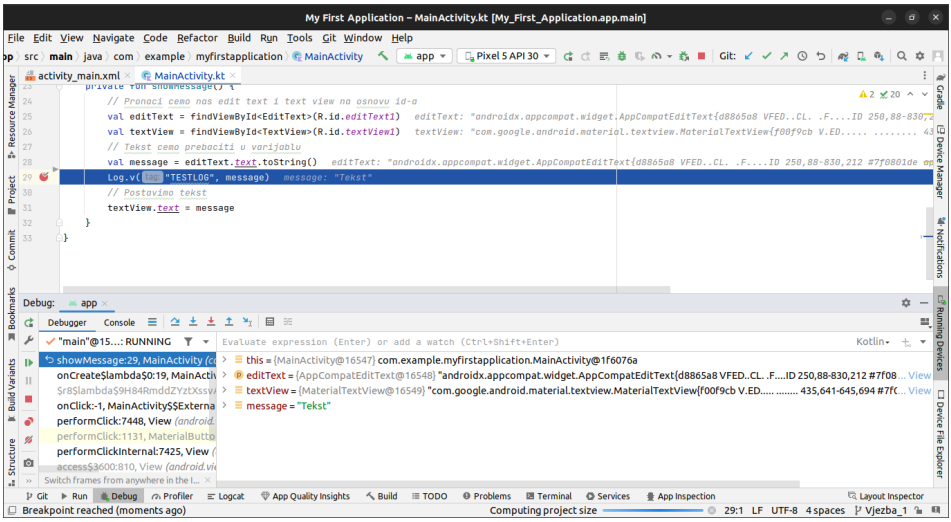
5.4. Debug aplikacije i logiranje

Postaviti *breakpoint* na aplikaciji i izvršiti logiranje teksta iz `EditText`-a. Slijedi prikaz dodavanja *breakpoint*-a, kako izgleda odgovarajuća linija logiranja podataka i gdje se pokreće *debug* mode.



Snimak zaslona 28. Postavljanje breakpoint-a

Nakon što se izvrše odgovarajuće akcije na aplikaciji, *breakpoint* će zaustaviti izvršavanje. Otvorit će se `DEBUG` tab u kojen se vidi trenutno stanje aplikacije. Izvršavanje se može nastaviti liniju po liniju ili do samog kraja.

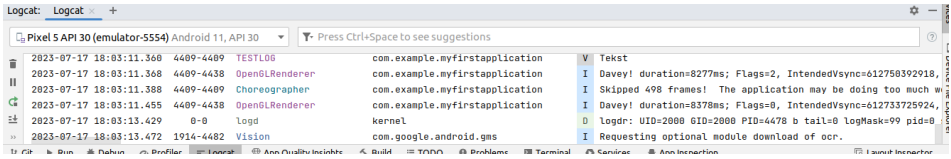


Snimak zaslona 29. Debugiranje aplikacije

Logiranje se vrši korištenjem `Logcat`-a. Logovi se mogu vidjeti u odgovarajućem `Logcat` tab-a. Postoje sljedeće vrste logova:

- Log.e(String, String) (error)
- Log.w(String, String) (warning)
- Log.i(String, String) (information)
- Log.d(String, String) (debug)
- Log.v(String, String) (verbose)

Prvi parametar predstavlja TAG, a drugi poruku. Odgovarajući log aplikacije ima sljedeći izgled:

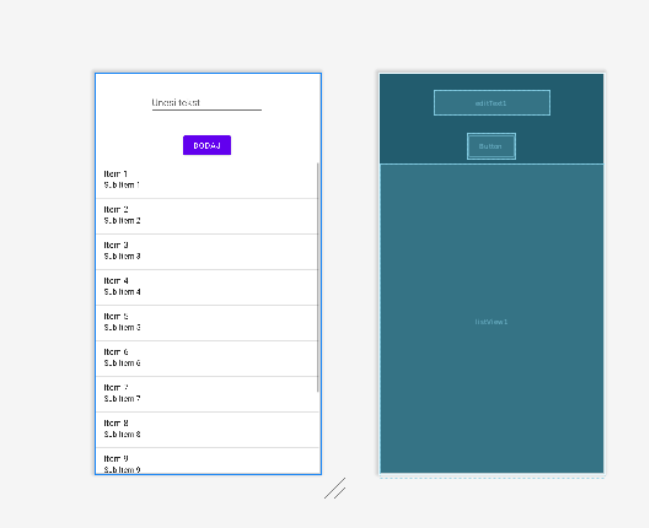


Snimak zaslona 30. Pregled log-ova

Omogućena je i detaljna pretraga logcat-a.

5.5. Resursi i napredni layout

Kako bi se omogućilo dodavanje elementa u listu korištenjem Android adaptera potrebno je u prvom koraku, umjesto `TextView`-a u aplikaciji, dodati ListView s ID `listview1`.



Snimak zaslona 31. Kreiranje korisničkog interfejsa

Napomena: U slučaju da se elementi pomijeraju prikazivanjem tastature, dodati `ListView` u `FrameLayout`. Pored toga, potrebno je dodati slijedeću liniju u `AndroidManifest.xml` u `activity` tag-u:

```
android:windowSoftInputMode=”adjustPan|adjustResize”
```

Kôd *layout*-a je:

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

    <EditText
        android:id=”@+id/editText1”
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="32dp"
        android:ems="10"
        android:hint="Unesi tekst"
        android:inputType="textPersonName"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="32dp"
        android:layout_marginBottom="8dp"
        android:text="Dodaj"
        app:layout_constraintBottom_toTopOf="@+id/listView1"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editText1" />

    <ListView
        android:id="@+id/listView1"
        android:layout_width="409dp"
        android:layout_height="572dp"
        android:layout_marginStart="32dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="32dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/button1" />

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

Svi elementi navedeni u kôdu *layout*-a će biti potrebni, te će se definisati unutar klase s ključnom riječi `lateinit`, zato što se mogu inicijalizirati tek nakon što se postavi *layout*. Ova ključna riječ se najčešće koristi za `View` elemente koji se naknadno moraju inicijalizirati. Olakšava korištenje

varijabli, odnosno ne mora se koristiti `??` ili `!!` koji su potrebni za `null` vrijednosti. Pored ovog načina, postoje i drugi korištenjem ključne riječi `lazy` ili kreiranjem `get()` metode, te delegata. Međutim, ovi načini nisu prilagođeni radu s `Fragment` ili `Activity`.

Pored `View`, potrebno je definisati adapter i listu vrijednosti, pri čemu će biti korištena konstrukcija `arrayListOf` ili `mutableListOf`.

```
private lateinit var listView: ListView
private lateinit var editText: EditText
private lateinit var button: Button
// moze se koristiti i mutableListOf
private val  listaVrijednosti = arrayListOf<String>()
private lateinit var adapter : ArrayAdapter<String>
```

U nastavku, incijalizirati odgovarajuće elemente i dodijeliti listi adapter, te *listener* dugmetu. Korišteni adapter je `ArrayAdapter` Androida kojem se proslijeđuje *context*, *layout* koji se želi koristiti i lista vrijednosti. *Layout* koji se proslijeđuje je interni Android *layout* elementa liste.

```
button = findViewById(R.id.button1);
editText = findViewById(R.id.editText1)
listView = findViewById(R.id.listView1)
//Koristi se androidov layout
adapter = ArrayAdapter(this, android.R.layout.simple_list_
item_1, listaVrijednosti)
listView.adapter=adapter
//Definisat cemo akciju u slucaju klik akcije
button.setOnClickListener {
    addToList()
}
```

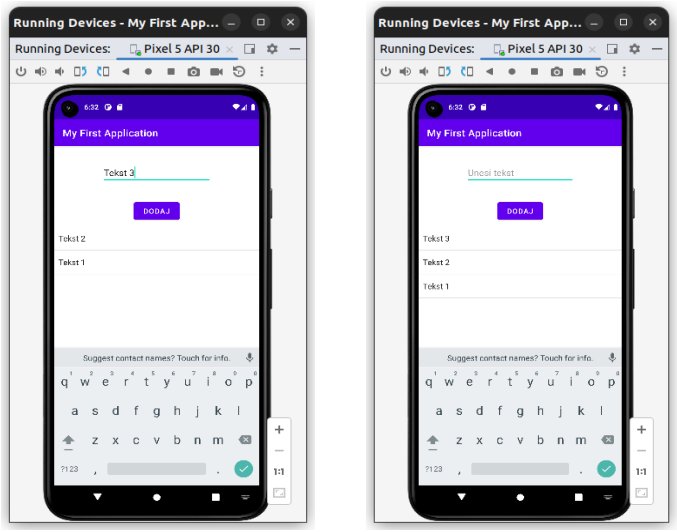
U funkciji `addToList` element se dodaje u listu, pošalje se obavijest da je došlo do promjene adapteru i obriše se uneseni tekst.

```
//Poziva se na klik dugmenta
private fun addToList() {
    // Novi tekst se dodaje kao prvi element
    listaVrijednosti.add(0,editText.text.toString())
    adapter.notifyDataSetChanged();
    editText.setText(“”);
}
```

Cjelokupni kôd klase je:

```
class MainActivity : AppCompatActivity() {
    private lateinit var listView: ListView
    private lateinit var editText: EditText
    private lateinit var button: Button
    // moze se koristiti i mutableListOf
    private val  listaVrijednosti = arrayListOf<String>()
    private lateinit var adapter : ArrayAdapter<String>
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //Inicijalizirat cemo elemente
        button = findViewById(R.id.button1);
        editText = findViewById(R.id.editText1)
        listView = findViewById(R.id.listView1)
        //Koristi se androidov layout
        adapter = ArrayAdapter(this, android.R.layout.simple_
        list_item_1, listaVrijednosti)
        listView.adapter=adapter
        //Definisat cemo akciju u slucaju klik akcije
        button.setOnClickListener {
            addToList()
        }
    }
    //Poziva se na klik dugmenta
     private fun addToList() {
        // Novi tekst se dodaje kao prvi element
        listaVrijednosti.add(0,editText.text.toString())
        adapter.notifyDataSetChanged();
        editText.setText(“”);
    }
}
```

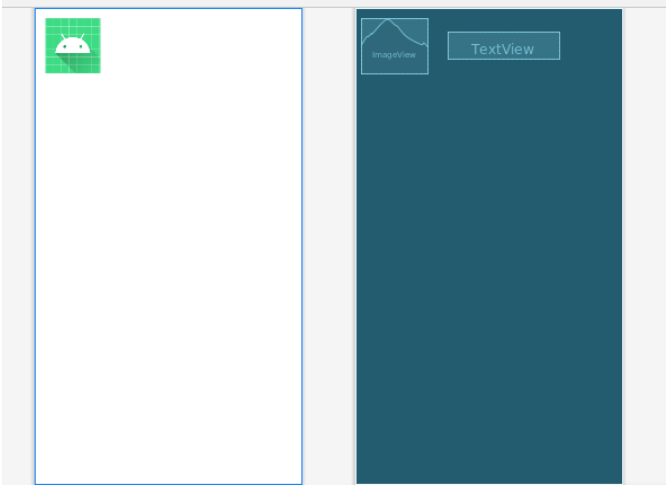
Izgled aplikacije nakon pokretanja i testiranja je:



Snimak zaslona 32. Ponašanje aplikacije

U nastavku definisati izgled elementa liste, takav da se pored svakog teksta nalazi slika. Kako se radi o posebnom *layout*-u, treba kreirati vlastiti `ArrayAdapter`.

U prvom koraku definisati izgled elementa liste. U odgovarajućem `res/layout` folder-u dodati novi *layout* kroz desni klik `New->XML->XML Layout File`. U *layout* dodati `ImageView` i postaviti neku *default* sliku, te dodati jedan `TextView` s ID `textElement`. Izgled *layout*-a `element_list.xml` je:



Snimak zaslona 33. Kreiranje korisničkog interfejsa

Kôd *layout*-a je:

```
<?xml version=”1.0” encoding=”utf-8”?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android=”http://schemas.android.com/apk/res/android”
    xmlns:app=”http://schemas.android.com/apk/res-auto”
    xmlns:tools=”http://schemas.android.com/tools”
    android:layout_width=”fill_parent”
    android:layout_height=”fill_parent”
    android:orientation=”horizontal” >
    <ImageView
        android:id=”@+id/icon”
        android:layout_width=”102dp”
        android:layout_height=”85dp”
        android:layout_marginStart=”8dp”
        android:layout_marginTop=”8dp”
        android:layout_marginBottom=”8dp”
        app:layout_constraintBottom_toBottomOf=”parent”
        app:layout_constraintEnd_toStartOf=”@+id/textElement”
        app:layout_constraintHorizontal_bias=”0.1”
        app:layout_constraintStart_toStartOf=”parent”
        app:layout_constraintTop_toTopOf=”parent”
        app:layout_constraintVertical_bias=”0.012”
        app:srcCompat=”@mipmap/ic_launcher” />
    <TextView
        android:id=”@+id/textElement”
        android:layout_width=”0dp”
        android:layout_height=”wrap_content”
        android:layout_marginStart=”32dp”
        android:layout_marginTop=”36dp”
        android:layout_marginEnd=”98dp”
        android:paddingTop=”10dp”
        android:textSize=”24sp”
        app:layout_constraintEnd_toEndOf=”parent”
        app:layout_constraintHorizontal_bias=”0.0”
        app:layout_constraintStart_toEndOf=”@+id/icon”
        app:layout_constraintTop_toTopOf=”parent” />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Definisati sada klasu vlastitog `ArrayAdapter-a` čiji je kôd:

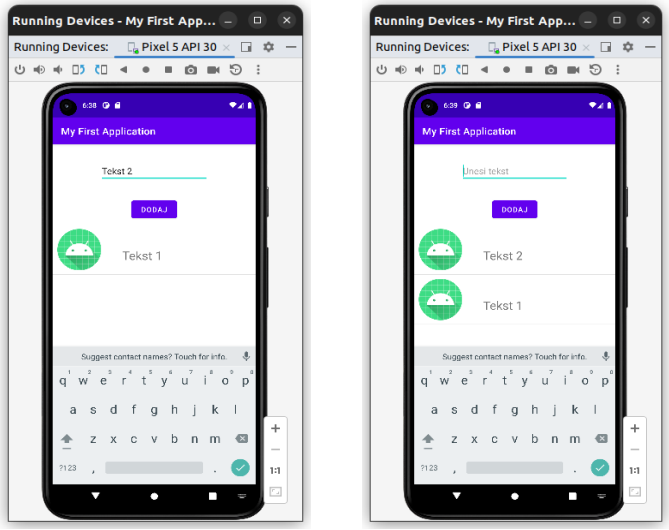
```
class MyArrayAdapter(context: Context, @LayoutRes private val
layoutResource: Int, private val elements: ArrayList<String>):
    ArrayAdapter<String>(context, layoutResource, elements) {
```

```
    override fun getView(position: Int, newView: View?, parent:
    ViewGroup): View {
        var newView = newView
        newView = LayoutInflater.from(context).inflate(R.layout.
        element_list, parent, false)
        val textView = newView.findViewById<TextView>(R.
        id.textElement)
        val element = elements.get(position)
        textView.text=element
        return newView
    }
}
```

Izmjene nad glavnom klasom se odnose na definisanje i inicijalizaciju adaptera, a one su:

```
private lateinit var adapter : MyArrayAdapter
adapter = MyArrayAdapter(this, R.layout.element_list,
listaVrijednosti)
```

Rezultat izvršenja i unosa par vrijednosti je dat u nastavku:



Snimak zaslona 34. Ponašanje aplikacije

Napomena: U praksi se sve više primjenjuje `RecyclerView` i odgovarajući adapter, pa će isti biti i ovdje primijenjen. S ovim je zvanično završen rad na pokaznoj aplikaciji, a u nastavku će biti kreirana *MoviApp* aplikacija.

5.5.1. Movie App

Slijedi opis osnovnih principa razvoja moderne Android aplikacije (`Cineaste`) koja prikazuje najnovije filmove, omiljene filmove, detalje o filmovima, omogućava pretragu filmova putem TMDB API, te njihovo dodavanje u favorite.

5.5.1.1. Zadatak 1.

Kreirati data klasu Movie, koja sadrži naziv filma (title - string), žanr (genre-string), datum izdavanja (releaseDate- string), službenu web stranicu (homepage - string) i kratki tekst sadržaja (overview - string). Napraviti datoteku MoviesStaticData u kojoj se nalaze funkcije koje vraćaju listu testnih podataka za omiljene filmove i najnovije filmove.

`Movie` klasa:

```
data class Movie(
    val id: Long,
    val title: String,
    val overview: String,
    val releaseDate: String,
    val homepage: String,
    val genre: String
)
```

`MovieStaticData` datoteka s funkcijama za dobavljanje omiljenih filmova i najnovijih filmova je:

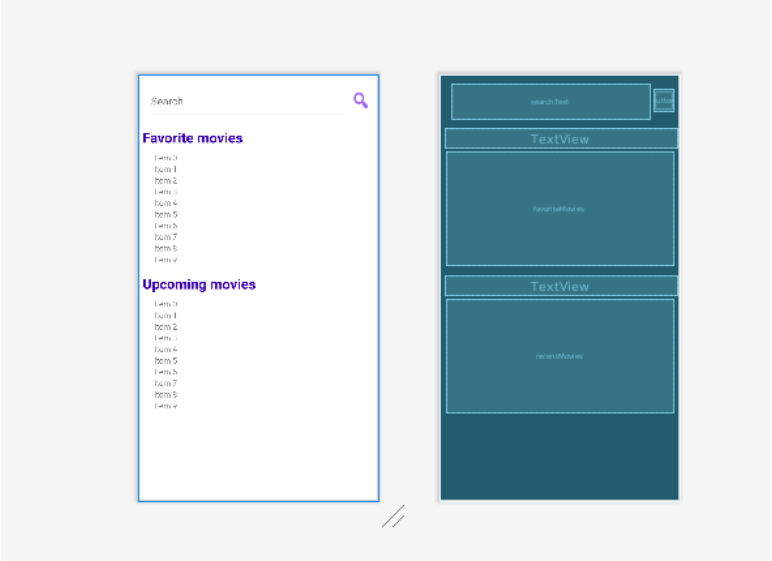
```
fun getFavoriteMovies(): List<Movie> {
    return listOf(
         Movie(1,”Pride and prejudice”,
            “Sparks fly when spirited Elizabeth Bennet
            meets single, rich, and proud Mr. Darcy. But
            Mr. Darcy reluctantly finds himself falling in love
            with a woman beneath his class. Can each overcome
            their own pride and
            prejudice?”,”16.02.2005.”,”https://www.imdb.com/
            title/tt0414387/”,
```

```
            “drama”),
        //Dodati filmove po zelji
    )
}
fun getRecentMovies(): List<Movie> {
    return listOf(
        Movie(1,”Creed III”,
            “Adonis has been thriving in both his career and
            family life, but when a childhood friend and former
            boxing prodigy resurfaces, the face-off is
            more than just a fight.”, “03.03.2023.”,
            “https://www.imdb.com/title/tt11145118”, “drama”),
        //Dodati filmove po zelji
    )
}
```

5.5.1.2. Zadatak 2.

Kreirati početni layout aplikacije u kojem će postojati `EditText` *i* `Button` *za pretragu filmova, te dvije horizontalne liste omiljenih filmova i najnovijih filmova.*

Kreirati početni *layout* aplikacije slijedećeg izgleda:



Snimak zaslona 35. Kreiranje korisničkog interfejsa

XML kôd *layout*-a je:

```
<?xml version=”1.0” encoding=”utf-8”?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android=”http://schemas.android.com/apk/res/android”
    xmlns:app=”http://schemas.android.com/apk/res-auto”
    xmlns:tools=”http://schemas.android.com/tools”
    android:layout_width=”match_parent”
    android:layout_height=”match_parent”
    tools:context=”.MainActivity”>
    <EditText
        android:id=”@+id/searchText”
        android:layout_width=”341dp”
        android:layout_height=”60dp”
        android:layout_marginStart=”16dp”
        android:layout_marginTop=”16dp”
        android:layout_marginEnd=”4dp”
        android:hint=”@string/search”
        app:layout_constraintEnd_toStartOf=”@+id/searchButton”
        app:layout_constraintStart_toStartOf=”parent”
        app:layout_constraintTop_toTopOf=”parent” />

    <Button
        android:id=”@+id/searchButton”
        android:layout_width=”34dp”
        android:layout_height=”39dp”
        android:layout_marginTop=”25dp”
        android:layout_marginEnd=”8dp”
        android:background=”@android:drawable/ic_menu_search”
        app:layout_constraintEnd_toEndOf=”parent”
        app:layout_constraintStart_toEndOf=”@+id/searchText”
        app:layout_constraintTop_toTopOf=”parent” />
    <TextView
        android:id=”@+id/textView”
        android:layout_width=”398dp”
        android:layout_height=”33dp”
        android:layout_marginStart=”8dp”
        android:layout_marginTop=”16dp”
        android:layout_marginEnd=”8dp”
        android:text=”@string/favorite”
        android:textColor=”@color/purple_700”
        android:textSize=”24sp”
        android:textStyle=”bold”
        app:layout_constraintEnd_toEndOf=”parent”
```

```
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/searchText" />
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/favoriteMovies"
        android:layout_width="388dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:clipToPadding="false"
        android:paddingStart="16dp"
        android:paddingEnd="16dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView" />
    <TextView
        android:id="@+id/textView1"
        android:layout_width="398dp"
        android:layout_height="33dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="8dp"
        android:text="@string/recent"
        android:textColor="@color/purple_700"
        android:textSize="24sp"
        android:textStyle="bold"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/
favoriteMovies" />
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recentMovies"
        android:layout_width="388dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:clipToPadding="false"
        android:paddingStart="16dp"
        android:paddingEnd="16dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
```

```
        app:layout_constraintTop_toBottomOf="@+id/textView1" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

5.5.1.3. Zadatak 3.

Kreirati layout za listu omiljenih filmova, te `RecyclerViewAdapter` *za istu.*

Bit će kreiran *layout* za element liste `item_movie` u obliku kartice. Sadržavat će sliku žanra filma koju je potrebno dodati u `res/drawable` folder. Bit će kreirano više ikonica za različite žanrove filma i naziv filma.

Kôd *layout*-a je:

```
<androidx.cardview.widget.CardView xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="128dp"
    android:layout_height="172dp"
    android:layout_marginEnd="8dp"
    app:cardCornerRadius="4dp">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <ImageView
            android:id="@+id/movieImage"
            android:layout_width="match_parent"
            android:layout_height="100dp"
            android:scaleType="centerCrop"
            app:srcCompat="@drawable/picture1" />
        <TextView
            android:id="@+id/movieTitle"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_gravity="center_horizontal"
            android:layout_margin="10dp"
            android:textSize="14sp" />
    </LinearLayout>
</androidx.cardview.widget.CardView>
```

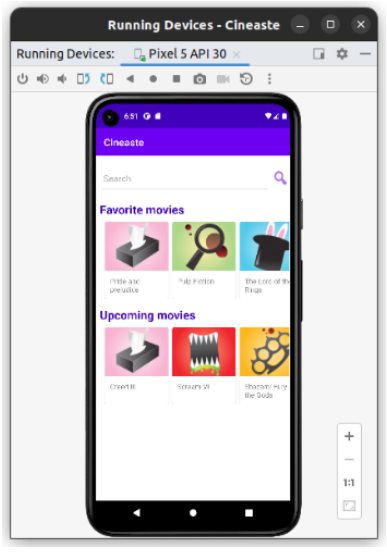
U nastavku slijedi kreiranje odgovarajućeg `RecyclerViewAdapte`-a u kojem se povezuju naziv filma i slika žanra.

```
class MovieListAdapter(
    private var movies: List<Movie>
) : RecyclerView.Adapter<MovieListAdapter.MovieViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup, viewType:
    Int): MovieViewHolder {
        val view = LayoutInflater
            .from(parent.context)
            .inflate(R.layout.item_movie, parent, false)
        return MovieViewHolder(view)
    }
    override fun getItemCount(): Int = movies.size
    override fun onBindViewHolder(holder: MovieViewHolder,
    position: Int) {
        holder.movieTitle.text = movies[position].title;
        val genreMatch: String = movies[position].genre
        //Pronalazimo id drawable elementa na osnovu naziva zanra
        val context: Context = holder.movieImage.context
        var id: Int = context.resources
            .getIdentifier(genreMatch, “drawable”, context.
            packageName)
        if (id==0) id=context.resources
            .getIdentifier(“picture1”, “drawable”, context.
            packageName)
        holder.movieImage.setImageResource(id)
    }
    fun updateMovies(movies: List<Movie>) {
        this.movies = movies
        notifyDataSetChanged()
    }
    inner class MovieViewHolder(itemView: View) : RecyclerView.
    ViewHolder(itemView) {
        val movieImage: ImageView = itemView.findViewById(R.
        id.movieImage)
        val movieTitle: TextView = itemView.findViewById(R.
        id.movieTitle)
    }
}
```

U `MainActivity` klasi se definiše i inicijalizira `RecyclerView` i `RecyclerViewAdapter`, te dodijeljuju filmovi. `RecyclerView`-u će biti dodijeljen `LinearLayoutManager` i to horizontalni.

```
class MainActivity : AppCompatActivity() {
    private lateinit var favoriteMovies: RecyclerView
    private lateinit var favoriteMoviesAdapter: MovieListAdapter
    private var favoriteMoviesList =  getFavoriteMovies()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        favoriteMovies = findViewById(R.id.favoriteMovies)
        favoriteMovies.layoutManager = LinearLayoutManager(
            this,
            LinearLayoutManager.HORIZONTAL,
            false
        )
        favoriteMoviesAdapter = MovieListAdapter(listOf())
        favoriteMovies.adapter = favoriteMoviesAdapter
        favoriteMoviesAdapter.updateMovies(favoriteMoviesList)
    }
}
```

Korištenjem navedenog kôda, moguće je napraviti analogni prikaz za najnovije filmove. Rezultat izvršenja implementiranog kôda je:



Snimak zaslona 36. Početni ekran aplikacije

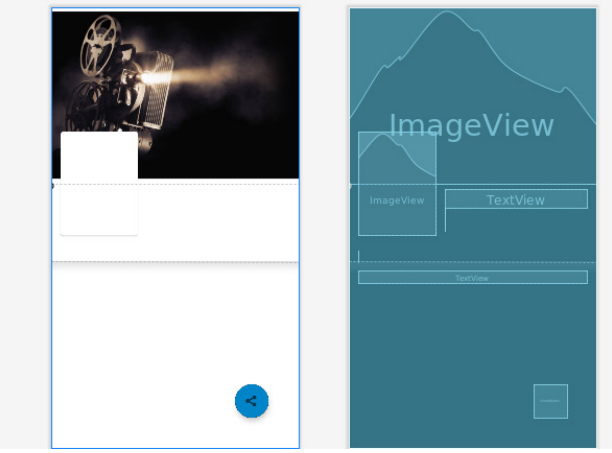
5.6. Intent i broadcast receiver

5.6.1. Zadatak 1.

Kreirati novu aktivnost u kojoj će biti prikazani detalji o filmu.

Desni klik na `app` i odabir stavki `New->Activity->EmptyActivity` otvara prozor za kreiranje aktivnosti. Kreirati novu aktivnost `MovieDetailActivity`, te potom definisati njen *layout*.

Primjer izgleda je:



Snimak zaslona 37. Kreiranje korisničkog interfejsa

XML datoteka nove aktivnosti je:

```
<androidx.core.widget.NestedScrollView 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”
    android:fillViewport=”true”>

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

        <ImageView
            android:id=”@+id/movie_backdrop”
```

```
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="@+id/
        backdrop_guideline"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/backdrop" />

    <androidx.cardview.widget.CardView
        android:id="@+id/movie_poster_card"
        android:layout_width="128dp"
        android:layout_height="172dp"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="8dp"
        app:cardCornerRadius="4dp"
        app:layout_constraintBottom_toBottomOf="@+id/
        backdrop_guideline"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/backdrop_
        guideline">

        <ImageView
            android:id="@+id/movie_poster"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

    </androidx.cardview.widget.CardView>

    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/backdrop_guideline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintGuide_percent="0.4" />

    <TextView
        android:id="@+id/movie_title"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="16dp"
        android:textColor="@android:color/white"
```

```
            android:textSize="24sp"
            android:textStyle="bold"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/movie_
            poster_card"
            app:layout_constraintTop_toBottomOf="@+id/backdrop_
            guideline" />

        <TextView
            android:id="@+id/movie_release_date"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#757575"
            android:textSize="14sp"
            app:layout_constraintStart_toStartOf="@+id/movie_
            title"
            app:layout_constraintTop_toBottomOf="@+id/movie_
            title" />

        <TextView
            android:id="@+id/movie_genre"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#757575"
            android:textSize="14sp"
            app:layout_constraintStart_toStartOf="@id/movie_
            release_date"
            app:layout_constraintTop_toBottomOf="@id/movie_
            release_date" />

        <TextView
            android:id="@+id/movie_website"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="24dp"
            android:textColor="@color/highlightColor"
            android:textSize="14sp"
            app:layout_constraintStart_toStartOf="@id/movie_
            poster_card"
            app:layout_constraintTop_toBottomOf="@id/movie_
            poster_card" />

        <androidx.constraintlayout.widget.Barrier
            android:id="@+id/movie_poster_title_barrier"
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:barrierDirection="bottom"
            app:constraint_referenced_ids="movie_release_
            date,movie_poster_card,movie_genre,movie_website"
            tools:layout_editor_absoluteY="379dp" />

        <TextView
            android:id="@+id/movie_overview"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="16dp"
            android:layout_marginTop="16dp"
            android:layout_marginEnd="16dp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/movie_
            poster_title_barrier"
            tools:textSize="16sp" />
        <com.google.android.material.floatingactionbutton.
        FloatingActionButton
            android:id="@+id/shareButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="339dp"
            android:layout_marginTop="132dp"
            android:layout_marginEnd="32dp"
            android:backgroundTint="@color/highlightColor"
            android:clickable="true"
            android:src="@android:drawable/ic_menu_share"
            android:tint="#FFFFFF"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintHorizontal_bias="0.0"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/movie_
            overview" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</androidx.core.widget.NestedScrollView>
```

5.6.2. Zadatak 2.

Dodati akciju koja će se deiti prilikom klika na listu filmova, takvu da se otvara aktivnost koja prikazuje detalje o filmu. Potrebno je proslijediti naziv

filma iz liste u sljedeću aktivnost, te pronaći isti film u listi i prikazati vrijednosti u odgovarajućim poljima.

Izvršiti prvo kreiranje intenta. U glavnoj aktivnosti definisati funkciju koja će se izvršiti na klik. Unutar funkcije se kreira intent i dodatni parametri.

```
private fun showMovieDetails(movie: Movie) {
    val intent = Intent(this, MovieDetailActivity::class.java).
    apply {
        putExtra("movie_title", movie.title)
    }
    startActivity(intent)
}
```

Nakon definisanja funkcije, odrediti gdje postaviti `onListItemClick-Listener` ili `onClickListener`. Preporuka je da se isti deifniše unutar `onBindViewHolder-a`.

U prvom koraku u konstruktor adaptera dodati funkciju višeg reda (*high-order function*) koja prima funkciju i vraća *void*.

```
class MovieListAdapter(
    private var movies: List<Movie>,
    private val onItemClicked: (movie:Movie) -> Unit
    ) : RecyclerView.Adapter<MovieListAdapter.MovieViewHolder>()
{
```

Sada je potrebno definisati da se funkcija koristi u `onBindViewHolder-u`. Na kraj metode dodati:

```
holder.itemView.setOnClickListener{
onItemClicked(movies[position]) }
```

Obzirom da je izvršena izmjena nad konstruktorom, sada je potrebno drugačije inicijalizirati adapter.

```
favoriteMoviesAdapter = MovieListAdapter(arrayListOf()) { movie
-> showMovieDetails(movie) }
recentMoviesAdapter = MovieListAdapter(arrayListOf()) { movie ->
showMovieDetails(movie) }
```

Omogućeno je pokretanje nove aktivnosti na klik. Definisati sada ponašanje nove aktivnosti. Kreirati metodu unutar `MovieDetailActivity` koja omogućava pretragu filma:

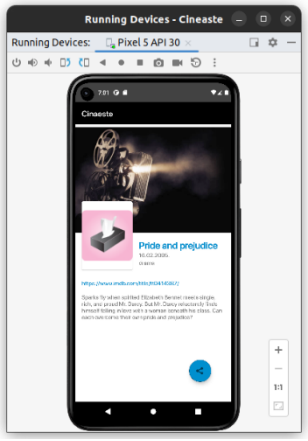
```
private fun getMovieByTitle(name:String):Movie{
    val movies: ArrayList<Movie> = arrayListOf()
    movies.addAll(getRecentMovies())
    movies.addAll(getFavoriteMovies())
    val movie= movies.find { movie -> name == movie.title }
    return movie?:Movie(0,”Test”,”Test”,”Test”,”Test”,”Test”)
}
}
```

Slijedi prikaz klase `MovieDetailActivity`:

```
class MovieDetailActivity : AppCompatActivity() {
    private lateinit var movie: Movie
    private lateinit var title : TextView
    private lateinit var overview : TextView
    private lateinit var releaseDate : TextView
    private lateinit var genre : TextView
    private lateinit var website : TextView
    private lateinit var poster : ImageView
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_movie_detail)
        title = findViewById(R.id.movie_title)
        overview = findViewById(R.id.movie_overview)
        releaseDate = findViewById(R.id.movie_release_date)
        genre = findViewById(R.id.movie_genre)
        poster = findViewById(R.id.movie_poster)
        website = findViewById(R.id.movie_website)
        val extras = intent.extras
        if (extras != null) {
            movie = getMovieByTitle(extras.getString(“movie_
            title”,””))
            populateDetails()
        } else {
            finish()
        }
    }
    private fun populateDetails() {
        title.text=movie.title
        releaseDate.text=movie.releaseDate
        genre.text=movie.genre
        website.text=movie.homepage
        overview.text=movie.overview
        val context: Context = poster.context
```

```
        var id: Int = context.resources
            .getIdentifier(movie.genre, “drawable”, context.
            packageName)
        if (id===0) id=context.resources
            .getIdentifier(“picture1”, “drawable”, context.
            packageName)
        poster.setImageResource(id)
    }
   private fun getMovieByTitle(name:String):Movie{
        val movies: ArrayList<Movie> = arrayListOf()
        movies.addAll(getRecentMovies())
        movies.addAll(getFavoriteMovies())
        val movie= movies.find { movie -> name == movie.title }
        return movie?:Movie(0,”Test”,”Test”,”Test”,”Test”,”Test”)
    }
}
```

Nakon pokretanja implementiranog kôda izgled detalja filma je:



Snimak zaslona 38. Pokrenuta aplikacija

5.6.3. Zadatak 3.

Napraviti da se pokrene web preglednik kada se klikne na link web stranice filma i da se učita navedena stranica.

Postaviti prvo `setOnClickListener`:

```
website.setOnClickListener{
        showWebsite()
}
```

Definisati funkciju `showWebsite()`:

```
private fun showWebsite(){
    val webIntent: Intent = Intent(Intent.ACTION_VIEW,
    Uri.parse(movie.homepage))
    try {
        startActivity(webIntent)
    } catch (e: ActivityNotFoundException) {
        // Definisati naredbe ako ne postoji aplikacija za
        navedenu akciju
    }
}
```

5.6.4. Zadatak 4.

Napraviti da aplikacija odgovara na akciju tipa `ACTION_SEND`, ako se putem te akcije prosljeđuje tekst. Navedeni tekst treba da popuni editText u početnoj aktivnosti koji se nalazi iznad button-a pretrage.

Prvo u `manifest` datoteci unutar tag-a glavne aktivnosti definisati `intent-filter` na koji odgovara aplikacija.

```
<intent-filter>
    <action android:name=”android.intent.action.SEND” />
    <category android:name=”android.intent.category.DEFAULT” />
    <data android:mimeType=”text/plain” />
</intent-filter>
```

Unutar `onCreate` glavne aktivnosti provjeriti da li postoji intent s odgovarajućim tipom i proslijediti ga funkciji ako postoji:

```
if(intent?.action == Intent.ACTION_SEND && intent?.type ==
“text/plain”)
    handleSendText(intent)
```

Funkcija koja će rukovati intent je:

```
private fun handleSendText(intent: Intent) {
    intent.getStringExtra(Intent.EXTRA_TEXT)?.let {
        searchText.setText(it)
    }
}
```

5.7. Testiranje mobilne aplikacije

5.7.1. Zadatak 1.

Napraviti instrumented testove za aktivnost `MovieDetailsActivity`*. U testovima provjeriti da li se ispravno popunjava korisnički interfejs aktivnosti s podacima odgovarajućeg filma čiji naziv je proslijeđen u extra podacima intenta. Nakon što se testira instanciranje aktivnosti, testirati da li klik na link poziva intent sa akcijom* `Intent.ACTION_VIEW`*.*

Prvo je potrebno dodati *dependency* u `app/gradle.build` i dodati klasu za testove u folder `app/src/androidTest/` s nazvom `IntentInstrumentedTest`.

Testiranje instanciranja klase i popunjavanja korisničkog interfejsa

Nakon što je klasa pripremljena dodati metodu prvog testa:

```
@Test
fun testDetailActivityInstantiation(){
    val pokreniDetalje: Intent =
Intent(MovieDetailActivity::javaClass.name)
    pokreniDetalje.putExtra("movie_title","Pulp Fiction")
    val scenario =
launchActivity<MovieDetailActivity>(pokreniDetalje)
    onView(withId(R.id.movie_title)).
check(matches(withText("Pulp Fiction")))
    onView(withId(R.id.movie_genre)).
check(matches(withText("crime")))
    onView(withId(R.id.movie_overview)).
check(matches(withSubstring("pair of diner bandits")))
}
```

U prve dvije linije se priprema intent kojim će otvoriti aktivnost detalja filma sa nazivom *'Pulp fiction'*. Intent se pokreće koristeći `launchActivity()`. U testu se provjerava da li `TextView` elementi naziva, žanra i opisa filma imaju odgovarajuće vrijednosti.

U ovaj test treba dodati i provjeru da li je slika žanra filma ispravno popunjena. Da bi se ovo uradilo treba implementirati vlastiti *view matcher*. Za implementaciju koristiti apstraktnu klasu `TypeSafeMatcher<View>`. Ova klasa zahtijeva da budu implementirane metode `describeTo` i `matchesSafely`. Metoda `describeTo(decrtiption:Description)` dodaje poruku u rezultat testa ukoliko provjera nije rezultirala s *true* vrijednosti, a u metodi

`matchesSafely(item:View)` se vrši odgovarajuća provjera. Ovaj *matcher* vrši provjeru da li je item koji se testira ispravnog tipa i da nije null.

Provjera da li slika žanra odgovara slici žanra iz resursa android aplikacije:

```
fun withImage(@DrawableRes id: Int) = object :
TypeSafeMatcher<View>(){
    override fun describeTo(description: Description) {
        description.appendText("Drawable does not contain image
        with id: $id")
    }

    override fun matchesSafely(item: View): Boolean {
        val context:Context = item.context
        val bitmap: Bitmap? = context.getDrawable(id)?.toBitmap()
        return item is ImageView && item.drawable.toBitmap().
        sameAs(bitmap)
    }

}
```

Ovaj *matcher* se može dodati unutar klase testa ili ga izdvojiti u posebnu *utility* klasu. Provjera koja se vrši u *matcher*-u je da je item koji se testira tipa `ImageView` i da bitmap zapis slike koji se nalazi u tom itemu odgovara bitmap zapisu slike *drawable* resursa sa id-em koji je proslijeđen kao parametar custom `withImage(id:Int)` *matcher*-a.

U test `testDetailActivityInstantiation()` se sada može dodati i provjera da li je slika žanra filma ispravno popunjena:

```
onView(withId(R.id.movie_poster)).check(matches(withImage(R.
drawable.crime)))
```

Ovaj test bi sada trebao ispravno proći. Ukoliko se izmijeni *drawable* i stavi *npm* `R.drawable.family` test će pasti i vidjet će se poruka koja se generiše u metodi `describeTo` custom *matcher*-a.

Testiranje da li klik na link aktivira intent sa akcijom `Intent.ACTION_VIEW`

Dodati novi test u prethodnu klasu `IntentInstrumentedTest` s nazvom `testLinksIntent()`:

```
@Test
fun testLinksIntent(){
```

```
    Intents.init()
    val pokreniDetalje: Intent =
Intent(MovieDetailActivity::javaClass.name)
    pokreniDetalje.putExtra("movie_title","Pulp Fiction")
    val scenario =
launchActivity<MovieDetailActivity>(pokreniDetalje)
    onView(withId(R.id.movie_website)).perform(click())
    intended(hasAction(Intent.ACTION_VIEW))
    Intents.release()
}
```

Ovaj test pokreće aktivnost detalja filma kao i prethodni. Nakon pokretanja aktivnosti simulira se klik na link (element sa id-em `R.id.moview_website`). S metodom `intended()` se provjerava da li pozvani intent ima akciju `Intent.ACTION_VIEW`. Ovaj test bi trebao ispravno proći, ali mu može biti potrebno više vremena da prođe nego prethodni, jer test čeka da vidi rezultat intenta. Ako se promijenti akcija u testu i stavi npr. `Intent.ACTION_SEND` test će pasti nakon što istekne time-out, jer se nije pojavio intent sa ovakvom akcijom. U prvoj liniji testa je pripremljena `Intents` klasa koja će zapisivati pokrenute intente, na kraju testa treba osloboditi klasu kako bi se mogla korisitit u drugim testovima.

5.7.2. Zadatak 2.

Napraviti unit test za testiranje statičkih podataka.

Za razliku od instrumented testova, unit testovi se pokreću lokalno na računaru gdje se razvija aplikaciju. Testna klasa lokalnih unit testova se treba kreirati u folderu `app/src/test/` i imat će nazv `UnitTests`.

Da bi koristili *matcher*-e koji olakšavaju provjere stanja komponenti, u `app/build.gradle` treba dodati *dependency*:

```
testImplementation("org.hamcrest:hamcrest:2.2")
```

U testnu klasu dodati metodu `testGetFavoriteMovies`:

```
@Test
fun testGetFavoriteMovies(){
    val movies = getFavoriteMovies()
    assertEquals(movies.size,6)
    assertThat(movies, hasItem<Movie>(hasProperty("title",
    Is("Pulp Fiction"))))
    assertThat(movies, not(hasItem<Movie>(hasProperty("title",
```

```
    Is("Black Widow")))))
}
```

U ovom testu se provjerava da li metoda `getFavoriteMovies()` vraća odgovarajući niz. Prva provjera je veličina rezultujućeg niza koja treba biti 6. Druga provjera je da niz filmova sadrži element tipa `Movie` koji ima atribut `title` sa vrijednosti *"Pulp Fiction"*, a treća provjera gleda da niz ne sadrži element tipa `Movie` sa atributom `title` i vrijednosti *"Black Widow"*.

Treba voditi računa da je `assertThat` metoda importovana kao `import org.hamcrest.MatcherAssert.assertThat` (postoji ova metoda u drugim paketima koji su deprecated).

5.8. Fragmenti i navigacijska komponenta

5.8.1. Zadatak 1

Izdvojiti pretragu, nove filmove i favorite u posebne fragmente koji se mijenjaju klikom na donju navigacijsku traku (Bottom navigation bar)

U Androidu jedan od najlakših načina navigacije je korištenjem `Bottom navigation bar`-a. Ona se koristi kada aplikacija ima 3-5 lokacija, kojima se može direktno pristupati. Jedan od primjera navigacije je *Google Discover Android App.*

`BottomNavigationView` je u ovisnosti od verzije dostupan odmah ili se mora u `build gradle` dodati:

```
dependencies {
    implementation 'com.google.android.material:material:+'
}
```

`MainActivity.kt` klasa će sada biti prazna:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Sada je potrebno kreirati fragment klase i *layout*-e za sve tri stavke.

– Omiljeni filmovi

```
fragment_favorite_movies.xml
<?xml version=”1.0” encoding=”utf-8”?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android=”http://schemas.android.com/apk/res/android”
    xmlns:app=”http://schemas.android.com/apk/res-auto”
    xmlns:tools=”http://schemas.android.com/tools”
    android:layout_width=”match_parent”
    android:layout_height=”match_parent”
    >
    <TextView
        android:id=”@+id/textView”
        android:layout_width=”398dp”
        android:layout_height=”33dp”
        android:layout_marginStart=”16dp”
        android:layout_marginTop=”16dp”
        android:layout_marginEnd=”8dp”
        android:text=”@string/favorite”
        android:textSize=”20sp”
        app:layout_constraintEnd_toEndOf=”parent”
        app:layout_constraintHorizontal_bias=”0.0”
        app:layout_constraintStart_toStartOf=”parent”
        app:layout_constraintTop_toTopOf=”parent” />
    <androidx.recyclerview.widget.RecyclerView
        android:id=”@+id/favoriteMovies”
        android:layout_width=”388dp”
        android:layout_height=”wrap_content”
        android:layout_marginStart=”8dp”
        android:layout_marginTop=”16dp”
        android:layout_marginEnd=”8dp”
        android:clipToPadding=”false”
        android:paddingStart=”20dp”
        android:paddingTop=”10dp”
        android:paddingEnd=”20dp”
        android:paddingBottom=”10dp”
        app:layout_constraintEnd_toEndOf=”parent”
        app:layout_constraintStart_toStartOf=”parent”
        app:layout_constraintTop_toBottomOf=”@id/textView2” />
    <TextView
        android:id=”@+id/textView2”
        android:layout_width=”wrap_content”
        android:layout_height=”wrap_content”
        android:layout_marginStart=”16dp”
```

```
        android:layout_marginTop=”8dp”
        android:layout_marginBottom=”16dp”
        android:text=”@string/favoritesText”
        app:layout_constraintBottom_toTopOf=”@+id/favoriteMovies”
        app:layout_constraintStart_toStartOf=”parent”
        app:layout_constraintTop_toBottomOf=”@+id/textView” />
</androidx.constraintlayout.widget.ConstraintLayout>
FavoriteMoviesFragment.kt:
class FavoriteMoviesFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater,
    container: ViewGroup?, savedInstanceState: Bundle?): View? =
        inflater.inflate(R.layout.favorites_fragment, container,
        false)
    companion object {
        fun newInstance(): FavoriteMoviesFragment =
        FavoriteMoviesFragment()
    }
}
```

– Novi filmovi:

```
fragment_recent_movies.xml:
<?xml version=”1.0” encoding=”utf-8”?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android=”http://schemas.android.com/apk/res/android”
    xmlns:app=”http://schemas.android.com/apk/res-auto”
    xmlns:tools=”http://schemas.android.com/tools”
    android:layout_width=”match_parent”
    android:layout_height=”match_parent”
    >
    <TextView
        android:id=”@+id/textView1”
        android:layout_width=”398dp”
        android:layout_height=”33dp”
        android:layout_marginStart=”16dp”
        android:layout_marginTop=”16dp”
        android:layout_marginEnd=”8dp”
        android:text=”@string/recent”
        android:textSize=”20sp”
        app:layout_constraintEnd_toEndOf=”parent”
        app:layout_constraintHorizontal_bias=”0.0”
        app:layout_constraintStart_toStartOf=”parent”
        app:layout_constraintTop_toTopOf=”parent” />
    <androidx.recyclerview.widget.RecyclerView
```

```
        android:id="@+id/recentMovies"
        android:layout_width="388dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="8dp"
        android:clipToPadding="false"
        android:paddingStart="20dp"
        android:paddingTop="10dp"
        android:paddingEnd="20dp"
        android:paddingBottom="10dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/textView3" />
    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        android:layout_marginBottom="8dp"
        android:text="@string/recentTexts"
        app:layout_constraintBottom_toTopOf="@+id/recentMovies"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView1" />
</androidx.constraintlayout.widget.ConstraintLayout>
RecentMoviesFragment.kt:
class RecentMoviesFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater,
    container: ViewGroup?, savedInstanceState: Bundle?): View? =
        inflater.inflate(R.layout.recents_fragment, container,
        false)
    companion object {
        fun newInstance(): RecentMoviesFragment =
        RecentMoviesFragment()
    }
}
```

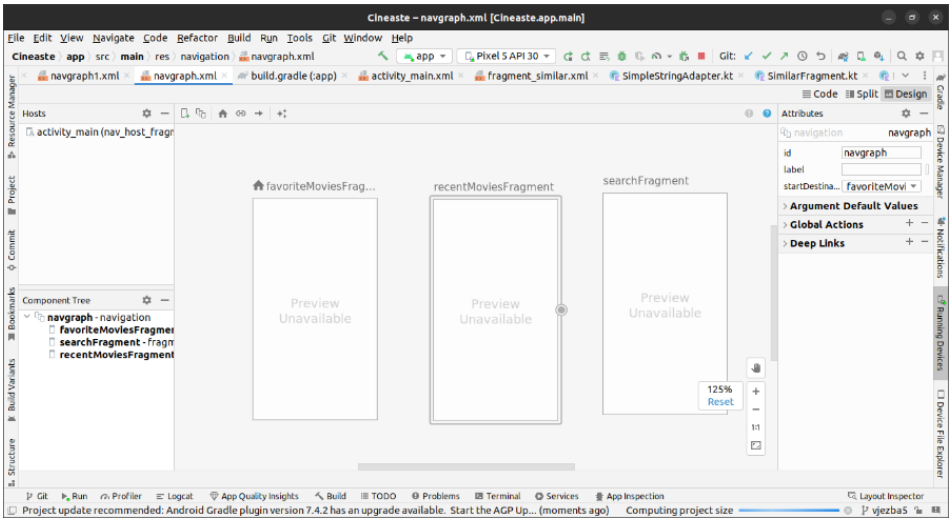
– Pretraga fragmenta

```
fragment_search.xml:
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <EditText
        android:id="@+id/searchText"
        android:layout_width="341dp"
        android:layout_height="60dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="4dp"
        android:hint="@string/search"
        app:layout_constraintEnd_toStartOf="@+id/searchButton"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <ImageButton
        android:id="@+id/searchButton"
        android:layout_width="34dp"
        android:layout_height="34dp"
        android:layout_marginTop="36dp"
        android:layout_marginEnd="8dp"
        android:background="?android:attr/
        actionModeWebSearchDrawable"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/searchText"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
SearchFragment.kt:
class SearchFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater,
    container: ViewGroup?, savedInstanceState: Bundle?): View? =
        inflater.inflate(R.layout.search_fragment, container, false)
    companion object {
        fun newInstance(): SearchFragment = SearchFragment()
    }
}
```

Kreirati sada navigacijsku komponentu s `New->Android Resource File`. Odabrati tip `Navigation` i nazvati datoteku `navgraph.xml`.

Dodati tri fragmenta:



Snimak zaslona 39. Kreiranje navigacijskog grafa

Kreirati navigacijsku komponentu s `New->Android Resource File`. Odabrati tip `Menu` i nazvati datoteku `menu.xml`.

```
<?xml version=”1.0” encoding=”utf-8”?>
<menu xmlns:android=”http://schemas.android.com/apk/res/android”>
    <item
        android:id=”@id/favoriteMoviesFragment”
        android:icon=”@android:drawable/btn_star_big_off”
        android:title=”Fafvorites” />

    <item
        android:id=”@id/recentMoviesFragment”
        android:icon=”@android:drawable/ic_popup_reminder”
        android:title=”Recent” />

    <item
        android:id=”@id/searchFragment”
        android:icon=”@android:drawable/ic_menu_search”
        android:title=”Search” />
</menu>
```

Layout glavne aktivnosti sada je:

```
<?xml version=”1.0” encoding=”utf-8”?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android=”http://schemas.android.com/apk/res/android”
```

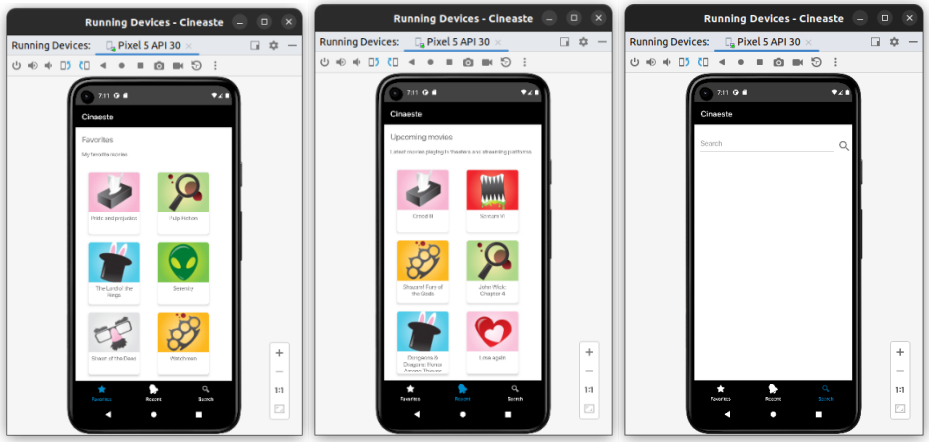
```
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="@id/bottomNavigation"
        app:defaultNavHost="true"
        android:layout_marginBottom="60dp"
        app:navGraph="@navigation/navgraph" />

    <com.google.android.material.bottomnavigation.
    BottomNavigationView
        android:layout_height="60dp"
        android:layout_width="match_parent"
        android:id="@+id/bottomNavigation"
        app:layout_constraintTop_toBottomOf="@id/nav_host_fragment"
        app:layout_constraintBottom_toBottomOf="parent"
        app:menu="@menu/menu" />

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

Navigacija se radi preko `NavController`-a koji će biti inicijaliziran u glavnoj aktivnosti:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val navHostFragment = supportFragmentManager.
        findFragmentById(R.id.nav_host_fragment) as
        NavHostFragment
        val navController = navHostFragment.navController
        val navView: BottomNavigationView = findViewById(R.
        id.bottomNavigation)
```

```
        navView.setupWithNavController(navController)
    }
}
```

Kada se aplikacija pokrene, vidi se uspješno implementirana navigacija:



Snimak zaslona 40. Prikaz fragmenata aplikacije

Sada je potrebno izmjeniti odgovarajuće klase za fragmente, da rade predviđeno.

Napomena: Za `context` se sada koristi `activity`. `LayoutManager` `RecyclerView`-eva neka sada bude `GridLayoutManager`.

`RecentMoviesFragment.kt`:

```
class RecentMoviesFragment : Fragment() {
    private lateinit var recentMovies: RecyclerView
    private lateinit var recentMoviesAdapter: MovieListAdapter
    private var recentMoviesList =  getRecentMovies()
    override fun onCreateView(inflater: LayoutInflater,
container: ViewGroup?, savedInstanceState: Bundle?): View? {
        var view =  inflater.inflate(R.layout.fragment_recent_
movies, container, false)
        recentMovies = view.findViewById(R.id.recentMovies)
        recentMovies.layoutManager = GridLayoutManager(activity, 2)
        recentMoviesAdapter = MovieListAdapter(arrayListOf()) {
movie -> showMovieDetails(movie) }
        recentMovies.adapter=recentMoviesAdapter
        recentMoviesAdapter.updateMovies(recentMoviesList)
```

```
        return view;
    }
    private fun showMovieDetails(movie: Movie) {
        val intent = Intent(activity,
        MovieDetailActivity::class.java).apply {
            putExtra(“movie_title”, movie.title)
        }
        startActivity(intent)
    }
}
```

Analogno uraditi i za `FavoriteMoviesFragment`.

Na ovaj način će biti prikazane odgovarajuće liste filmova. Preostaje još da se implementira odgovor na akciju za pretragu. Fragment za pretragu izmijeniti na sljedeći način:

```
class SearchFragment : Fragment() {
    private lateinit var searchText: EditText
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        var view = inflater.inflate(R.layout.fragment_search,
        container, false)
        searchText = view.findViewById(R.id.searchText)
        arguments?.getString(“search”)?.let {
            searchText.setText(it)
        }
        return view;
    }
}
```

U inicijalizaciji fragmenta se sada traži string. String se postavlja u `Bundle`-u. Kada se inicijalizira fragment u glavnoj aktivnosti, potrebno je proslijedit odgovarajući string pretrage ili prazan string.

U `onCreate` dodati slijedeće:

```
 if(intent?.action == Intent.ACTION_SEND && intent?.type ==
“text/plain”)
        {
            intent.getStringExtra(Intent.EXTRA_TEXT)?.let {
```

```
                val bundle = bundleOf("search" to it)
                navView.selectedItemId= R.id.searchFragment
                navController.navigate(R.
                id.searchFragment,bundle)
            }
        }
```

Sada aplikacija radi jednako kao i ranije, ali sa fragmentima.

5.9. Web servisi i coroutines

5.9.1. Zadatak 1

Registrovati se na stranici The Movie Database i upoznati se sa pretragom po nazivu.

Prvi korak implementacije zadataka je registracija na stranici: The Movie Database.[34]

Nakon registracije, potrebno je zatražiti izdavanje API KEY-a koji je potreban prilikom poziva endpoint-a servisa. Detaljnije o njegovom izdavanju se nalazi na: Introduction.[35]

Za potrebe zadatka je potrebno pogledati dokumentaciju pretrage filmova po nazivu, koja je data na stranici Search Movies[36] za servis Movie.[37]

Parametri koji se mogu proslijediti prilikom poziva ovog web servisa su dati u tabeli.

Tabela 1. Opis poziva web servisa

Parametar	Opis
api_key	Obavezni parametar - omogućava korištenje servisa.
query	Obavezni parametar - predstavlja traženi upit
language	Neobavezni parametar (en-US) - izbor jezika
page	Neobavezni parametar (1) - redni broj stranice prikaza. Svaki query može vratiti veliki broj rezultata, pri čemu servis vraća samo određeni broj, te omogućava paginaciju.
include_adult	Neobavezni parametar (false) - da li će vratiti filmove za odrasle

[34] https://www.themoviedb.org/account/signup

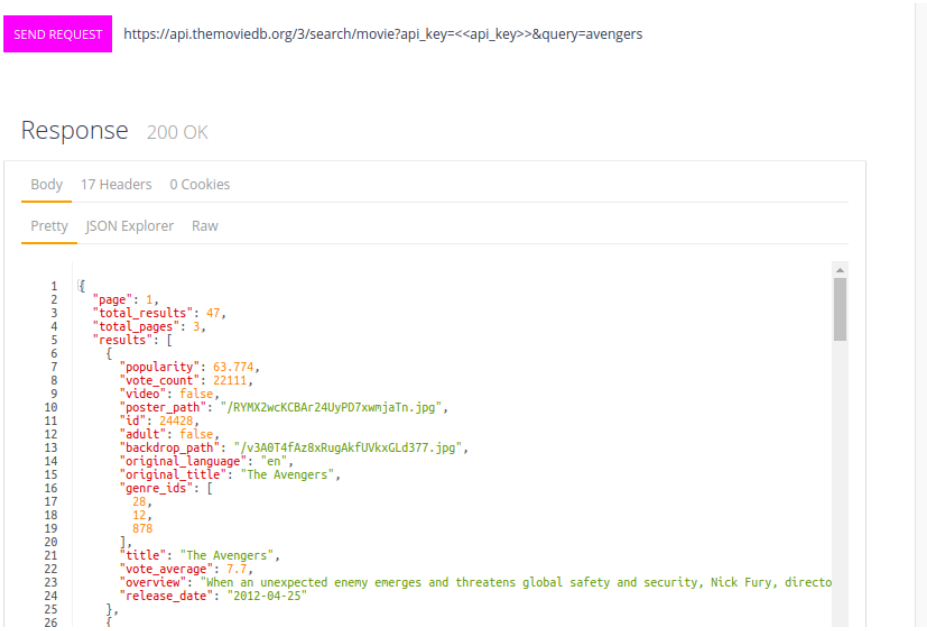
[35] https://developers.themoviedb.org/3/getting-started/introduction

[36] https://developers.themoviedb.org/3/search/search-movies

[37] https://api.themoviedb.org/3/search/movie

Parametar	Opis
region	Neobavezni parametar - speicifikacija ISO kôda za datum
year	Neobavezni parametar - godina
primary_release_year	Neobavezni parametar - godina prvog prikazivanja

Ovaj web servis se može isprobati u dokumentacije ili na način da se u web browser upiše URL web servisa zajedno sa parametrima po kojim se želi izvršiti pretraga. Na sljedećoj slici je dat primjer pretrage filmova s nazivom: *Avengers*.



Snimak zaslona 41. Rezultat poziva web servisa

U URL-u prosljeđujemo `api_key` i `query`.

Kao rezultat se dobije JSON objekat koji kaže da smo na prvoj stranici pretrage, da je ukupan broj rezultata 47 i da su ukupno 3 stranice. Filmovi se nalaze u nizu objekata `results` vraćenog objekta. Ono što se može primijetiti je da određenih stavki o filmu nema u implementiranoj `Movie` klasi, npr. `poster_path`, `backdrop_path` itd. Moguće je primijetiti da se žanr vraća u vidu niza integera, i web stranica ne postoji.

5.9.2. Zadatak 2

Potrebno je izvršiti izmjenu postojeće implementacije i prilagoditi je web servisu.

U prvom koraku će biti dodane odgovarajuće `INTERNET` permisije u `manifest` datoteci. Klasa `Movie` će biti proširena s `posterPath` i postavljeno da su atributi žanr i homepage `nullable`. Proširivanjem klase `Movie` treba izmijeniti listu statičkih podataka. U *layout* pretrage će biti dodan `RecyclerView` ispod `EditText`, u kojem će biti prikazani rezultati pretrage.

5.9.3. Zadatak 3

Napraviti funkcionalnost pretrage filmova po nazivu putem web servisa. Korisnik treba imati mogućnost upisivanja naziva filma i pokretanja pretrage putem button-a. Ukoliko je pretraga uspješna, lista filmova se treba ispuniti sa najviše šest pronađenih filmova.

Kako odgovor od web servisa može trajati proizvoljno dugo (loša konekcija, pad performansi servera na kojem se nalazi web servis i sl) izvršavanje navedene funkcionalnosti u niti aktivnosti nije poželjno. Takva implementacija bi zaustavljala rad aplikacije sve dok web servis ne vrati odgovor. Zbog ovoga ima smisla kreirati novu nit i unutar nje implementirati pozivanje web servisa i obradu njegovog odgovora. Za izvršavanje van glavne niti koristit će se `Kotlin Coroutines`.

Da bi se koristio `Coroutines` prvo se treba dodati *dependency* u `Gradle.module`:

```
implementation “org.jetbrains.kotlinx:kotlinx-coroutines-
android:+”
```

U prvom koraku će se pripremiti sve potrebno u `SearchFragment`-u. Referencirati odgovarajuće `view`-ove, postaviti adapter i `onClickListener` na `AppCompatImageButton`. Kreirati i `MovieRepository` klasu u kojoj će se nalaziti metode za dobavljanje podataka.

Poziv ka web servis-u će se izvršavati u pozadinskoj niti, pri čemu će biti `main-safe`, odnosno neće blokirati izmjene nad korisničkim interfejsom. Kako bi se postiglo da funkcija koja vrši poziv ka web servis-u bude `main-safe` koristit će se `withContext()` funkcija iz *coroutine* biblioteke koja vrši

prebacivanje na drugu nit. Poziv ka web servisu će se, za sada, izvršavati iz `MovieRepository` klase.

Kreirati prvo `Result` klasu koja će modelirati povrat mrežnog poziva.

```
sealed class Result<out R> {
    data class Success<out T>(val data: T) : Result<T>()
    data class Error(val exception: Exception) : Result<Nothing>()
}
```

U `MovieRepository` dodati funkciju `searchRequest` koja prima parametar pretrage.

```
object MovieRepository {

    private const val tmdb_api_key : String =
    "47a9d3194b76b969aee48f74f34e57dd"

    suspend fun searchRequest(
        query: String
    ): Result<List<Movie>>{
        return withContext(Dispatchers.IO) {
            try {
                val movies = arrayListOf<Movie>()
                val url1 =
                    "https://api.themoviedb.org/3/search/
                    movie?api_key=$tmdb_api_key&query=$query"
                val url = URL(url1)
                (url.openConnection() as? HttpURLConnection)?.run {
                    val result = this.inputStream.
                    bufferedReader().use { it.readText() }
                    val jo = JSONObject(result)
                    val results = jo.getJSONArray("results")
                    for (i in 0 until results.length()) {
                        val movie = results.getJSONObject(i)
                        val title = movie.getString("title")
                        val id = movie.getInt("id")
                        val posterPath = movie.
                        getString("poster_path")
                        val overview = movie.getString("overview")
                        val releaseDate = movie.
                        getString("release_date")
                        movies.add(Movie(id.toLong(), title,
                        overview, releaseDate, null, null,
```

```
                        posterPath, " "))
                        if (i == 5) break
                    }
                }
                return@withContext Result.Success(movies);
            }
            catch (e: MalformedURLException) {
                return@withContext Result.
                Error(Exception("Cannot open
                HttpURLConnection"))
            } catch (e: IOException) {
                return@withContext Result.
                Error(Exception("Cannot read stream"))
            } catch (e: JSONException) {
                return@withContext Result.
                Error(Exception("Cannot parse JSON"))
            }

        }
    }
}
```

`Dispatchers.IO` koji se proslijeđuje metodi `withContext` ukazuje da će se *coroutine*-a izvršiti na niti rezervisanoj za I/O operacije. Iskorištena je i `suspend` ključna riječ kako bi se forsiralo da se ova funkcija zove samo iz `coroutine`.

Objašnjenje metode:

1. Varijabla `tmdb_api_key` je trenutno privatni atribut metode
2. Formira se ispravni URL
3. Otvara se konekcija i vrši se poziv web servisa
4. Rezultat poziva web servisa je u obliku `InputStream-a`. Ovaj objekat će biti pretvoren u `String`.
5. Navedeni string sadrži rezultat poziva web servisa. Ovaj rezultat je u `JSON` formatu. Da bi se radilo s podacima u JSON formatu koristit će se `JSONObject` klasa. Bit će kreiran novi `JSONObject` koji će biti incijaliziran sa stringom rezultata. Ovaj objekat će sadržavati čitav JSON rezultata.
6. Iz navedenog objekta se mogu izdvojiti djelovi rezultata koji su interesantni. Da bi se dobio niz objekata koji je interesantan, izdvojit će se `JSONArray` s nazivnom `results` iz JSON objekta rezultata.

7. Dalje treba proći kroz listu svih rezultata, preuzeti podatke koji su interesantni (naziv filma, id, posterPath, overview, releaseDate).
8. Vratiti podatke.

Odgovarajuća funkcija će se pozivati iz metode `search` unutar fragmenta. Ova metoda će isto biti *coroutine*, ali će se izvršavati na glavnoj niti, te će samim tim moći izvršiti izmjene nad UI, odnosno proslijediti odgovarajuće podatke fragmentu nakon što ih dobije. Prvo će biti definisan `CoroutineScope` kao privatna varijabla.

```
val scope = CoroutineScope(Job() + Dispatchers.Main)
```

`CoroutineScope` vodi računa o svim pokrenutim `Coroutine`-ama.

Kreirati odgovarajuću metodu:

```
//On Click handler
private fun onClick() {
    val toast = Toast.makeText(context, “Search start”, Toast.
    LENGTH_SHORT)
    toast.show()
    search(searchText.text.toString())
}
fun search(query: String){
    val scope = CoroutineScope(Job() + Dispatchers.Main)
    // Kreira se Coroutine na UI
    scope.launch{
        // Vrti se poziv servisa i suspendira se rutina dok se
        `withContext` ne zavrsi
        val result = MovieRepository.searchRequest(query)
        // Prikaze se rezultat korisniku na glavnoj niti
        when (result) {
            is Result.Success<List<Movie>> -> searchDone(result.
            data)
            else-> onError()
        }
    }
}
```

Unutar ove metode se pozivaju metode `searchDone` i `onError` koje služe za ažuriranje UI-a, a njihova implementacija je prikazana u nastavku.

```
fun searchDone(movies:List<Movie>){
    val toast = Toast.makeText(context, “Search done”, Toast.
```

```
    LENGTH_SHORT)
    toast.show()
    searchMoviesAdapter.updateMovies(movies)
}
fun onError() {
    val toast = Toast.makeText(context, “Search error”, Toast.
    LENGTH_SHORT)
    toast.show()
}
```

S ovim je uspješno napravljena pretraga u aplikaciji.

5.9.4. Zadatak 4

Prikazati poster filma umjesto slike žanra.

Potrebno je izmijeniti adapter i prilagoditi ga pretrazi. U njemu, ako je atribut `posterPath` postavljen, isti će biti prikaz u odgovarajućem *view*-u. Kako bi se prikazao poster, bit će korištena `Glide` biblioteka koja se dodaje preko *dependency*-a:

```
implementation ‘com.github.bumptech.glide:glide:+’
annotationProcessor ‘com.github.bumptech.glide:compiler:+’
```

`Glide` predstavlja učitavač slika za Android. Jednostavan je za korištenje i zahtijeva minimalnu konfiguraciju. Korištenjem `Glide`-a omogućen je prikaz slika dobijenih iz URL-a. Detaljnije o `Glide` se možete pronaći na: Glide.[38]

Postavljanje slike unutar adaptera:

```
val genreMatch: String? = movies[position].genre
val context: Context = holder.movieImage.getContext()
var id: Int = 0;
if (genreMatch!==null)
id = context.getResources()
    .getIdentifier(genreMatch, “drawable”, context.
    getPackageName())
if (id===0) id=context.getResources()
    .getIdentifier(“picture1”, “drawable”, context.
    getPackageName())
Glide.with(context)
```

[38] https://bumptech.github.io/glide/

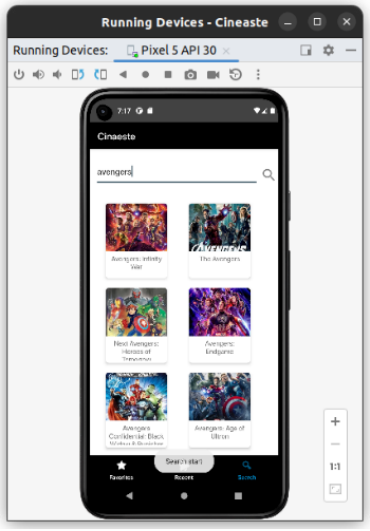
```
        .load(posterPath + movies[position].posterPath)
        .centerCrop()
        .placeholder(R.drawable.picture1)
        .error(id)
        .fallback(id)
        .into(holder.movieImage);
```

Varijabla `posterPath` predstavlja privatni atribut adaptera i ona glasi:

```
private val posterPath = “https://image.tmdb.org/t/p/w342”
```

Ona se dodaje s putanjom koju vrati web servis. Izvrši se `centerCrop` kako bi slika stala u okvir, te doda `placeholder`, `error` i `fallback` u slučaju da nije moguće učitati sliku. `error` i `fallback` su *default* slika ili slika po žanru.

Izgled aplikacije na kraju zadatka je:



Snimak zaslona 42. Pokrenuta aplikacija

5.10. Servisi

5.10.1. Zadatak 1

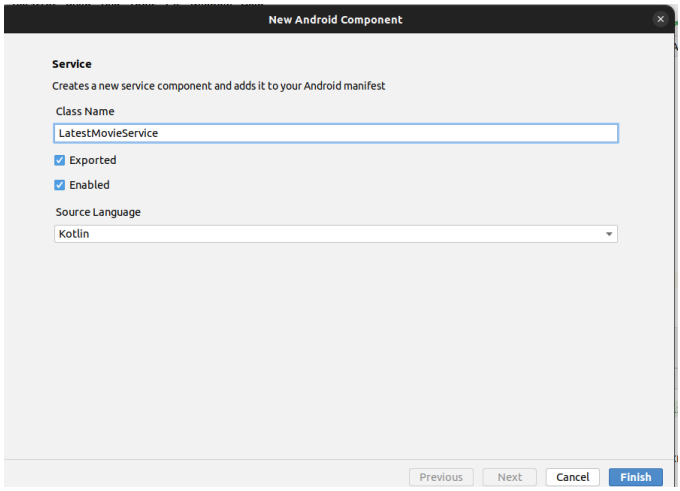
Kreirati servis u prvom planu koji je aktivan i kad aplikacija nije. Servis svaki sat vrši poziv ka web servisu koji vraća najnoviji dodani film u bazu. Nakon klika na notifikaciju koja se pojavi, otvara se nova simbolična aktivnost koja sadrži osnovne informacije o novom filmu.

U prvom koraku će se zahtijevati određene permisije u `manifest` datoteci.

```
<uses-permission android:name="android.permission.FOREGROUND_
SERVICE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.POST_
NOTIFICATIONS" />
```

Ovim se zahtijevaju permisije za pokretanje servisa u prvom planu i za `WAKE_LOCK` kako ne bi došlo do prekidanja servisa u `DOZE` ili `STANDBY` režimu rada.

Nakon zahtijevanja permisija, kreirati servis `New->Service->Service`, koji se zove `LatestMovieService`.



Snimak zaslona 43. Kreiranje servisa

Kako se servis ne veže za komponentu, tako se sljedeća metoda *override*-a:

```
override fun onBind(intent: Intent): IBinder? {
    return null
}
```

Definisati sljedeće privatne varijable:

```
//za sprecavanje prekida u slucaja Daze
private var wakeLock: PowerManager.WakeLock? = null
//oznaka da je servis pokrenut
private var isServiceStarted = false
```

```
//api kljuc
private val tmdb_api_key : String = BuildConfig.TMDB_API_KEY
//primjer filma- novi filmovi ne moraju sadrzavati sve podatke
private var movie =
Movie(1,”test”,”test”,”test”,”test”,”test”,”test”,”test”)
```

Override-ana `onCreate` metoda je:

```
override fun onCreate() {
    super.onCreate()
    val notification = createNotification()
    startForeground(1, notification)
}
```

U prvom koraku se kreira notifikacija da se traži novi film. Ova notifikacija će biti uvijek aktivna - što je po zahtjevima novijih verzija OS-a. Nakon toga se pokrene servis u prvom planu.

Metoda `createNotification` je:

```
private fun createNotification(): Notification {
    val notificationChannelId = “LATEST MOVIE SERVICE CHANNEL”

    val notificationManager = getSystemService(Context.
    NOTIFICATION_SERVICE) as NotificationManager
    //Kreiramo notifikacijski kanal - notifikaicje se salju na
    isti kanal
    val channel = NotificationChannel(
        notificationChannelId,
        “Latest Movie notifications channel”,
        NotificationManager.IMPORTANCE_HIGH
    ).let {
    //Definisemo karakteristike notifikacije
        it.description = “Latest Movie Service channel”
        it.enableLights(true)
        it.lightColor = Color.RED
        it.enableVibration(true)
        it.vibrationPattern = longArrayOf(100, 200, 300, 400,
        500, 400, 300, 200, 400)
        it
    }
    notificationManager.createNotificationChannel(channel)

    val builder: Notification.Builder =  Notification.Builder(
```

```
        this,
        notificationChannelId
    )
//Kreira se notifikacija koja ce se prikazati kada se servis
pokrene
    return builder
        .setContentTitle(“Finding latest film”)
        .setSmallIcon(android.R.drawable.ic_popup_sync)
        .setTicker(“Film”)
        .build()
}
```

Kada se servis pokrene poziva se `onStartCommand` metoda koja se *override*-a.

```
override fun onStartCommand(intent: Intent?, flags: Int,
startId: Int): Int {
    startService()
    // servis ce se restartovati ako ovo vratimo
    return START_STICKY
}
```

Unutar ove metode se poziva `startService` metoda.

```
private fun startService() {
//U slucaju da se ponovo pokrene ne mora se pozivati ova metoda
    if (isServiceStarted) return
//Postavimo da je servis pokrenut
    isServiceStarted = true
//Koristit cemo wakeLock da sprije&#x10D;imo ga&#x161;enje servisa
    wakeLock =
            (getSystemService(Context.POWER_SERVICE) as
            PowerManager).run {
                newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                “LatestMovieService::lock”).apply {
                    acquire()
                }
            }
    //Beskonacna petlja koja dohvati podatke svakih sat vremena
    GlobalScope.launch(Dispatchers.IO) {
        while (isServiceStarted) {
            launch(Dispatchers.IO) {
                getData()
            }
            //Sacekaj sat vremena prije nego se ponovo
```

```
            pokrene&#x161;
            delay(3600000)
        }
    }
}
```

U ovoj metodi poziv ka servisu je odvojen u *coroutine*. Već je ranije navedeno da se servisi ne izvršavaju na odvojenoj niti. Pošto poziv ka web servisu blokira rad aplikacije, potrebno ga je odvojiti u *coroutine* koja se odvija na pozadinskoj niti.

Nakon što se prikupi film iz poziva servisa, bit će kreirana nova notifikacija koja sa sobom sadrži i intent za otvaranje nove aktivnosti. Da bi se otvorila aktivnost na ovaj način, potrebno je koristiti `PendingIntent`.

`PendingIntent` je referenca na token koji sistem čuva i koji opisuje odgovarajuće podatke. U slučaju da se aplikacija ugasi i njeni procesi unište, `PendingIntent` će ostati i moći će se iskoristiti kasnije zajedno sa svim svojim podacima. On ostaje validan sve dok se ne ukloni.

U nastavku je dat prikaz poziva web servisa i definisanja `PendingIntent`.

```
private fun getData() {
    try {
        val url1 =
            “https://api.themoviedb.org/3/movie/latest?api_
            key=${tmdb_api_key}”
        val url = URL(url1)
        (url.openConnection() as? HttpURLConnection)?.run {
            val result = this.inputStream.bufferedReader().use {
            it.readText() }
            val jsonObject = JSONObject(result)
            movie.title = jsonObject.getString(“title”)
            movie.id = jsonObject.getLong(“id”)
            movie.releaseDate = jsonObject.getString(“release_
            date”)
            movie.homepage = jsonObject.getString(“homepage”)
            movie.overview = jsonObject.getString(“overview”)
            if (!jsonObject.getBoolean(“adult”)) {
                movie.backdropPath = jsonObject.
                getString(“backdrop_path”)
                movie.posterPath = jsonObject.getString(“poster_
                path”)
            }
```

```
        }
        val notifyIntent = Intent(this,
            MovieDetailResultActivity::class.java).apply {
            flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.
            FLAG_ACTIVITY_CLEAR_TASK
            putExtra("movie",movie)
        }
        val notifyPendingIntent = PendingIntent.getActivity(
            this, 0, notifyIntent, PendingIntent.FLAG_UPDATE_
            CURRENT
        )
        val notification = NotificationCompat.
        Builder(baseContext, "LATEST MOVIE SERVICE CHANNEL").
        apply{
            setSmallIcon(android.R.drawable.stat_notify_sync)
            setContentTitle("New movie found")
            setContentText(movie.title)
            setContentIntent(notifyPendingIntent)
            setOngoing(false)
            build()
        }
        with(NotificationManagerCompat.from(applicationContext)) {
            if (ActivityCompat.checkSelfPermission(
                    baseContext,
                    Manifest.permission.POST_NOTIFICATIONS
                ) != PackageManager.PERMISSION_GRANTED
            ) {
                return
            }
            notify(123, notification.build())
        }
    } catch (e: MalformedURLException) {
        Log.v("MalformedURLException", "Cannot open
        HttpURLConnection")
    } catch (e: IOException) {
        Log.v("IOException", "Cannot read stream")
    } catch (e: JSONException) {
        Log.v("IOException", "Cannot parse JSON")
    }
}
```

Cjelokupni servis je:

```
class LatestMovieService : Service() {
    private var wakeLock: PowerManager.WakeLock? = null
    private var isServiceStarted = false
    private val tmdb_api_key : String = BuildConfig.TMDB_API_KEY
    private var movie = Movie(1,”test”,”test”,”test”,”test”,”
    “,”test”,”test”)

    override fun onBind(intent: Intent): IBinder? {
        return null
    }

    override fun onStartCommand(intent: Intent?, flags: Int,
    startId: Int): Int {
        startService()
        return START_STICKY
    }

    override fun onCreate() {
        super.onCreate()
        val notification = createNotification()
        startForeground(1, notification)
    }

    private fun startService() {
        if (isServiceStarted) return
        isServiceStarted = true

        wakeLock =
            (getSystemService(Context.POWER_SERVICE) as
            PowerManager).run {
                newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                “LatestMovieService::lock”).apply {
                    acquire()
                }
            }

        // we’re starting a loop in a coroutine
        GlobalScope.launch(Dispatchers.IO) {
            while (isServiceStarted) {
                launch(Dispatchers.IO) {
                    getData()
                }
                delay(3600000)
            }
```

```
        }
    }

    private fun getData() {
        try {
            val url1 =
                “https://api.themoviedb.org/3/movie/latest?api_
                key=${tmdb_api_key}”
            val url = URL(url1)
            (url.openConnection() as? HttpURLConnection)?.run {
                val result = this.inputStream.bufferedReader().
                use { it.readText() }
                val jsonObject = JSONObject(result)
                movie.title = jsonObject.getString(“title”)
                movie.id = jsonObject.getLong(“id”)
                movie.releaseDate = jsonObject.
                getString(“release_date”)
                movie.homepage = jsonObject.
                getString(“homepage”)
                movie.overview = jsonObject.
                getString(“overview”)
                if (!jsonObject.getBoolean(“adult”)) {
                    movie.backdropPath = jsonObject.
                    getString(“backdrop_path”)
                    movie.posterPath = jsonObject.
                    getString(“poster_path”)
                }
            }
            val notifyIntent = Intent(this,
            MovieDetailResultActivity::class.java).apply {
                flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.
                FLAG_ACTIVITY_CLEAR_TASK
                putExtra(“movie”,movie)
            }
            val notifyPendingIntent = PendingIntent.getActivity(
                this, 0, notifyIntent, PendingIntent.FLAG_
                UPDATE_CURRENT
            )
            val notification = NotificationCompat.
            Builder(baseContext, “LATEST MOVIE SERVICE
            CHANNEL”).apply{
                setSmallIcon(android.R.drawable.stat_notify_sync)
                setContentTitle(“New movie found”)
                setContentText(movie.title)
```

```
                setContentIntent(notifyPendingIntent)
                setOngoing(false)
                build()
            }
            with(NotificationManagerCompat.
            from(applicationContext)) {
                if (ActivityCompat.checkSelfPermission(
                        baseContext,
                        Manifest.permission.POST_NOTIFICATIONS
                    ) != PackageManager.PERMISSION_GRANTED
                ) {
                    return
                }
                notify(123, notification.build())
            }
        } catch (e: MalformedURLException) {
            Log.v("MalformedURLException", "Cannot open
            HttpURLConnection")
        } catch (e: IOException) {
            Log.v("IOException", "Cannot read stream")
        } catch (e: JSONException) {
            Log.v("IOException", "Cannot parse JSON")
        }
    }

    private fun createNotification(): Notification {
        val notificationChannelId = "LATEST MOVIE SERVICE CHANNEL"

        val notificationManager = getSystemService(Context.
        NOTIFICATION_SERVICE) as NotificationManager
        val channel = NotificationChannel(
            notificationChannelId,
            "Latest Movie notifications channel",
            NotificationManager.IMPORTANCE_HIGH
        ).let {
            it.description = "Latest Movie Service channel"
            it.enableLights(true)
            it.lightColor = Color.RED
            it.enableVibration(true)
            it.vibrationPattern = longArrayOf(100, 200, 300,
            400, 500, 400, 300, 200, 400)
            it
        }
        notificationManager.createNotificationChannel(channel)
```

```
        val builder: Notification.Builder =  Notification.Builder(
            this,
            notificationChannelId
        )

        return builder
            .setContentTitle("Finding latest film")
            .setSmallIcon(android.R.drawable.ic_popup_sync)
            .setTicker("Film")
            .build()
    }
}
```

Kreirati sada novu aktivnost koja treba prikazati podatke o filmu. Bit će iskorišten *layout* koji je definisan za detalje o filmu bez liste sličnih filmova i glumaca.

Nova aktivnost će drugačije biti definisana u `manifest` datoteci. Potrebno je postaviti roditeljsku aktivnost, a zatim način pokretanja ove aktivnosti. Ova aktivnost je `Task` koji se pokrene jednom i koji se ne stavlja u `backstack` i čiji je zadatak samo da pokaže podatke o filmu.

```
<activity
    android:name=".MovieDetailResultActivity"
    android:parentActivityName=".MainActivity"
    android:launchMode="singleTask"
    android:taskAffinity=""
    android:excludeFromRecents="true"/>
```

Podaci iz `PendingIntent-a` se preuzimaju na isti način kao iz običnog.

Kôd nove aktivnosti `MovieDetailResultActivity`:

```
class MovieDetailResultActivity : AppCompatActivity() {

    private  var movie= Movie(0, "Test", "Test", "Test", "Test",
    "Test", "Test", "Test")
    private lateinit var title : TextView
    private lateinit var overview : TextView
    private lateinit var releaseDate : TextView
    private lateinit var genre : TextView
    private lateinit var website : TextView
    private lateinit var poster : ImageView
```

```
    private lateinit var backdrop : ImageView
    private val posterPath = “https://image.tmdb.org/t/p/w780”
    private val backdropPath = “https://image.tmdb.org/t/p/w500”

    @RequiresApi(Build.VERSION_CODES.TIRAMISU)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_movie_detail_result)

        title = findViewById(R.id.movie_title)
        overview = findViewById(R.id.movie_overview)
        releaseDate = findViewById(R.id.movie_release_date)
        genre = findViewById(R.id.movie_genre)
        poster = findViewById(R.id.movie_poster)
        website = findViewById(R.id.movie_website)
        backdrop = findViewById(R.id.movie_backdrop)
        val notificationManager = getSystemService(NOTIFICATION_
        SERVICE) as NotificationManager
        notificationManager.cancel(123)
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.
        TIRAMISU) {
            if(intent?.getParcelableExtra(“movie”, Movie::class.
            java)!==null) {
                movie = intent?.getParcelableExtra(“movie”,
                Movie::class.java)!!
                populateDetails()
            }
        } else {
            if (intent?.getParcelableExtra<Movie>(“movie”) !==
            null) {
                movie = intent?.getParcelableExtra<Movie>(“movie”)!!
                populateDetails()
            }
        }

    }

    private fun populateDetails() {
        title.text=movie.title
        releaseDate.text=movie.releaseDate
        genre.text=movie.genre
        website.text=movie.homepage
        overview.text=movie.overview
        val context: Context = poster.getContext()
```

```
        var id = 0;
        if (movie.genre!==null)
            id = context.getResources()
                .getIdentifier(movie.genre, “drawable”, context.
                getPackageName())
        if (id===0) id=context.getResources()
            .getIdentifier(“picture1”, “drawable”, context.
            getPackageName())
        Glide.with(context)
            .load(posterPath + movie.posterPath)
            .placeholder(R.drawable.picture1)
            .error(id)
            .fallback(id)
            .into(poster);
        var backdropContext: Context = backdrop.getContext()
        Glide.with(backdropContext)
            .load(backdropPath + movie.backdropPath)
            .centerCrop()
            .placeholder(R.drawable.backdrop)
            .error(R.drawable.backdrop)
            .fallback(R.drawable.backdrop)
            .into(backdrop);
    }

}
```

Ono što se može primijetiti u ovom kôdu jete da se film šalje kroz `Intent`, što ranije nije rađeno. Kako bi se poslala instanca klase pomoću intenta, klasa treba da implementira interfejs `Parcelable`. Samim tim klasa `Movie` će sada izgledati ovako:

```
data class Movie (
        var id: Long,
        var title: String,
        var overview: String,
        var releaseDate: String,
        var homepage: String?,
        var genre: String?,
        var posterPath: String,
        var backdropPath: String
):Parcelable {
    constructor(parcel: Parcel) : this(
            parcel.readLong(),
```

```
            parcel.readString()!!,
            parcel.readString()!!,
            parcel.readString()!!,
            parcel.readString(),
            parcel.readString(),
            parcel.readString()!!,
            parcel.readString()!!) {
    }
    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeLong(id)
        parcel.writeString(title)
        parcel.writeString(overview)
        parcel.writeString(releaseDate)
        parcel.writeString(homepage)
        parcel.writeString(genre)
        parcel.writeString(posterPath)
        parcel.writeString(backdropPath)
    }
    override fun describeContents(): Int {
        return 0
    }
    companion object CREATOR : Parcelable.Creator<Movie> {
        override fun createFromParcel(parcel: Parcel): Movie {
            return Movie(parcel)
        }
        override fun newArray(size: Int): Array<Movie?> {
            return arrayOfNulls(size)
        }
    }
}
```

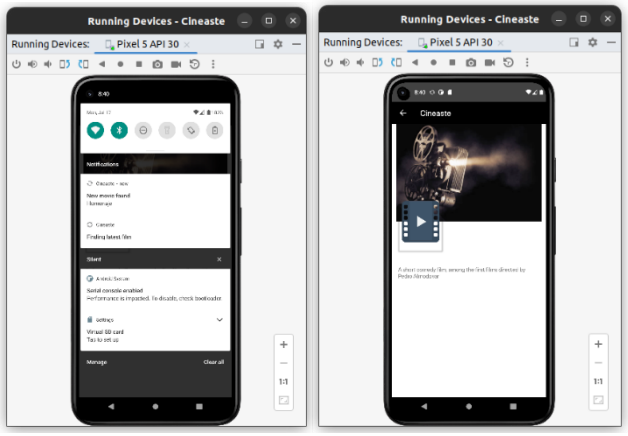
`Parcelable` predstavlja Android implementaciju Java `Serializable` klase.

Da bi se implementirao ovaj intefejs potrebno je implementirati metode `writeToParcel`, `describeContents`, konstruktor koji prima `Parcel`, te objekat `Parcelable.Creator`.

Za kraj još ostaje da se pokrene servis u glavnoj aktivnosti u `onCreate` metodi:

```
Intent(this, LatestMovieService::class.java).also {
    startForegroundService(it)
    return
}
```

Konačni izgled aplikacije je:



Snimak zaslona 44. Pokrenuta aplikacija

Napomena: API_KEY je sada sakriven u *BuildConfig*-u, dodana je varijabla u `local.properties`, a u `build.gradle` je unutar `build_types` dodano:

```
debug{
    buildConfigField ‘String’, “TMDB_API_KEY”, TMDB_API_KEY
}
```

5.11. Retrofit

5.11.1. Zadatak 1

Kreirati poziv pomoću Retrofit klijenta koji dohvata najnovije filmove. Prikazati filmove u RecentFragment-u.

U prvom koraku je potrebno dodati *dependency* u `gradle` datoteku.

```
implementation ‘com.squareup.retrofit2:retrofit:+’
implementation ‘com.squareup.retrofit2:converter-gson:+’
```

Klas `Movie` prilagoditi da bude serializible i da je konverter može pretvoriti u JSON i obratno:

```
data class Movie (
    @SerializedName(“id”) var id: Long,
    @SerializedName(“title”)  var title: String,
    @SerializedName(“overview”)  var overview: String,
```

```
    @SerializedName(“release_date”)   var releaseDate: String,
    @SerializedName(“homepage”)   var homepage: String?,
    @SerializedName(“poster_path”) var posterPath: String?,
    @SerializedName(“backdrop_path”)  var backdropPath: String?
)
```

Napomena: Obzirom da se žanr različito vraća u ovisnosti od poziva, izbaciti ga radi lakšeg rada.

Dodati još jednu klasu `GetMoviesResponse` koja predstavlja rezultate pretrage:

```
data class GetMoviesResponse(
    @SerializedName(“page”) val page: Int,
    @SerializedName(“results”) val movies: List<Movie>,
    @SerializedName(“total_pages”) val pages: Int
)
```

Kreirati interfejs `Api` u kojem će se čuvati odgovarajući pozivi korištenjem Retrofit klijenta:

```
interface Api {
    @GET(“movie/upcoming”)
    suspend fun getUpcomingMovies(
        @Query(“api_key”) apiKey: String = BuildConfig.TMDB_API_
KEY
    ): Response<GetMoviesResponse>
}
```

Trenutno se u interfejsu nalazi samo poziv za dobavljanje najnovijih filmova. Dokumentacija je data na: linku[39]

Kreirati singleton objekat u kojem će se nalaziti Retrofit instanca. Definisan je bazni URL web servisa, konverter i interfejs koji će klijent implementirati.

```
object ApiAdapter {
    val retrofit : Api = Retrofit.Builder()
        .baseUrl(“https://api.themoviedb.org/3/”)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
        .create(Api::class.java)
}
```

[39] https://developer.themoviedb.org/reference/movie-upcoming-list

U `MovieRepository` klasi je kreirana metoda koja će dohvatiti podatke unutar pozadinske *coroutine*. Korištenjem Retrofita se ne moraju koristiti *coroutine*, ali je sada dobra praksa koristiti ih.

```
suspend fun getUpcomingMovies(
) : GetMoviesResponse?{
    return withContext(Dispatchers.IO) {
        var response = ApiAdapter.retrofit.getUpcomingMovies()
        val responseBody = response.body()
        return @ withContext responseBody
    }
}
```

Unutar `RecentMoviesFragment`-a kreirana je metoda u kojoj se pokreće data *coroutine*-a :

```
fun getUpcoming( ){
    val scope = CoroutineScope(Job() + Dispatchers.Main)
    // Create a new coroutine on the UI thread
    scope.launch
        // Opcija 1
        val result = MovieRepository.getUpcomingMovies()
        // Display result of the network request to the user
         when (result) {
             is GetMoviesResponse -> onSuccess(result.movies)
             else-> onError()
         }
    }
}
```

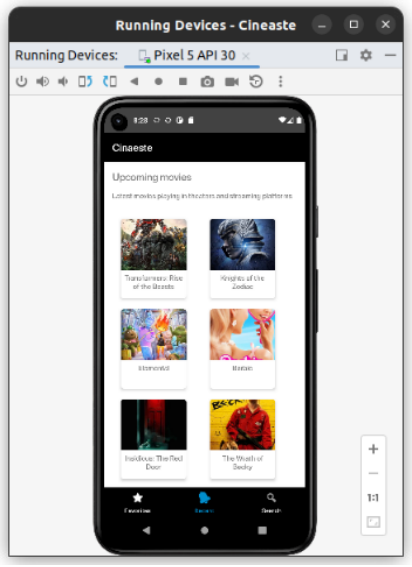
U fragmentu, u `onCreate` metodi će se izvršiti poziv ove metode, i implementirati funkcije `onScucess` i `onError`.

Njihov kôd je:

```
fun onSuccess(movies:List<Movie>){
    val toast = Toast.makeText(context, “Upcoming movies found”,
Toast.LENGTH_SHORT)
    toast.show()
    recentMoviesAdapter.updateMovies(movies)
}
fun onError() {
    val toast = Toast.makeText(context, “Search error”, Toast.
LENGTH_SHORT)
```

```
    toast.show()
}
```

Izgled aplikacije na kraju zadatka:



Snimak zaslona 45. Pokrenuta aplikacija

5.12. Perzistencija podataka u Android aplikacijama

5.12.1. Zadatak 1

Omogućiti da se klikom na odgovarajuće dugme u detaljima o filmu, film doda u bazu podataka kao omiljeni film. Film se treba prikazivati u listi omiljenih filmova.

U prvom koraku dodati odgovarajuće *dependency*-e u `gradle`.

```
implementation(“androidx.room:room-runtime:+”)
annotationProcessor “androidx.room:room-compiler:+”
implementation(“androidx.room:room-ktx:+”)
kapt(“androidx.room:room-compiler:+”)
```

Za potrebe `kapt` dodati i `plugin`:

```
id ‘kotlin-kapt’
```

Proširiti klasu `Movie` s odgovarajućim anotacijama:

```
@Entity
data class Movie (
        @PrimaryKey @SerializedName(“id”) var id: Long,
        @ColumnInfo(name = “title”) @SerializedName(“title”)
        var title: String,
        @ColumnInfo(name = “overview”) @
SerializedName(“overview”)  var overview: String,
        @ColumnInfo(name = “release_date”) @
SerializedName(“release_date”)   var releaseDate: String,
        @ColumnInfo(name = “homepage”) @
SerializedName(“homepage”)   var homepage: String?,
        @ColumnInfo(name = “poster_path”) @
SerializedName(“poster_path”) var posterPath: String?,
        @ColumnInfo(name = “backdrop_path”) @
SerializedName(“backdrop_path”)  var backdropPath: String?
)
```

Kreirati `MovieDAO` s metodama za dobavljanje svih filmova i za upis filmova u bazu (za sada):

```
@Dao
interface MovieDao {
    @Query(“SELECT * FROM movie”)
    suspend fun getAll(): List<Movie>
    @Insert
    suspend fun insertAll(vararg movies: Movie)
}
```

Kreirati `AppDatabase` klasu i u njoj dodati i `companion` objekat koji će instancirati bazu, te će se na taj način održati *singleton pattern*.

```
@Database(entities = arrayOf(Movie::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun movieDao(): MovieDao
    companion object {
        private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase {
            if (INSTANCE == null) {
                synchronized(AppDatabase::class) {
                    INSTANCE = buildRoomDB(context)
                }
            }
```

```
            return INSTANCE!!
        }
        private fun buildRoomDB(context: Context) =
            Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "cinaeste-db"
            ).build()
    }
}
```

Napomena: `Context` treba proslijeđivati kroz odgovarajuće metode.

Baza se može podesiti da blokira glavnu nit. Međutim, ovo se ne preporučuje u praksi, a u prethodno urađenom je naglašeno da je bolje koristiti *coroutine* i prebaciti sve na IO nit.

Definisati odgovarajuće metode za upis filma i dobavljanje filmova u/iz baze podataka unutar `MovieRepository` klase.

```
suspend fun getFavoriteMovies(context: Context) : List<Movie> {
    return withContext(Dispatchers.IO) {
        var db = AppDatabase.getInstance(context)
        var movies = db!!.movieDao().getAll()
        return @ withContext movies
    }
}
suspend fun writeFavorite(context: Context,movie:Movie) :
String?{
    return withContext(Dispatchers.IO) {
        try{
            var db = AppDatabase.getInstance(context)
            db!!.movieDao().insertAll(movie)
            return @ withContext "success"
        }
        catch(error:Exception){
            return @ withContext null
        }
    }
}
```

U `FavoriteMoviesFragment-u` definisati metodu za poziv metode iz repozitorija. U `onCreate` pozvati metodu i implementirati metode `onSuccess` i `onError`:

```
context?.let {
        getFavorites(it)
    }
```

```
fun getFavorites(context: Context){
    val scope = CoroutineScope(Job() + Dispatchers.Main)
    // Create a new coroutine on the UI thread
    scope.launch{

        // Make the network call and suspend execution until it
finishes
        val result = MovieRepository.getFavoriteMovies(context)

        // Display result of the network request to the user
        when (result) {
            is List<Movie> -> onSuccess(result)
            else-> onError()
        }
    }
}
```

```
fun onSuccess(movies:List<Movie>){
    favoriteMoviesAdapter.updateMovies(movies)
}
fun onError() {
    val toast = Toast.makeText(context, “Error”, Toast.LENGTH_
SHORT)
    toast.show()
}
```

Definisati i odgovarajuću metodu za upis filma u `MovieDetailActivity`, koja će se pozivati nakon klika na dugme (potrebno je dodati `button` za dodavanje u favorite), kao i metode za uspjeh i grešku:

```
addFavorite.setOnClickListener{
    writeDB(this,movie)
}
```

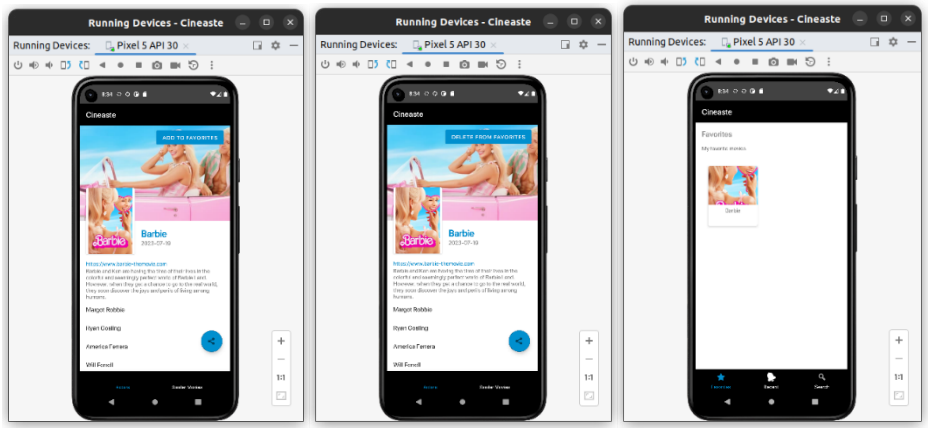
```
fun writeDB(context: Context, movie:Movie){
    scope.launch{
```

```
        val result = MovieRepository.
writeFavorite(context,movie)
        when (result) {
            is String -> onSuccess1(result)
            else-> onError()
        }
    }
}
```

```
fun onSuccess1(message:String){
    val toast = Toast.makeText(applicationContext, “Spaseno”,
Toast.LENGTH_SHORT)
    toast.show()
    addFavorite.visibility= View.GONE
}
fun onError() {
    val toast = Toast.makeText(applicationContext, “Error”,
Toast.LENGTH_SHORT)
    toast.show()
}
```

Kroz ovaj kôd je omogućeno dodavanje filma u omiljene i prikaz omiljenih filmova korištenjem SQLite baze podataka.

Izgled aplikacije na kraju zadatka je:



Snimak zaslona 46. Pokrenuta aplikacija

6.
REFERENCE

1. Ted Hagos, Learn Android Studio 3 with Kotlin: Efficient Android App Development, 2018.
2. Kotlin Language Specification (https://kotlinlang.org/spec/introduction.html)
3. Basic types (https://kotlinlang.org/docs/reference/basic-types.html)
4. Control Flow: if, when, for, while (https://kotlinlang.org/docs/reference/control-flow.html)
5. Exceptions (https://kotlinlang.org/docs/reference/exceptions.html)
6. Null Safety (https://kotlinlang.org/docs/reference/null-safety.html)
7. Functions (https://kotlinlang.org/docs/reference/functions.html)
8. Lambda functions (https://www.baeldung.com/kotlin-lambda-expressions)
9. Scope functions (https://kotlinlang.org/docs/reference/scope-functions.html)
10. Classes (https://kotlinlang.org/docs/reference/classes.html)
11. Data classes (https://kotlinlang.org/docs/reference/data-classes.html)
12. Sealed classes (https://kotlinlang.org/docs/reference/sealed-classes.html)
13. Mark L. Murphy, Android's Architecture Components, 2019.
14. Lifecycle (https://developer.android.com/jetpack/androidx/releases/lifecycle)
15. ViewModel Overview (https://developer.android.com/topic/libraries/architecture/viewmodel)
16. LiveData Overview (https://developer.android.com/topic/libraries/architecture/livedata)
17. Using Room Database | Android Jetpack (https://medium.com/mindorks/using-room-database-android-jetpack-675a89a0e942)
18. Peter Späth, Pro Android with Kotlin: Developing Modern Mobile Apps, 2018.
19. Yun Cheng & Aldo Olivares Domínguez, Advanced Android App Architecture, First Edition Real – world app architeture in Kotlin 1.3, 2019.
20. Android Architecture Patterns Part 1: Model–View–Controller (https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6)
21. Android Architecture Patterns Part 2: Model–View–Presenter (https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5)

22. Android Architecture Patterns Part 3: Model–View–ViewModel (https://upday.github.io/blog/model-view-viewmodel/)
23. About Layouts (https://developer.android.com/develop/ui/views/layout/declaring-layout)
24. Activity (https://developer.android.com/reference/android/app/Activity)
25. Resources (https://developer.android.com/reference/android/content/res/Resources)
26. Adapter (https://developer.android.com/reference/android/widget/Adapter)
27. Intents and intents filters (https://developer.android.com/guide/components/intents-filters)
28. Broadcasts overview (https://developer.android.com/guide/components/broadcasts)
29. Fragments (https://developer.android.com/guide/fragments)
30. Android Web Services (https://data-flair.training/blogs/android-web-services/)
31. Services overview (https://developer.android.com/guide/components/services)
32. Save data using SQLite (https://developer.android.com/training/data-storage/sqlite)
33. Save data in a local database using Room (https://developer.android.com/training/data-storage/room)
34. Android Jetpack (https://developer.android.com/jetpack?gclid=Cj0KCQjwiIOmBhDjARIsAP6YhSUaudINXhUEUU6CoygoQmh_CO3h_jdxP3N1UwZzoMd7s7nbDCryTvAaAnX6EALw_wcB&gclsrc=aw.ds)

Univerzitet u Sarajevu
Elektrotehnički fakultet Sarajevo

Recenzija rukopisa

RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

autora Vensade Okanović, Irfana Prazine, Šeile Bećirović Ramić i Jasmine Čeligije

1. Podaci o recenzentu:

red. prof. dr. SELMA RIZVIĆ, dipl. el. ing.

Pozicija: Redovni profesor

Radno mjesto: Univerzitet u Sarajevu - Elektrotehnički fakultet

Službeni telefon: +387 33 250 700

E-pošta: srizvic@etf.unsa.ba

2. Podaci o djelu:

Autori: **Vensada Okanović, Irfan Prazina, Šeila Bećirović Ramić, Jasmina Čeligija**

Naslov: **RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU**

Vrsta djela: Tehnička (naučno-stručna) literatura - udžbenik

Obim djela: Knjiga sadrži pet poglavlja i 219 stranica u formatu B5 uključujući sadržaj, uvod i literaturu. Knjiga referencira 33 djela (naslova) data u popisu literature.

3. Mišljenje-recenzija o djelu

Ova knjiga predstavlja važan doprinos nastavi na Odsjeku za računarstvo i informatiku Elektrotehničkog fakulteta Univerziteta u Sarajevu, jer će je studenti moći koristiti kao udžbenik iz predmeta Razvoj mobilnih aplikacija. Naravno, knjiga može podržati izučavanje ove oblasti na bilo kojem drugom tehničkom fakultetu. Obzirom da je programski jezik Kotlin jedan od vodećih jezika za razvoj mobilnih aplikacija, knjiga će omogućiti studentima da steknu znanja i vještine potrebne za rad u IT industriji. Detaljan opis jezika i mnoštvo primjera će im pomoći u savladavanju znanja i sticanju vještina programiranja mobilnih aplikacija. Knjiga prati predavanja i vježbe iz spomenutog predmeta, ali je korisna i čitaocima koji nisu studenti, a žele da savladaju ovaj programski jezik. Pisana je na razumljiv način i pokriva sve potrebne elemente za programiranje mobilnih aplikacija u Kotlin programskom jeziku.

Knjiga se sastoji iz sljedećih poglavlja:

1. Uvod

2. Kotlin

3. Android

4. Android Jetpack komponente

5. Dodatak - Praktični primjer

6. Reference

U Uvodu se opisuje motivacija za pisanje ove knjige, izbor programskog jezika Kotlin i Android platforme.

Poglavlje 2 predstavlja programski jezik Kotlin, razloge njegovog odabira i osnovne karakteristike. Opisuje ga kroz kontrolu toka programa, upravljanje izuzecima, null vrijednostima i funkcije. Na kraju poglavlja dat je opis klasa i njihovih funkcionalnosti.

U poglavlju 3 čitaoci se upoznaju sa Android operativnim sistemom i njegovom strukturom. Objašnjava se razvoj aplikacija sa raznim vrstama arhitektura i prezentiraju osnovne komponente/gradivni blokovi Android aplikacija. Prezentira se razvojna alatka Android studio.

Poglavlje 4 bavi se prezentacijom skupa biblioteka pod nazivom “Android Jetpack komponente”, koji čitaocima omogućava da maksimalno iskoriste sav potencijal Kotlin programskog jezika i pomaže im u razvoju kvalitetnih aplikacija koje je jednostavno održavati, nadgrađivati i testirati.

U poglavlju 5 naveden je detaljno opisani praktični primjer, koji je možda i najvažniji dio ove knjige. Tu se nalazi opis osnovnih principa razvoja moderne Android aplikacije (Cineaste) koja prikazuje najnovije filmove, omiljene filmove, detalje o filmovima, omogućava pretragu filmova putem TMDB API, te njihovo dodavanje u favorite. Ovo je način da studenti isprobaju sve što su naučili u prethodnim poglavljima.

4. Zaključak i preporuka

Zaključak: U skladu s prethodno iznesenim mišljenjem o ovome djelu, može se donijeti zaključak da ono u potpunosti ispunjava zahtjeve naučno-stručne literature za područje razvoja mobilnih aplikacija i nudi teoretski i praktični osnov za izučavanje te oblasti na tehničkim fakultetima.

Preporuka: Shodno prethodno iznesenom mišljenju o djelu (rukopisu) RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU autora Vensade Okanović, Irfana Prazine, Šeile Bećirović Ramić i Jasmine Čeligije, preporučujem objavljivanje navedenog djela kao knjige.

S poštovanjem,

prof. dr Selma Rizvić

Sarajevo, 9.8. 2023. god.

RECENZIJA RUKOPISA

RAZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

autora Vensade Okanović, Irfana Prazine, Šeile Bećirović Ramić i Jasmine Čeligija

Podaci o recenzentu:

V. prof. dr Belma Ramić-Brkić, dekan

belma.ramic@ssst.edu.ba

Sarajevo School of Science and Technology

Fakultet za Računarske nauke

Podaci o djelu:

Naslov: AZVOJ MOBILNIH APLIKACIJA U KOTLIN PROGRAMSKOM JEZIKU

Autori: Vensada Okanović, Irfan Prazina, Šeila Bećirović Ramić, Jasmina Čeligija

Obim djela: Knjiga sadrži šest poglavlja i 219 strana u B5 formatu

Vrsta djela: Tehnička (naučno-stručna) literatura

Mišljenje:

Knjiga "Razvoj Mobilnih Aplikacija u Kotlin Programskom Jeziku" pruža sveobuhvatan uvod u razvoj mobilnih aplikacija koristeći Kotlin, programski jezik za Android platformu. Autori detaljno pokrivaju osnove Kotlin jezika i objašnjavaju kako koristiti tehnologije kao što su Android Studio i Android SDK za razvoj visokokvalitetnih mobilnih aplikacija.

Impresionirana sam temeljnošću kojom su autori pristupili ovoj knjizi. Koraci su jasno objašnjeni, a primjeri su veoma lijepo koncipirani. Ova knjiga predstavlja odličan materijal kako za početnike, tako i sve one koji žele naučiti Kotlin jezik i primijeniti ga na razvoj mobilnih aplikacija.

Veoma su mi se svidjeli konkretni primjeri na kraju knjige, a obrađuju sve teme pokrivene u knjizi. Ovi primjeri pomažu čitaocima da efektivno primijene naučeno u stvarnom svijetu.

Smatram da su autori uspjeli napraviti izuzetno koristan resurs za sve koji se bave razvoje mobilnih aplikacija.

U skladu s gore iznesenim mišljenjem o djelu (rukopisu), toplo preporučujem njego objavljivanje kao knjige.

V. prof. dr Belma Ramić-Brkić

Belma Ramic-Brkic

U Sarajevu, 18.08.2023.