УНИВЕРЗИТЕТ У БАЊОЈ ЛУЦИ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Алексије Мићић

РАЗВОЈ АНДРОИД АПЛИКАЦИЈА КОРИШТЕЊЕМ ПРОГРАМСКОГ ЈЕЗИКА КОТЛИН

дипломски рад

Бања Лука, март 2020

Тема: **РАЗВОЈ АНДРОИД АПЛИКАЦИЈА КОРИШТЕЊЕМ ПРОГРАМСКОЈ ЈЕЗИКА КОТЛИН**

Кључне ријечи:
Андроид
Котлин
Јава

Комисија:
проф. др Славо Марић, предсједник
проф. др. Зоран Ђурић, ментор
Александар Келеч, ма, члан

Кандидат:
Алексије Мићић

УНИВЕРЗИТЕТ У БАЊОЈ ЛУЦИ

ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

КАТЕДРА ЗА РАЧУНАРСТВО И ИНФОРМАТИКУ

Предмет: МОБИЛНО РАЧУНАРСТВО

Тема: РАЗВОЈ АНДРОИД АПЛИКАЦИЈА КОРИШТЕЊЕМ ПРОГРАМСКОГ ЈЕЗИКА КОТЛИН

Задатак: Описати процес развоја Андроид апликација и подршку платформе за различите програмске језике. Детаљно описати програмски језик Котлин и извршити поређење са другим програмским језицима у домену развоја Андроид апликација. Анализирати развојна окружења и пратеће алате за рад са програмским језиком Котлин. У практичном дијелу рада реализовати Андроид апликацију која демонстрира коришћење описаног програмског језика.

Ментор: проф. др Зоран Ђурић

Кандидат: Алексије Мићић (1104/15)

Бања Лука, март 2020.

САДРЖАЈ

1. УВОД

У почетку приступ рачунарима имала су искључиво стручна лица која су прошла ригорозну обуку прије него што би били способни да их користе. Могућности тих првих рачунара биле су ограничене, а самим тим и њихова примјењивост. Појавом првих персоналних рачунара, ствари су се знатно промјениле. Сада су и обични људи имали могућност кориштења рачунара и њихову примјену за рјешавање одређених проблема. До нагле експанзије рачунара и рачунарства уопште долази почетком деведесетих година, када се развио интернет као глобална комуникациона мрежа.

Развој интернета праћен је и све већом глобализацијом тржишта, јер сада су се могли без већих препрека обављати послови на далеко, што су многе компаније искористиле како би се пробили на нова тржишта, али и за проналазак квалитетне радне снаге у другим дијеловима свијета. Потреба да се ова комуникација додатно упрости, али и надогради јесте довела до убрзаног развоја мобилних телефона (мобилних рачунара).

Први мобилни телефони имали су ограничене могућности као што је био случај са првим персоналним рачунарима, те је њихова примјена била атрактивна за пословни свијет, а не толико и за обичне кориснике. Како су се исти нагло развијали, а самим тим и њихове могућности за рјешавање различитих проблема, као и за забаву (кроз видео игре, фотографисање и видео снимање) унапријеђивале то су они постајали атрактивнији и за свакодневну употребу, чиме је и потражња за овим уређајима нагло порасла. Највећа експанзија десила се са појавом тзв. паметних телефона (енг. Smartphones), чији је први представник био ајфон (енг. iPhone) развијен од стране компаније Епл (енг. Apple). Након појаве овог уређаја на тржишту, конкурентни су имали потребу да понуде сличан производ што је и довело до стварања првих паметних телефона базираних на Андроид оперативном систему. Паметнии телефони нуде одређен скуп основних функционалности за корисника, а тај скуп функционалности је додатно проширен преко апликација (апликативних програма) које за потребе корисника развијају програмери. Иза ових апликација могу да стоје велике компаније, али и самостални програмери. Овакав приступ је омогућио компанијама које се баве развојем телефона и ОС (оперативног система) да се фокусирају на унапријеђивање основних функционалности телефона, а да остатак посла препусте у руке апликативних програмера.

За развој апликација постоје у зависности од оперативног система различити алати као и различити програмски језици. Када је ријеч о Андроиду главни програмски језик за развој апликација био је Јава. Овај језик је био примарни и препоручени избор све до 2019. године када је Гугл (енг. Google), као компанија која стоји иза Андроида одлучио да је од сада препоручени језик за развој модерних апликација Котлин. У питању је модеран програмски језик, који пружа адекватну подршку за развој нове генерације апликација.

У другој глави рада тема је Андроид платформа, односно оне компоненте које чине ову платформу. Прво је објашњено шта је Андроид ОС, на чему је базиран шта су његове предности и мане, те је под посебном тачком је обрађен проблем фрагментације. Под другим параграфом је обрађена архитектура Андроид платформе као и појединачне компоненте које ју чине. У трећем параграфу су обрађене врсте мобилних апликација које су подржане на платформи, као и њихове предности и мане.

У трећој глави рада обрађен је Котлин програмски језик, објашњени су основни концепти везани за Котлин, од којих су неки слични онима из Јаве, док су неки други искључиво везани за Котлин. У првом параграфу су објашњени неки основни концепти

Котлин програмског језика, као што су нулабилни типови података и корутине, али и основни концепти који се јављају у већини модерних програмских језика. У другом параграфу су наведени и објашњени они Котлин концепти који су везани за рад са функцијама, посебан акценат стављен је на функције екстензије које имају своју широку примјену на Андроид платформи. У трећем параграфу обрађени су концепти везани за ООП, од којих су многи слични онима присутним у Јави.

У четвртој глави рада обрађено је стандардно развојно окружење које се користи за писање Андроид апликација, Android Studio. Објашњен је поступак креирања новог пројекта и разне опције које су програмерима при томе на располагању, као и неке од могућности које ово напредно окружење нуди апликативним програмерима.

У петој глави рада разматране су различити архитектурални стилови који се користи приликом развоја Андроид апликација, узете су обзир њихове предности и мане, подржаност од стране платформе, лакоћа имплементације и тестабилност пројеката.

У шестој глави рада обрађен је процес развоја Андроид апликација. У првом параграфу се разматра Android Jetpack као скуп стандардних библиотека које су ту да у многоме олакшају процес писања апликација, али и дају недвосмислене смјернице програмерима. У другом параграфу се разматра развој корисничког интерфејса апликације и опција које су на располагању програмерима, посебан акцент је стављен на ConstraintLayout као модерни интерфејс који замијењује многе раније опције, али и на RecyclerView који се користи када се ради са колекцијом података, а број елемената колекције је непознат и промјењљив. У трећем параграфу је обрађен поступак навигација кроз екране унутар апликације, односно Navigation API који је дат у склопу Jetpack-а и у многоме поједностављује навигацију у односу на старија рјешења, те омогућава раздвајање компоненти по улози. У четвртом параграфу су обрађене опције за чување података на Андроид платформи, од рада са фајловима до рада са базом података, а посебан акцент је стављен на Room. Room је ORM који олакшава рад са SQLite базом података. Под петим параграфом је обрађен рад са мрежом, разамтране су неке од библиотека које су у оптицају за те сврхе и њихове предности и мане. У шестом параграфу су описани сви они кораци који се морају предузети прије него што се крене са дистрибуцијом апликације крајњим корисницима.

У седмој глави рада дато је кратко објашњене апликација које су практично имплементиране користећи се свим оним концептима који су обрађени у теоријским сегментима. Прва апликација служи као школски примјер употребе MVI архитектуралног стила и неких Jetpack библиотека, а друга апликација јесте школски примјер употребе MVVM архитектуралног стила као и рада са мрежом кориштењем Retrofit клиејнта. Кроз оба пројекта кориштене су Котлин функционалности попут корутина, класа података, функција екстензија, затворених класа итд.

У осмој глави рада дат је кратак осврт на обрађене теме у раду као и коментар и разматрања нивоа адаптације Котлина за развој Андроид апликација те о томе шта се може очекивати у неком будућем периоду.

2. АНДРОИД ПЛАТФОРМА

2.1. Андроид оперативни систем

Андроид је оперативни систем намијењен за мобилне уређаје. Ријеч је о Unix-like оперативном систему базираном на LTS[1] верзијама Линукс кернела. Може се користити на већини мобилних уређаја, укључујући поред мобилних телефона и таблет, лаптоп, читач електронских књига, ручне сатове, паметне телевизоре итд. Иако је Андроид базиран на Линукс кернелу он не посједује стандардни X Windows System нити стандардни скуп GNU библиотека па није у стању да покреће апликације развијене за друге стандардне Линукс системе.

Компанија Android Inc. је основана у октобру 2003. године у Калифорнији. Њихова прва идеја је била да се развије оперативни систем за дигиталне камере, али како је то тржиште било и сувише мало, фокус је пребачен на развој оперативног система за мобилне телефоне. Компанија Гугл која је препознала потенцијал овог пројекта, купује Android Inc. у јулу 2005. године, при чему и челни људи из развјонг тима прелазе да раде за Гугл, у склопу Андроид пројекта. Андроид је развијан као оперативни систем за мобилне уређаје који су имали физичку тастатуру, но појава Ајфона на тржишту је довела до промјене фокуса, при чему је сада био циљ да се дода подршка за уређаје са екраном осјетљивим на додир. Крајем 2007. године Гугл оснива алијансу OHA (Open Handset Alliance) чији је први производ био Андроид. Алијансу поред Гугла чине и друге велике компаније из свијета телекомуникације, микрочипова, као и произвођачи мобилних телефона. Коначно 23. септембра 2008. године на тржиште излази прва верзија Андроид оперативног система, која је била намијењена за уређаје HTC Dream и T-Mobile G1. Ова верзија је имала доста недостатака, те Андроид у овој фази још увијек не представља правог конкурента за OSX оперативни систем који користи Ајфон. Убрзан развој Андроида почиње са појавом верзије 1.5 која носи кодни назив „Cupcake”[2]. Свака нова верзија система донијела је са собом низ побољшања постојећих могућности, али и многе нове функционалности које нису до тада постојале, у табели 2.1 су наведене све досадашње верзије.

Табела 2.1 Верзије Андроид оперативног система [1]

Кодни назив	Број верзије	Датум објаве	АПИ Ниво
Нема	1.0	23. септембар 2008	1
	1.1	9. фебруар 2009	2
Cupcake	1.5	27. април 2009	3
Donut	1.6	15. септембар 2009	4
Eclair	2.0-2.1	26. октобар 2009	5-7

[1] LTS (Long-term support) је полиса одржавања животног вијека софтвера код које се стабилна верзија софтвера одржава на дужи временски период од стандардне верзије. Овај приступ се користи како би се смањила могућност регресије, односно појаве грешака у коду услијед додавања нових функционалности или проширивања постојећих. [2]

[2] Почевши од ове верзије, Гугл је одлучио да свакој наредној верзији да кодни назив на основу неког слаткиша. Ово је била пракса све до августа 2019. када је објелодањено да ће наредне верзије користити нумерички формат, па је и прва наредна верзија система названа „Андроид 10“. [1]

Froyo	2.2-2.2.3	20. мај 2009	8
Gingerbread	2.3-2.3.7	6. децембар 2010	9-10
Honeycomb	3.0-3.2.6	22. фебруар 2011	11-13
Ice Cream Sandwich	4.0-4.0.4	18. октобар 2011	14-15
Jelly Bean	4.1-4.3.1	9. јул 2012	16-18
KitKat	4.4-4.4.4	31. октобар 2013	19-20
Lollipop	5.0-5.1.1	12. новембар 2014	21-22
Marshmallow	6.0-6.0.1	5. октобар 2015	23
Nougat	7.0-7.1.2	22. август 2016	24-25
Oreo	8.0-8.1	21. август 2017	26-27
Pie	9.0	6. август 2018	28
Android 10	10.0	3. септембар 2019	29

2.1.1. Проблем фрагментације

Андроид ОС (оперативни систем) се користи на мноштву уређаја, у питању је једна отворена платформа коју могу да користе сви и коју свако може да прилагоди за своје потребе. Ово јесте уједно и највећа предност Андроида у односу на конкурента у виду OSX и Ајфон уређаја. Данас имамо само годишње на стотине нових различитих уређаја који се користи Андроидом, купци имају избор из широког асортимана уређаја чије су карактеристике и функционалности као и цијена прилагођени њиховим потребама. То значи да се Андроидом користе како телефони са великим, тако и они са малим екранима, како они са најбољим перформансама, тако и они са осредњим перформансама. Већина произвођача додатно проширује функционалности Андроида за уређаје који они пласирају на тржиште кориштењем тзв. Skins од којих су можда најпознатији: Samsung One UI, EMUI, MIUI и Android One. Могућност избора је пак уједно и главна мана Андроид ОС. Док Епл годишње представи један нови уређај, при чему се могу максимално посветити да њихов ОС оптимизују за баш тај уређај, тако нешто није могуће на Андроиду. Ово додатно усложњава ствари када је ријеч о ажурирању, сигурности, али и развијању и дистрибуцији апликација. Наиме иако је Гугл задњих година веома конзистентан при представљању нових верзија ОС[3], већина произвођача знатно каска за њима, углавном због тога што је потребно њихов Skin прилагодити новој верзији система. Неки од телефона који имају слабије хардверске перформансе једноставно не могу да се користе са новом верзијом ОС, али како је технологија, а самим тим и хардвер телефона напредовао ово све ређе представља проблем, јер су сада често и они приступачнији телефони много снажнији него што су то били и најскупљи уређаји од прије пар година. Са овим се додуше почео јављати један нови тренд и проблем, а то је да људи једноставно немају потребу да тако често мијењају своје мобилне уређаје као што су то некада раније чинили. Док су до пар година уназад нови уређаји нудили знатно боље перформансе у односу на претходну генерацију и мноштво то тада невиђених нових функционалности, данас су унапријеђена углавном инкрементална као што је то на примјер случај код персоналних рачунара. Због свега овога

[3] Почевши од 2016. године Гугл организује конференцију за програмере у августу када се представља нова верзија ОС, али и разна друга унапријеђења која су ту да олакшају развој апликација и прошире постојеће могућности.

је присутна огромна фрагментација у виду различитих верзија Андроида које су присутне на тржишту. Сваке године у првој седмици маја Гугл врши мјерења на основу који се утврђује заступљеност појединих верзија Андроид на тржишту, резултати посљедњен мјерења су приказани у табели 2.2, при чему су приказане само оне верзије Андроида које имају преко 0.1% удио на тржишту.

Табела 2.2 Фрагментација Андроид ОС [3]

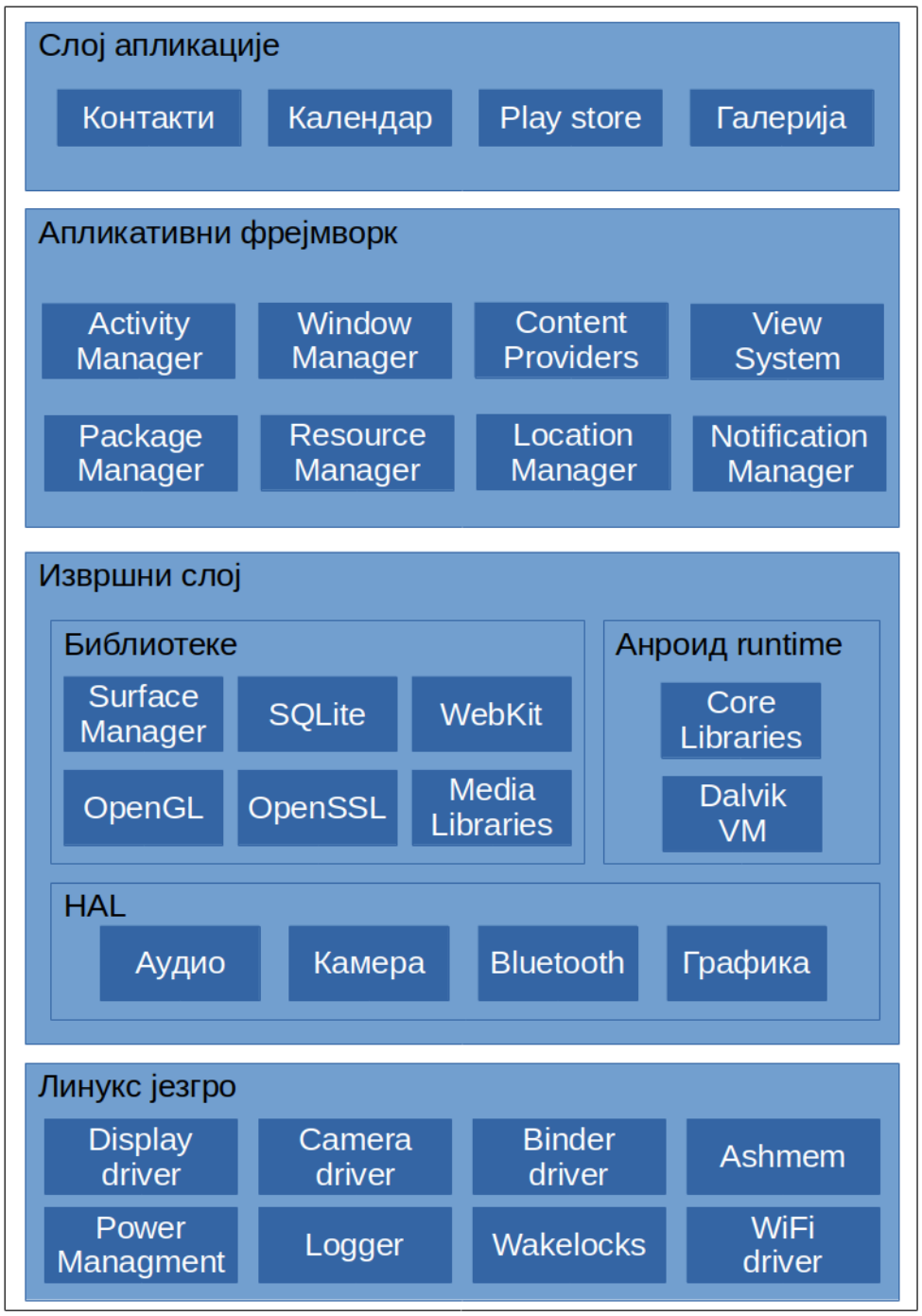
Кодни назив	АПИ ниво	Дистрибуција
Gingerbread	10	0.30%
Ice Cream Sandwich	15	0.30%
Jelly Bean	16	1.20%
	17	1.50%
	18	0.50%
KitKat	19	6.90%
Lollipop	21	3.00%
	22	11.50%
Marshmallow	23	16.90%
Nougat	24	11.40%
	25	7.80%
Oreo	26	12.90%
	27	15.40%
Pie	28	10.40%

За апликативне програмере ово представља низ проблема. Прво потребно је да своје апликације прилагоде великом броју уређаја, што отежава процес тестирања корисничког интерфејса апликације. Друго, не смију се олако користити новим функционалностима које су додате у некој од новијих верзија Андроида, јер често велики број корисника који нема ту верзију система неће моћи да се користи том функционалношћу, у таквим ситуацијама је често потребно понудити алтернативни приступ за онај дио корисника који нема одговарајућу верзију система. Ово додатно продужава развојни процес, тако да се често мора донијети одлука до које верзије система желимо имати подршку уназад. Тренутно је препорука да се не развијају апликације које подржавају верзије система старије од KitKat.

Како би се изборио са овим проблемом Гугл је покушао да примјени низ рјешења од којих нису баш сва била успјешна. Сигурносна унапријеђења, као и функционалности Гуглових апликација који долазе стандардно са већином Андроид телефона се ажурирају неовисно о Андроид платформи, коришћењем Google Play сервиса. Уведене су и смјернице за дизајн апликација кроз Material Design смјернице, како би се постигао универзалан изглед и олакшао рад крајњим корисницима, али и дизајнерима. Осмишљен је Android One програм, гдје Гугл даје низ смјерница произвођачима телефона, како би они на тржиште пласирали уређаје који су више у складу са стандардном Андроид дистрибуцијом и самим тим много брже добијају најновија ажурирања, овај програм је имао мјешовит успијех до сада. Од великог значаја је још и Project Treble којим је омогућено бржи циклус ажурирања, јер је њиме додан нови интерфејс између основног Андроид система и Skin-а произвођача, што у основи значи да се ова два слоја сада могу ажурирати неовисно један о другом. Као резултат имамо продужен животни циклус за старије телефоне који сада добијају ажурирања дужи временски период. [4]

2.2. Архитектура Андроид платформе

Архитектура Андроид платформе састоји се из четири слоја: Линукс језгро (енг. Linux Kernel), извршни слој (енг. Runtime Layer), апликативни фрејмворк (енг. Application Framework) и слој апликација (енг. Applications), приказаних на слици 2.1.



Слика 2.1 Архитектура Андроид платформе

2.2.1. Линукс језгро

Развој програмског језгра које ће директно комуницирати са уређајима је скуп и дуготрајан процес, због тога се развојни тимови често одлучују да користе постојећа рјешења отвореног кода. Из истог разлога је Андроид платформа заснована на Линукс језгру, као најнижем софтверском слоју који директно комуницира са хардвером уређаја. Овај слој одговоран је за управљање: процесима, меморијом, напајањем уређаја, фајл системом, драјверима, мрежом и сл.

Основна идеја Андроид ОС-а јесте да се извршава на мањим уређајима, који имају ограничене процесорске и меморијске ресурсе, као и напајање. Линукс језгро морало је бити прилагођено овим потребама. У складу с тим, уведене су одређене промјене у постојеће Линукс језгро, од којих су значајније:

- Binder, драјвер који обезбјеђује подршку за комуникацију међу процесима (енг. Inter Process Communication) и удаљено позивање метода (енг. Remote Method Invocation),
- Ashmem, механизам дијељења меморије прилагођен мобилним уређајима,
- Power Management, механизам за ефикасније управљање напајањем,
- Logger, механизам за евидентирање (енг. Logging) информација и порука од различитих компнената ОС, али и апликација гдје је значајан са становишта откривања грешака у коду, али и за аналитику,
- Wakelocks[4], механизам који омогућава апликацијама да обављају одређене задатке када се уређај не користи,
- LMK (Low Memory Killer), механизам за насилно прекидање процеса који се дуже вријеме не користе или који имају низак приоритет, зарад оптималне искоришћености меморијских ресурса,
- PMEM (Process Memory Allocator), драјвер који управља процесом пресликавања физичке меморије у корисничком простору,
- Paranoid network security, механизам за рестрикцију мрежне комуникације,
- Alarm timer, механизам који омогућава покретање неког процеса из корисничког простора у одговарајућем временском тренутку итд.

2.2.2. Извршни слој

Извршни слој Андроид платформе наслања се на слој кернела и састоји се од библиотека (енг. Libraries), Далвик виртуелне машине (енг. Dalvik Virtual Machine – Dalvik VM) и слоја корисничког простора (енг. Native Userspace).

Библиотеке чине спону између Линукс језгра и апликативног фрејмворка. Имплементиране су у програмским језицима C и C++, а апликативним програмерима су изложени кроз одговарајуће Јава и Котлин API-је. Компоненте и апликације које су написане у програмским језицима Јава и Котлин комуницирају са нативним компонентама помоћу Јава нативног интерфејса (енг. Java Native Interface – JNI). Библиотеке се учитавају

[4] Један од тих задатака јесте и синхронизација апликације са сервером, али проблем настаје онда када апликације сувише дуго користе овај механизам или га користе и онда када он није пријеко потребан, што негативно утиче на потрошњу батерије уређаја. [5]

динамички и обезбјеђују различите функционалности процесима који се извршавају на Андроид платформи. Неке од билбиотека су:

- Bionic libc, омогућава програмерима да пишу нативне сервисе за Андроид,
- Media Framework, омогућава стримовање аудио и видео садржаја у позадини,
- FreeType, приказивање битмап и векторских фонтова,
- WebKit, користе га разни веб читачи како на мобилним уређајима тако и на персоналним рачунарима.
- OpenGL ES, SGL, библиотека која нуди механизме за цртање 2Д и 3Д графике и често се користи у видео играма,
- SQLite, омогућава складиштење података у релациону базу података,
- OpenSSL, библиотека за сигурну комуникацију путем интернета итд.

Далвик ВМ је главна компонента извршног слоја, која је задужена за интеракцију са апликацијама и њихово извршавање на Андроид платформи. Далвик је виртуелна машина развијена специјално за Андроид платформу, уводећи оптимизације у два веома битна сегмента за која се стандардна Јава виртуелна машина (енг. Java Virtual Machine – JVM) није показала као ефикасно рјешење, а то су животни вијек батерије и мања процесорска моћ Андроид уређаја. Други разлог зашто је развијен Далвик ВМ јесте због лиценцирања. Иако су Јава програмски језик и пратеће Јава библиотеке бесплатне, сама виртуелна машина је у власништву компаније Оракл (енг. Oracle)[5]. Развојем потпуно бесплатне ВМ отвореног кода и са одговарајућим лиценцирањем[6] Андроид је охрабрио многе да га усвоје и користе на различитим типовима уређаја без потребе за бригом око лиценцирања.

Котлин или Јава изворни код се прослиједе компајлеру који генерише одговарајући Јава бајт код, а исти се прослиједи DEX-компајлеру који генерише Далвик бајт код који се може извршавати на Далвик ВМ, као што је и приказано на слици 2.2.

Слика 2.2 Процес компајлирања на Андроид платформи

[5] Компанија Оракл је 2010. године тужила Гугл због развоја Далвик-а наводећи као разлог кршење ауторских и патентних права. [6]

[6] Apache лиценца. [7]

Свака Андроид апликација се извршава у посебном процесу, са својом иначицом Далвик ВМ. Далвик не користи стек као што је то случај код стандардне Јава ВМ већ је базиран на регистрима. Овај приступ резултује у 47% мање извршених инструкција у односу на стандардни Јава ВМ, али је регистарски код 25% већи него одговарајући стек базиран код, у пракси ово повећање у линијама кода је занемариво. [8]

Постоје двије верзије Далвик ВМ, које користе различите приступе у превођењу и извршавању Далвик бајткода:

- Далвик ВМ која користи динамичко превођење бајткода у машински код током извршавања апликације (енг. Just In Time compilaton - JIT). Ова верзија Далвик ВМ се користила на уређајима са Андроид верзијом старијом од верзије 5.0,
- Далвик ВМ која користи АРТ (енг. Android RunTime - ART) прије-времено превођење тј. бајткод се преводи у машински код током инсталације апликације (енг. Ahead of Time Compilation - AOT). Овај приступ користи се од верзије 5.0 Андроид ОС-а. АРТ пружа боље перформансе, али уз употребу већег простора за складиштење апликација.

Слој корисничког простора обухвата све компоненте које функционишу изван Далвик ВМ, а не припадају слоју Линукс језгра. Најзначајнија компонента овог слоја је HAL (енг. Hardware Abstraction Layer) , која омогућава проширење карактеристика платформе и додавање подршке за нове хардверске компоненте. Генерално, на Линукс платформи, драјвери за хардвер су уграђени у језгро или се динамички учитавају у виду модула. Међутим, Андроид примјењује нешто другачији приступ када је у питању подршка за нови хардвер, тако што се за сваки тип хардвера дефинише АПИ који ће користити виши слојеви приликом интеракције са овим типом хардвера. На тај начин, одговорност за обезбјеђивање софтверског модула који ће имплементирати АПИ за одговарајући тип хардвера, пребачена је на произвођача тог типа хардвера. Ово рјешење је омогућило да се у Андроид језгро не уграђују сви могући драјвери и да се не користи динамичко учитавање модула, чиме се постижу значајна побољшања када су у питању перформансе Андроид уређаја. Компонента која обезбјеђује ову функционалност јесте управо HAL.

Додатно, слој корисничког простора састоји се и од низа извршних фајлова који су компајлирани за извршавање на циљаној процесоркој платформи. Покрећу се аутоматски за вријеме booting процеса, или их покреће init процес, или их програмери покрећу кроз командну линију. Ови извршни фајлови обично имају висок ниво привилегија, чиме им је омогућен директан приступ системским компонентама и нативним библиотекама.

2.2.3. Андроид фрејмворк

Андроид фрејмворк је скуп АПИ-ја који омогућава развојним програмерима да брзо и једноставно пишу Андроид апликације. Овај слој прилагођава функционалности извршног слоја за употребу у апликацијама. У извршном слоју се користе првенствено програмски језици C и C++, а апликације су писане у Котлину или Јави, задатак апликативног фрејмворка је да премости ове разлике. Из тог разлога ово је најбоље документован дио Андроид платформе. Такође садржи алате за дизајнирање корисничког интерфејса као што су дугмићи, поља за унос текста, слике и сл. као и разне системске алате. Основни сервиси које пружа овај слој су:

- Activity Manager, контролише животни циклус апликације и њених активности,
- Location Manager, пружа приступ сервисима на основу којих се може одредити географска локација уређаја,
- Notifications Manager, омогућава апликацијама да прикажу кориснику упозорења и нотификације,
- Content Providers, омогућава апликацијама да дијеле своје податке са другим апликацијама,
- Window Manager, пружа интерфејс преко кога апликације могу да користе управљач прозорима, у чијој је надлежности организација Z-поретка листе прозора, који прозори су видљиви и на који начин су презентовани на екрану.
- Package Manger, управља са инсталацијом, брисањем и надоградњом апликација итд.

Поред наведених, фрејмворк пружа библиотеке за приступ различитим функционалностима уређаја, као што су телефонирање, слање порука, приступ уграђеним сензорима, Bluetooth, WiFi, итд.

2.2.4. Слој апликације

Слој апликације је највиши слој Андроид платформе, у ком се налазе све апликације које се извршавају на уређају и обезбјеђује највише фунцкионалности за крајње кориснике Андроид уређаја. Овај слој обухвата уграђене апликације, познате и као системске апликације као и апликације развијене од стране трећих лица, које крајњи корисници инсталирају на уређаје преко различитих канала дистрибуције, од којих је вјероватно најпознатији Google Play Store.

Андроид апликације су углавном написане у програмском језику Котлин, ако је ријеч о новијим апликацијама, или у програмском језику Јава ако је ријеч о старијим апликацијама. Могуће је користити и друге програмске језике за развој Андроид апликација. Апликације се дистрибуирају у форми Андроид апликативног пакета (енг. Android Application Package APK), тј. фајла који има екстензију .apk, а представаља ЗИП архиву. Чине га следеће компоненте:

- Далвик бајткод, односно већи број .dex фајлова, по један за сваку класу,
- разни ресурси, као што су фајлови који описују графички интерфејс апликације (.xml), векторска графика, фајлови са подацима за локализацију и сл.,
- (опционално) нативне библиотеке,
- Андроид манифест датотека (eng. AndroidManifest.xml) која садржи кључне информације о апликацији, овде се на примјер наводи која Активност треба да се прикаже приликом покретања апликације, али и упити за разне дозволе као што је приступ интернету, контактима и сл.

Алтернативно за дистрибуцију апликација може се користити app bundle. App bundle је дизајниран за велике и комплексне апликације, гдје ће увијек само дио оног што бива смјештено у АПК бити релевантно за неки дати уређај. Примјер гдје је ово корисно, јесте код познатих апликација као што су оне од друштвених мрежа, гдје постоји много фајлова са подацима за локализацију, али корисници углавном не користе више од једног или два језика. Сви остали преводи непотребно заузимају простор на диску. App bundle омогућава

Гуглу да креира АПК за апликацију који је прилагођен потребама индивидуалних корисника и њихових уређаја. Проблем са овим приступом је што развојни програмер више нема контролу над тим шта бива спаковано у његову апликацију. Јер иако је наведено шта све може да се одстрани из апликације, нема никаквих навода о томе да ли се ишта у њу додаје или на неки начин се модификује њено понашање.

2.3. Врсте мобилних апликација

Развојни програмери се могу одлучити за развој једне од три врсте мобилних апликација, а то су: веб апликације, нативне апликације и хибридне апликације. Избор увелико зависи од потреба, али и рестрикција неког развојног тима или клијента за чије сврхе се апликација развија.

2.3.1. Веб апликације

Веб апликације се покрећу преко веб читача и обично су написане коришћењем стандардних веб технологија тј. HTML, CSS и Јаваскрипта на клијентској страни и неког програмског језика на серверској страни нпр. Јава, Котлин, PHP, Python и сл. Веб апликације апликације се "инсталирају" тако што корисник креира обиљеживач (енг. bookmark) до одговарајућег сајта.

Главне предности веб апликација су:

- Ове апликације чувају све податке на серверској страни па тиме не оптерећују диск крајњег корисника,
- Најуниверзалније су јер веб читаче користи огроман број различитих уређаја,
- Аутоматско ажурирање апликације и садржаја, који се постављају на серверској страни,
- Коришћене технологије су добро познате са мноштвом пратеће литературе,
- Цијена развоја оваквих апликације је најчешће најмања јер постоји само једна база кода за све платформе,
- Нема рестрикција које намеће дистрибутер, нити се дијели дио зараде.

Главне мане веб апликација су:

- Како се подаци чувају на серверској страни, за употребу ових апликација је потребна мрежна конекција, а ако је у питању спорија конекција, то додатно може да отежа интеракцију са апликацијом,
- Нема интеракције са локалним ресурсима,
- Мањи број GUI контрола и мање моћне контроле које немају нативан изглед за дату платформу,
- Трендови у развоју веб апликација често се мијењају,
- Лоше перформансе у поређењу са друга два типа апликација,
- Не постоји пречица на радној површини,
- Неефикасна комуникација, само HTTP(s) протокол,
- Једини вид монетизације јесте кроз рекламе и модел претплатника.

2.3.2. Нативне апликације

Нативне апликације су развијене за неку специвичну платформу, па су оне нативне за ту платформу. За развој нативних апликација обично се користи један програмски језик као што је Котлин, Јава, Swift. Инсталирају се преко стандардних канала за дистрибуцију, као што је Google Play Store или Apple App Store.

Главне предности нативних апликација су:

- Разноврсне и моћне GUI контроле које имају нативни изглед за дату платформу,
- Приступ свим могућим API-јима које дата платформа излаже, као и свим локалним ресурсима као што су позиви, поруке, контакти, локалне базе, геолокација, микрофон, камера итд.,
- Интернет конекција није увијек потребна, у зависности од типа апликације,
- Најбоље перформансе од свих врста апликација,
- Боље и интуитивније корисничко искуство,
- Дизајниране за рад са малим екранима осјетљивим на додир,
- Ефикасније комуникација јер је могуће користити било који мрежни протокол, али и сервисе попут NFC, Bluetooth и сл.,
- Монетизација, кроз рекламе, куповине у апликацији или продаја саме апликације по некој цијени.

Главне мане нативних апликација су:

- Нису универзалне, нативна апликација која је написана за Андроид платформу не може да се покрене на Ајфон уређајима и обрнуто. Потребно је одржавати двије различите кодне базе, а често и упослити већи број радника чиме цијена развоја додатно расте,
- Сложеније је ажурирање апликација, јер је потребна интервенција корисника,
- Мањи број програмера,
- Заузимају простор на диску.

2.3.3. Хибридне апликације

Рјешење које је између нативних и веб апликација јесу тзв. хибридне апликације. Хибридне апликације користе стандардне веб технологије, али се извршавају унутар WebView компоненте. Инсталирају се преко стандарних канала дистрибуције, као и нативне апликације.

Главне предности хибридних апликација су:

- За развој се користе добро познате технологије,
- Јефтиније су од нативних апликација и универзалније јер се користи једна кодна база за све платформе,
- Могућа је интеракција са локалним ресурсима и неким од API-ја,
- Монетизација је могућа на исти начин као код нативних апликација.

Главне мане хибридних апликација су:

- Спорије су од нативних апликација,
- Скупље су од веб апликација,
- Заузимају простор на диску,
- Технологије које се користе за писање хибридних апликација развијају се неовисно од платформе те је самим тим њихова подршка упитна на дуге стазе,
- Интерфејс хибридних апликација нема нативан изглед, али ако је нативан изглед апликације један од услова који се мора остварити тада цијена развоја хибридних апликација достиже или чак премашује цијену развоја посебних нативних апликација за сваку од подржаних платформи,
- Специфични захтјеви теже се имплементирају него код нативних апликаиција,
- Немају приступ баш свим API-јима као и нативне апликације.

3. КОТЛИН ПРОГРАМСКИ ЈЕЗИК

Котлин[7] је вишеплатформски, статички[8] програмски језик опште намјене. Дизајниран је да има потпуну интероперабилност са Јава програмским језиком. Котлин код може да се пише за Јава ВМ, али могуће га је и компајлирати у Јаваскрипт као и директно у машински код коришћењем LLVM компајлера. На Андроид платформи додатно се врши компајлирање Јава бајткода помоћу Далвик компајлера у одговарајући .dex формат, као што је то случај и за стандардни Јава код.

У јулу 2011. године компанија JetBrains је објавила да ради на новом програмском језику за Јава ВМ, те да је пројекат у развоју задњних годину дана. Мотивација за развој новог програмског језика потиче од потребе за продуктивнијим програмским језиком, који би задовољио у то вријеме интерне потребе у компанији. Језик Scala који се такође извршава на Јава ВМ је по ријечима челних људи у компанији задовољавао њихове потребе све до једне, а то је знатно спорије компајлирање кода него што је то случај за Јаву. Један од првих захтјева односно циљева Котлин програмског језика јесте да вријеме компајлирања буде приближно исто као код Јаве. Пројекат је постао отворени код (енг. Open source) у фебруару 2012 године. Наравно поред повећања продуктивности постоје и други разлози за развој новог језика, а то су у првом реду:

- Очекивање да ће Котлин поспијешити продају JetBrains развојних алата. Није случајност да је писање Котлин кода најлакше и најбоље подржано управо у њиховим развојним алатима. Прогамери много лакше доносе одлуку да промјене своје развојно окружење него језик у коме пишу свој код, ако би се јавио нови програмски језик који је боље подржан у неком другом развојном алату и представља добар избор за програмере, то би могло негативно да утиче на тржишни удио JetBrains развојних алата,
- Миграција програмера на програмски језик који је директно под њиховом контролом, дирекције у развоју језика су тиме у складу са потребама и могућностима њихових алата, али је ово уједно и најбржи начин да се осигура прилагођавање алата за унапријеђења која језик доноси кроз нове верзије,
- Повећање угледа компаније код програмерске заједнице, чиме расте и шанса да ће се програмери односно фирме и менаџмент одлучити да купе и неке друге алата ове компаније који можда немају директне везе са писањем кода (TeamCity, Space, YouTruck и сл.). Сличан приступ користи Мајкрософт (енг. Microsoft) са .NET платформом и својим програмским језицима и пратећим развојним алатима.

У табели 3.1 су приказани најважнији догађаји у животном циклусу Котлина.

---

[7] Име Котлин потиче од истоименог острва у близини Санкт Петербурга, дизајнери су одлучили да додијеле име језику по неком острву као што је случај за Јаву која носи назив по истоименом индонежанском оствру.

[8] Статичка провјера куцања захтјева унапријед познавање сваког типа варијабле, такође компајлер врши валидацију да поља и варијабле које су наведене у коду у стварности постоје, што спријечава грешке у коду. Ово је у контрасту са језицима динамичке природе, као што је нпр. Python или Groovy на Јава ВМ, код којих се разрјешавање типа података обавља за вријеме извршавања програма, мана овог приступа је да се грешке попут погрешног навођења имена варијабле не могу открити за вријеме компајлирања, али и генерално спорије извршавање кода написаног у овим језицима.

Табела 3.1 Догађаји у животном циклусу Котлина

Датум	Догађај
Јул 2011.	Компанија JetBrains објављује да ради на новом програмском језику
Фебруар 2012.	Котлин је објављен под лиценцом отвореног кода
Фебруар. 2016	Верзија 1.0 програмског језика Котлин
Март 2017.	Верзија 1.1 програмског језика Котлин
Мај 2017.	Гугл објављује да је Котлин подржан заједно са Јавом за развој Андроид апликација
Новембар 2017.	Верзија 1.2 програмског језика Котлин
Октобар 2018.	Верзија 1.3 програмског језика Котлин
Мај 2019.	Гугл објављује да је пожељно писати све нове Андроид апликације у Котлину

3.1. Основе

3.1.1. Варијабле изразрзи и константе

Са становишта Котлина све представља објекат, односно не постоје примитивни типови као у Јави. Постоје сви исти нумерички типови као у Јави: Byte, Short, Int, Long, Float, Double. Цијели бројеви у Котлину су подразумијевано Int вриједности, да би се број означио као нпр. Long потребно је навести суфикс L након броја: 1234 је Int, а 1234L је Long. Децимални бројеви подразумијевано су типа Double, а за тип Float је потребно након броја навести суфикс F. Постоје сви исти математички оператори као у Јави, те су за стандардне типове имплементиратни на исти начин као у Јави. За логичке операције користи се тип Boolean, а постоје и сви логички оператори који постоје у Јави: &&, ||, !.

3.1.2. Колекције

Котлин стандардна библиотека садржи разне колекције које се могу користити и из Јаве, а ако не постоји подршка за неку колекцију директно из Котлина, тада је могуће користити одговарајућу колекцију из Јава стандардне библиотеке или неке друге библиотеке написане за Јава ВМ. Неке од важнијих колекција су:

- Array, класа која пружа имплементацију аналогну јава низовима, те у позадини користи управо обичне низове Јава ВМ, мање се користе него листе. За основне типове података постоје посебне класе нпр: IntArray, BooleanArray и сл., користе се ријетко, углавном када је потребно радити са Јава кодом гдје се користе низови,
- List, аналогно ArrayList у Јави,
- Set, колекција која не допушта уметање дупликата,
- Map, аналогно HashMap у Јави

Колекције се праве позивом collectionOf(<elements>) функција нпр. listOf(), mapOf(). За приступање елементима колекције користи се оператор [], који за разлику од Јаве није само примјењив на обичне низове. У позадини се при навођењу оператора [] позива get() функција. Листе, мапе и скупови су колекције које садрже непромјењиве елементе, тако да

се оператор [] не може позивати над овим колекцијама, али може над промјењивом варијантом ових колекција које се добијају позивом mutableCollectionOf() функција. [9]

3.1.3. Гранање и петље

if/else if/else у Котлину представља израз, исто као што је то 2 + 2 и уједно и замјену за тернарни оператор. Као повратна вриједност се узима резултат извршавања задње линије унутар гране која се извршава услијед испуњена услова. У Котлину се ријетко користе if/else if/else ирази који имају else if гране, умјесто тога користи се when израз који представља унапријеђену замјену за switch из Јаве и других језика. За писање петљи ту су стандардне while, do while и for петља.

3.1.4. Коментари и документација

Коментари у Котлину се наводе на исти начин као у Јави, једина је разлика у томе што се блоковски коментари могу угњеждавати и што се за писање документације користи KDoc, који за разлику од JavaDoc користи Markdown за форматирање текста, с тим да проширује подршку за навођење линкова. Да би се изгенерисала документација користи се dokka алат.

3.1.5. Изузетци

Изузетци у Котлину се не морају провјеравати као што је то случај у Јави, иако је обавезна провјера изузетака својевремено била занимљива замисао, она није заживјела јер доноси више проблема него што их ријешава (референца). Поред овога једина разлика је што try / catch представља израз, као и if / else. Повратна вриједност је задња линија унутар try блока или ако се десио изузезатак задња линија catch блока који је ухватио изузетак. Након try / catch може се навести finally блок који се извршава неовисно о томе да ли је бачен изузезетак или не и углавном има за задатак да се почисти након try, нпр. да се затворе сви отворени токови и сл.

3.1.6. Нулабилност

NullPointerException један је од најчешћих изузетака у програмском језику Јава. Котлин овај проблем рјешава тако што разликује референце којима се не може додијетли вриједност null и референце којима се може додијелити null вриједност, при чему компајлер захтјева од програмера додатну предострожност при раду са оваквим типовима података. Главна идеја јесте, да се програмери обесхрабре и избјегавају писати код који би могао да изазове NullPointerException односно, да ријеђе раде са таквим типовима података.

```
var a: String = "abc"
val l1 = a.length // ok
a = null // compilation error
var b: String? = "abc"
b = null // ok
val l2 = b.length // compilation error
```

Слика 3.1 Типови референци у Котлину

Као што се види са слике 3.1 не може се додијелити null вриједност првој референси, али се зато може без проблема позвати нека функција над том референсом, с друге стране другој референси се може додијелити null вриједност, али се не може фунцкија позвати над њом само тако. Да би позвали функцију над нулабилном референсом потребно је да се осигура да она не показује на null, што се може провјерити изразом гранања, или употребом safe-call оператора (?.) и Елвис оператора (?:), оба приступа приказана су на слици 3.2.

```
var b: String? = "abc"
println(if (b != null) b.length else -1) // izraz grananja
println(b?.length ?: -1) // safe-call i Elvis operator
```

Слика 3.2 Позивање фунцкије над нулабилном рефернесом

Safe-call оператора ће вратити резултат извршавања функције над неком референсом ако она не показује на null, у супротном ће вратити null вриједност, а Елвис оператор ће вратити резултат извршавања израза на својој десној страни ако израз није null, а ако јесте тада ће вратити резултат извршавња израза на лијевој страни.

Коначно када је програмер сигуран да референса која је нулабилна не показује на null или не жели да провјерава вриједност референсе може искористити null-assertion оператор (!!), којим се дозвољава позив функције над том референсом, али по цијену да се деси NullPointerException при извршавању програма.

3.1.7. Преклапање оператора

Неки програмски језици попут C++ или Котлина омогућавају преклапање одређених оператора, други програмски језици попут Swift-a омогућавају и дефинисање скроз нових оператора поред оних стандарних, док се у трећима попут Јаве не могу дефинисати нови, а ни преклапати постојећи. Главна критика ове функционалности јесте што програмери имају тенденцију да користе преклапање оператора на погрешан и само њима својствен начин, што код чини неразумљивијим и отежава исправљање грешака, поготово ако се тестирање врши под претпоставком да оператори раде на неки одређен начин, а то није случај. Зато су дизајнери Јаве одлучили да онемогуће ову функционалност, док су пак дизајнери Котлина одлучили да програмерима понуде могућност преклапања уско ограниченог скупа оператора. Примјера ради у Котлину је немогуће помножити ниску са неким бројем и компајлер ће пријавити грешку, ако се тако нешто покуша у коду. Котлин не зна како да помножи ниску са цијелим бројем, али са преклапањем је могуће додати ту функционалност, као на слици 3.3.

```
operator fun String.times(count: Int) : String {
    val sb = StringBuilder()
    for (i in 1..count) {
        sb.append(this)
    }

    return sb.toString()
}

fun main() {
    println("Foo" * 3)
}
```

Слика 3.3 Преклапање оператора

3.1.8. Корутине

На Андроид платформи постоји главна апликациона нит, која је задужена за кориснички интерфејс. Ако је потребно да се кориснички интерфејс ажурира са новим подацима то се мора учинити преко главне нити, али проблем настаје ако се сав код извршава на главној нити, када је потребно да се подаци учитају са неког удаљеног извора или да се дохвања велика количина података из базе података, извршавање ових операција чије трајање није детерминистичко ће блокирати кориснички интерфејс за било какву интеракцију. Овај проблем се рјешава тако што се све операције које су процесорски интензивне или на чији резултат се мора чекати извршавају у некој позадинској нити, а да се онда резултат извршавања врати главној нити, како би се из ње могао ажурирати кориснички интерфејс, дакле пише се код који се асинхроно извршава.

Котлин корутине пружају једноставан механизам за писање асинхроног кода и дио су Котлин програмског језика од верзије 1.3, мада су многи механизми везани за корутине још увијек у експерименталној фази те се очекује да ће постати стабилни у некој наредној верзији Котлина. За рад са корутинама додата је нова кључна ријеч suspend, сви остали елементи који чине корутине долазе из библиотека које је потребно додати у пројекат. Када се фунцкија маркира са suspend то значи да се њен процес извршавања може суспендовати на неко вријеме и касније поново наставити, ако нека функција треба да дохвати податке преко мреже онда ће се њено извршавање суспендовати док ти подаци не буду спремни, а у међувремену се могу обављати други задаци. Све корутине се извршавају у неком опсегу видљивости, који је под директном контролом неког CoroutineScope објекта, чија је улога да прекида извршавање неке корутине или обрише корутине када је то потребно. Најдуже трајање има GlobalCoroutineScope, али се у пракси треба користити неки мањи ниво видљивости, јер нема смисла да се настави за извршавањем рада корутине, ако се изашло из оног опсега за који је она везана. У контектсу Андроид платформе највише се користи viewModelScope, који представља особину екстензију ViewModel класе, која је везана за неку активност или фрагмент, када се позове onClear() функција неког екрана тада се и прекида извршавање свих корутина које су биле везане за тај екран. За конструисање корутине користе се Coroutine builders, који прихватају неки ламбда израз који представља посао који та корутина треба да извршава. То су функције које се позивају над неком CoroutineScope инстанцом и враћају Job објекат који представља посао који треба да се уради. Овим функцијама се може прослиједити Dispatcher, којим се наводи мјесто извршавања корутине, дакле да ли се посао извршава на главној нити или на некој од споредних. Подразумијевано се користи Dispatchers.Default односно посао се врши у некој позадинској нити која је оптимизована за процесорски интензивне задатке. За рад са фајловима, базом података и мрежом користи се Dispatchers.IO, а за једноставније задатке који се могу извршавати на главној нити користи се Dispatchers.Main.

Кориштење suspend функција је добро када се не очекује повратна вриједност или је повратна вриједност тачно један објекат, али када се ради са током података потребно је користити друге механизме, а то су Flow и Channel. Први се користи за оне токове који емитују податке само онда када постоји неко ко их ослушкује, а други за оне токове који емитују податке безобзира на то да ли неко у датом тренутку ослушкује или не. Flow се често користи на Андроид платформи у раду са базом података. [10]

3.2. Функције

Котлин функције имају све оне функционалности карактеристичне за Јава методе, али поред тих основних функционалности Котлин доноси низ унапријеђена за функције које га чине посебним. Примјер двеју функција дат је на слици 3.4.

```
fun sayHello(name: String) {
    println("Hello, $name")
}

fun add(first: Int, second: Int) : Int {
    return first + second
}
```

Слика 3.4 Примјер Котлин функција

Дакле функција се дефинише као: *fun <fun name>(<parameters>) : <return type>*. Повратну вриједност функције није потребно навести и тада компајлер сматра да је повратна вриједност типа *Unit* који одговара кључној ријечи *void* у Јави. Функције се позивају исто као у Јави, тако што наведемо назив фунцкије и у заградама вриједности које прослиједјујемо параметрима, ако их функција има.

У ситуацијама када је тијело функције у ствари један велики израз у Котлину је могуће користити се *expression body* синтаксом, гдје се тијело функције наводи након знака једнакости (=) умјесто помоћу витичастих заграда. Предност овог приступа није само у мањем броју карактера јер се више не мора ни наводити кључна ријеч *return* која се сада подразумијева сама од себе, већ у томе што компајлер може сам да одреди тип повратне вриједности функције па се и тип више не мора наводити. Наравно ако желимо да тип повратне вриједности одговара некој родитељској класи онда га је могуће и експлицитно навести. Додатна предност је у томе што се изрази гранања могу писати помоћу ове синтаксе, као што се види са слике 3.5.

```
fun example(num: Int) = when {
    num % 2 == 0 || num % 3 == 0 -> "Result 1"
    num % 5 == 0 -> "Result 2"
    else -> "Result 3"
}

fun makeSound(animal: Animal) = when(animal) {
    is Dog, is Wolf -> println("Woof")
    is Cat -> println("Meow")
    is Chicken -> println("Cluck cluck")
    is Duck -> println("Quack")
    else -> println("Animal noises")
}
```

Слика 3.5 Функција са изразом као тијело

Котлин подржава и додјелу подразумијеваних вриједности параметрима функције, при чему се параметри са подразумијеваним вриједностима наводе након свих осталих. Да би се у Јави постигао исти ефекат потребно је преклопити функцију, што резултује са више линија кода. Функције у Котлину је могуће позивати и са експлицитно наведеденим параметрима и вриједностима које ти параметри треба да поприме, тада није више битан редослијед параметара, што олакшава рад са функцијама које имају подразумијеване вриједности параметара. Ако је дата функција као на слици 3.6, те је поред вриједности за обавезне параметре потребно још само додијелити вриједност за параметар *path* без навођења имена параметра позив ове функције би изгледао као на првој линији наведеној на слици 3.7, коришћењем именованих параметара скраћујемо позив на варијанту из друге линије, а како је могуће поштовати редослијед параметара до првог именованог параметра можемо додатно скратити позив ове функције на варијанту у трећој линији кода.

```
fun registerUser(
    username: String,
    password: String,
    hasModeratorPrivileges: Boolean = false,
    hasAdminPrivileges: Boolean = false,
    expires: LocalDateTime? = null,
    domain: String? = null,
    path: String? = null,
    secure: Boolean = false
) {
    // do something
}
```

Слика 3.6 Функција са подразумијеваним вриједностима параметара

```
registerUser("username", "password", false, false, null, null, "path") // 1
registerUser(username="username", password="password", path="path") // 2
registerUser("username", "password", path="path") // 3
```

Слика 3.7 Позив фунцкије са и без именованих параметара

Котлин омогућава додјелу назива функције између `` карактера, при чему је дозвољено фунцкије називати ријечима одвојеним размаком. Овај приступ није посебно користан за функције које се пише да би се користиле у продукцији, јер позивање оваквих функција је напорније него позивање функција именованих на стандардни начин, али има своју примјену при писању јединичних тестова. Наиме за именовање јединичних тестова постоје одређене конвенције, како би се могло одма на основу назива јединичног теста закључити шта се то тестира и у чему је проблем, када тест не пролази, наравно за компликованије тестове није могуће такве закључке донијети на прву, али добро именован тест свакако да олакшава разријешавање проблема. Једна конвенција за именовање JUnit тестова је:

<јединична операицја>_<стање које се тестира>_<очекивани резултат>

Примјера ради, ако је потребно тестирати функцију која треба да обрне редослије карактера у нискоj, један од граничних случајева је празна ниска, па је очекиван резултат поново празна ниска, коришћењем camel case-а односно стандардног именовања имамо:

reverse_EmptyString_EmptyStringReturned()

Коришћењем `` именовања имамо:

`reverse empty string and return empty string`()

Други приступ је прегледнији, а и како се јединични тестови позивају коришћењем рефлексије, а не директно, немамо проблем који постоји у продукцији. Битно је за нагласити да оваква синтакса не може да се користи са кодом који се ослања на Јава код и библиотеке, тако да на Андроид платформи није могуће користити `` за средње тестове. [9]

3.2.1. Ламбда изрази

Ламбда изрази у Котлину су слични онима у Јави, представљају комад кода који се извршава између витичастих заграда, у Котлину је популарна парадигма функционалног програмирања па се ламбде много чешће користе него у Јави. Ламбда изрази као и функције могу да прихватају параметре, а како су честе ламбде које примају тачно један параметар постоји посебна синтакса за рад са таквим ламбдама гдје се тај параметар наводи преко кључне ријечи *it*, примјери ламбда израза дати су на слици 3.8.

```
val multiply = { a : Int, b : Int -> a * b}

// izvršavamo lambdu pozivom funkcije invoke
val result1 = multiply.invoke(10, 12)

// zbog preklapanja operatora nije potrebno
// eksplicitno navoditi invoke()
val result2 = multiply(15, 17)

val things = listOf("foo", "bar", "goo")

things.forEach { thing -> println(thing) } // suvišno
things.forEach { println(it) } // it sintaksa

things
.map { it.toUpperCase() }
.forEach { println(it) }
```

Слика 3.8 Примјер ламбда ираза

У стандардној библиотеци су имплементирани одређени ламбда изрази који се често користе у функционалном програмирању, тако да постоје ламбде за редукцију, мапирање, филтрирање и друге. Предност ламбда ираза јесте у томе да се на резултат извршавања једне ламбде може надовезати позив неке наредне, а да се при томе добија једна јасна синтакса.

Код ламбда израза који имају више линија кода, као повратна вриједност се узима резултат извршавања послиједње линије. Ламбде које „немају“ повратну вриједност враћају *Unit*.

3.2.2. Scope функције

Котлин стандардна библиотека девинише шест тзв. *Scope* функција, чија је једина улога да изврше комад кода у контексту неког објекта над којим се позивају. Унутар тог контекста није потребно наводити име објекта, јер се оно подразумијева на неки начин који зависи од тога о којој функцији је ријеч, те функције су:

- *let()*, функција која се користи када желимо да се неки код изврши, ако објекат над којим се ова функција позива није *null*. Објекат над којима је функција позвана се прослиједи ламбда изразу, те како по дефиницији та вриједност не може да буде *null* објекат се унутар ламбде може користити без рестрикција. Посљедња линија унутар ламбда израза представља повратну вриједност ове функције. Када је *null* објекат, тада се код унутар ове функције уопште не извршава,
- *apply(),* користи се за једноставније подешавање објекта, умјесто да позивамо неки сетер над објектом, ми позовемо ову функцију над њим, ламбда изразу се тада прослиједи објекат као *this*, односно као тренутни објекат, што значи да се сетери могу позивати без да се референцира објекат, што чини код прегледнијим, такође је могуће убацити неке друге изразе у тијело ове функције. Повратна вриједност је сам објекат,
- *run(),* користи се када нам је потребна функционалност коју пружа *apply()*, а желимо да се врати резултат извршавања посљедње линије унутар ламбда ирзаза као што је то случај код *let(),*
- *with(),* слична функционалност као *run(),* разлика је само у томе што се ова функција не позива над објектом, него прихвата објекат као параметар, па се у њеном тијелу мора оградити од могућности да је као параметар прослиђена *null* вриједност,
- *also(),* позива се над објектом и прослиједује дати објекат као параметар свом ламбда изразу, а као повратну вриједност враћа сам објекат, представља комбинацију *let()* и *apply(),*
- *use(),* може само да се позива над објектима чија класа имплементира *Closable* интерфејс, прослиједује објекат над којим је позвана свом ламбда изразу, те као повратну вриједнсоти ма резултат извршавања посљедње линије кода унутар ламбда израза, након чега се над датим објектом позива функција *close()*, при чему ће се ова функција позвати без обзира да ли је дошло до изуетка приликом извршавања кода унутар ламбда израза, у позадини се користи *try/finally*. Користна је функција за рад са ресурсима, спријечава цурење података. [9]

У табели 3.2 су приказане разлике међу овим функцијама.

Табела 3.2 Разлике међу scope функцијама

Функција	Начин позивања	Референцирање објекта	Повратна вриједност
let()	Над објектом	it	Резултат извршавња блока
apply()	Над објектом	this	Објекат над којим се позива
run()	Над објектом	this	Резултат извршавња блока

with()	Објекат параметар	this	Резултат извршавња блока
also()	Над објектом	it	Објекат над којим се позива
use()	Над Closable	it	Резултат извршавња блока

3.2.3. Функционално програмирање

Котлин подржава писање кода помоћу парадигме функционалног програмирања, али за разлику од неких језика гдје је ово једини начин да се пише код, у Котлину је овакав стил писања опционалан, с тим да га велики број програмера користи и заговара, јер олакшава вишенитност, али и чине функције разумљивијим, како оне више не смију имати бочне ефекте, па се са сигурношћу зна да ће функција увијек вратити исти резултат када јој се прослиједе исти подаци. Котлин поджава писање ламбда фунцкија, рад са токовима података, али и функције вишег реда. Функције вишег реда су функције које прихватају друге функције као параметре. Већина функција у стандардној библиотеци, поготово оних које раде са колекцијама података, представља функције вишег реда. Примјера ради *first()* када се позове над неком колекцијом враћа први елемент у колекцији, али могуће је прослиједити одговарајући ламбда израз у ком се спецификује правило по ком ће се вратити први елемент који задовољава дати услов. Слично је и са функцијама за сортирања, редукцију и мапирање података. Ове функције су имплементиране тако да представљају чисте функције, односно немају бочне ефекте, али проблем је у томе што ламбда функције које им се прослиједују могу имати бочне ефекте, па је пожељно користити непромјењиве објекте који су декларисани помоћу кључне ријечи *val*, како би компајлер зауставио покушај додавања бочних ефеката и тиме спријечио добијање неочекиваних резултата.

Како би се функције могле просиједивати као параметри другим функцијама у Котлину постоје функције као типови података. За дефинисања оваквог типа података потребно је навести параметре функције заокружене заградама, стрелицу (->) и повратни тип функције, нпр. (Int, Int) -> Boolean је тип података функције који прима двије цијелобројне вриједности и враћа логичку вриједност као резултат. На слици 3.9 је дат примјер како се дефинишу овакви типови података.

```
// (Int, Int) -> Int
// Lambda
val sum1: (Int, Int) -> Int = {a , b -> a + b}

// Anonimna funkcija
var sum2: (Int, Int) -> Int = fun(x: Int, y: Int): Int = x + y

// Referenca na funkciju
fun sum(a: Int, b:Int) = a + b // top-level funkcija

val sum3: (Int, Int) -> Int = ::sum

class SomeClass {
    fun sum(a: Int, b:Int) = a + b // member funkcija
}

val sum4: (Int, Int) -> Int = SomeClass()::sum
```

Слика 3.9 Креирање фунцкије као типа података

Функције као типови података често могу имати компликовану дефиницју, поготово када имају велики број параметара, или су ти параметри објекти неких класа које имају дугачак назив. Како би се поједноставило дефинисање оваквих функција као типова података у Котлину је могуће дефинисати *typealias*, као што се види са слике 3.10.

```
typealias ResMap = Map<NetworkBoundResource, NetworkBoundResource>
typealias ResPair = Pair<NetworkBoundResource, NetworkBoundResource>
typealias Pairmonger = (NetworkBoundResource, NetworkBoundResource) -> ResPair

fun collectify(input: ResMap, transform: Pairmonger): List< ResPair> {
    val result = mutableListOf<ResPair>()
    for ((key, value) in input) { result.add(transform(key, value)) }
    return result.toList()
}
```

Слика 3.10 Коришћење typealias

Поред стандардних колекција, које постоје и у Јави, Котлин посједује секвенце, које су по својој природи „лијене“. Лијеност се огледа у томе, да се прије позива функције над секвенцом не може знати да ли она уопште има икакве елементе. Када позивамо одређен број функција над колекцијом потребно је прије сљедећег ланчаног позива прво да се обраде сви елементи колекције, а када радимо са секвенцом сваки елемент се понаособ обрађује, што у одређеним ситуацијама може да резултује много бољим перформансама. На слици 3.11 је дата колекција која садржи објекте типа Weather, ако је потребно обрадити колекцију тако да се из ње извуче прво очитавање које је било негативно и парно, ми морамо обрадити све елементе из колекције, прецизније ток се одиграва на следећи начин:

- Креира се колекција која има 6 елемената,
- Позивом map() креира се нова колекција са 6 цијелобројних елемената,
- Позивом filter() креира се нова колекција која има 2 цијелобројан елемента,
- Позивом firstOrNull() враћа се цијели број -8.

```
data class Weather(val temperature: Int)

fun main() {
    val readings = listOf(Weather(1), Weather(-8), Weather(55), Weather(12345), Weather(128), Weather(9001))

    val evenNegativeReading = readings
        .map { it.temperature }
        .filter { it % 2 == 0}
        .firstOrNull { it < 0 }

    println(evenNegativeReading)
}
```

Слика 3.11 Обрада колекције

Овај ток није проблематичан када је ријеч о 6 елемената, али када се ради о 6000 елемената или неком великом броју N, тада ствари постају већ проблематичне. Позив истих функција над секвенцом даје следећи ток догађаја:

- Креира се колекција која има 6 елемената,
- map() емтиује прву цијелобројну вриједност 1,

- filter() прескаче ту вриједност јер није парна,
- map() емитује другу цијелобројну вриједност -8,
- filter() емитује -8 јер је парна вриједност,
- firstOrNull() враћа -8 јер је уједно и негативна вриједност

Овакав ток догађаја је меморијски ефикаснији, а уштеда је знатна када су у питању велике колекције односно секвенце и компликовани ланци функција који се над њима позивају. [9]

Котлин стандардна библиотека пружа солидну, подршку за писање програма преко парадигме функционалног програмирања, али ова парадигма се не намеће програмерима као обавезна, с друге стране одређени програмери који преферирају овај стил програмирања јер су у прошлости више радили са програмским језицима гдје је ово једини начин писања кода, као што су Clojure, Haskell и сл. поред стандардне библиотеке користе и неке друге библиотеке као што су Arrow.kt. [11]

3.2.4. Функције екстензије

Често се при програмирању јави питање гдје би требало неку функцију смјестити, а често је и одговор једноставан јер задатак који та функција треба да одрадти је везан за неку тачно одређену класу. С друге стране постоје функције за које ово не важи.

Ако је нпр. потребно написати функције које провјеравају исправност формата унесене мејл адресе или броја мобилног телефона, проблем не постоји док год се провјера врши само на једном мјесту, али шта ако је потребно провјеру вршити на више мјеста као нпр. приликом регистрације, пријаве, додавања нових контаката и сл., гдје би требало смјестити ове функције? Постоји више приступа:

- Надкласа, једно од рјешења код парадигме ООП јесте смјештање функција које се користе на више мјеста у надкласу, у случају Андроид програмирања то би могла да буде нека активност, фрагмент или презентер, али форсирање насљеђивања да би се омогућио приступ неким једноставним функцијама и избјегло њихово дуплирање није добар приступ са становишта ОО дизајна,
- Utility класе, друго типично рјешење код ООП јесте употреба utility класе, може бити ријеч о некој utility класи која је уже специјализована нпр. садржи функције само за валидацију података (DataValidator), или може то бити нека генеричка utility класа која садржи разне функције које обављају задатке који нису уопште сличне природе. Што је уопштенија utility класа то су веће шансе да програмери почну смјештати много кода с временом у њу, што резултује класама са великим бројем линија кода које је тешко одржавати,
- Глобалне функције, у језицима који нуде глобалне функције као Котлин, ово је чест приступ, гдје се у неким фајловима дефинишу глобалне функције које обављају дату функционалност. Овакав приступ није значајно унапређење у односу на писање utility функција, јер су сада те функције распршене по разним фајловима, па их је тешко одржавати. Додатан проблем је што све глобалне функције дијеле један простор имена, па су честе колизије, поготово када програмери крену користити туђе библиотеке,
- Функције екстензије, у неким програмским језицима као што је и сам Котлин могуће је проширити функционалност постојећих класа, нпр. за валидацију мејлова и

бројева телефона, могуће је проширити функционалност класе String. Функције екстензије се смјештају у фајл као и глобалне функције, при чему њихова видљивост може бити public, тада су доступне кроз цијели пројекат, може бити private, тада су оне доступне само у фајлу у ком су дефинисане, ово је корисно када је у питању неке функционалност уско везана за тај фајл, али и због избјегавања колизија, те могу се дефинисати унутар неке класе, тако да је приступ њима лимитиран на објекте дате класе.

Функције екстензије се дефинишу навођењем класе чију функционалност проширују, тачке (.) након чега слиједи дефиниција функција као и уобичајено, што се види са слике 3.9.

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

toast("Ovo je poruka")
```

Слика 3.12 Примјер функције екстензије

Функције екстензије немају приступ приватним и заштићеним члановима неке класе, такође не би требало давати исти назив функцији екстензије као што је назив неке функције дефинисане унутар класе, јер ће се при сваком позиву извршавати она функција дефинисана у самој класи. Компајлер ће упозорити програмера, ако покуша овако нешто до уради. На Андроид платформи функције екстензије се често користе како би се проширила, али и поједноставила функционалност класа из стандардне библиотеке, које се прилагођавају потребама развојног тима. Често се и користе за писање сопствених библиотека, које развојни тимови користе у свим својим пројектима, што им у многоме може убрзати писање кода, јер се не губи више вријеме на редудантне ствари. [9]

3.2.5. Инфикс функције

Поред стандардне синтаксе за позивање функција, Котлин подржава и синтаксу за инфикс функције. Примјер гдје се ово користи јесте при декларацији мапе гдје појединачни елементи представљају објекте класе Pair<K, V>, а добијају се позивом инфикс функције *to*. Слично *in* се користи за провјеру да ли неки елемент припада колекцији. Ова синтакса се често користи и у Котлин библиотекама за јединично тестирање као што је раније споменути Klover. Примјер инфикс функције дат је на слици 3.13.

Инфикс функције се могу само декларисати као функције екстензије или унутар неке класе као функције чланови и морају имати тачно један параметар

```
infix fun String.decorate(replacement: String) = "$replacement$this$replacement"

fun main() {
    println("Ovo je niska" decorate "*")
}
```

Слика 3.13 Инфикс функција

Инфикс функције могу да се уланчавају, резултат је синтакса слична енглеском језику, израз приказан на слици 3.13 је валидан у Котлину и користи kxdate библиотеку.

```
val twoMonthsLater = 2 months fromNow
val yesterday = 1 days ago
```

Слика 3.14 Практична примјена инфикс функција

3.3. ООП концепти

Котлин примарно користи ОО парадигму, те нуди већину функционалности које постоје у Јави, а уз то и одређена унапријеђења која чине писање кода у Котлину мање церемонијалним.

3.3.1. Обичне класе

Обичне класе у Котлину дефинишу се као у Јави помоћу клучне ријечи class након чега слиједи име класе и опционално тијело класе које се пише у витичастим заградама. Објекти класе се инстанцирају слично као у Јави, само без кључне ријечи new. Класе као у Јави могу да се налазе унутар одговарајућег пакета и синтакса за дефинисање пакета је иста као у Јави, само што је ; опционално на крају. За рефенрецирање неке класе из другог фајла користи се кључна ријеч import након чега се наводе пакети па назив саме класе. На Андроид платформи чест проблем настаје због колизије, како постоји мноштво класа са истим именом, а који су у различитим пакетима односно из различитх библиотека (нпр. Observable), у Јави би било рјешење да се одабере једна од референцираних класа и референцира преко пуног назива, односно package.to.class.Observable, а да се за ону другу користи само назив класе. Котлин ријешава овај проблем, преко алиаса за укључене референсе, односно (import _ as _) синтаксе, гдје сада другој Obseravble класи можемо дати алтернативни назив нпр:

```
import android.databinding.Observable
import io.reactivex.Observable as RxObservable
```

Чланови класе у Котлину могу бити функције, особине (енг. Properties), угњеждене класе и пратећи објекти (енг. Companion objects).

Функције се као чланови декларишу тако што дефиницију функције ставимо унутар тијела класе, исто као у Јави.

Особине су концепт који не постоји у Јави, а сличан је особинама у C# програмском језику. У Јави класе имају поља односно атрибуте, који ако се енкапсулишу користе гетере и сетер за приступ односно измјену вриједности, Котлин особине представљају комбинацију атрибута и његових гетера и сетера, те када радимо са особинама у позадини се у ствари позивају гетер/сетер. При чему ако дефинишемо неку особину са val, тада она има само гетер, јер се њена вриједност не може мијењати. Овим приступом се смањује све општа церемонијалност, при чему ако је потребно да се дефинишу специвични гетери и сетери, могуће је и то урадити, што се види са слике. (ставити слику као примјер). Алтернатива писању посебних гетера и сетера јесте кориштење делегата, обрађених под тачком (тачка).

Као и у Јави, у Котлину постоји подразумијевани конструктор који се користи, када не постоји експлицитни. Конструктор се дефинише у заградама које слиједе након назива класе и није потребно навести кључну ријеч constructor јер се она подразумијева на овом мјесту, сви остали конструктори који се дефинишу унутар тијела класе морају имати ту кључну ријеч. У Котлин класама се ријетко јавља више конструктора јер се користе подразумијеване вриједности параметара, па још уз коришћење именованих параметара добијамо преко једног конструктора у Котлину исту функционалност као од више њих у Јави, што је посебно корисно на Андроид платформи, нпр. када је потребно имплементирати посебни View, гдје умјесто четири конструктора је довољно навести један са подразумијеваним вриједностима за параметре. Параметри конструктура ако су наведени одма након класе, а имају кључну ријеч var/val уједно представљају и особине, али без тога ријеч је о обичним параметрима који се нпр. могу користити у иницијализационим блоковима, који се дефинишу преко кључне ријечи init и тијела које се налази унутар витичастих заград, како би се додијелила вриједност некој особини унутар класе. При чему се мора водити рачуна да је одговарајући иницијализони блок постављен након дефиниције особине која се у њему наводи, у пракси се овај (ови) блок смијешта након што су дефинисане све особине. [9]

3.3.2. Видљивост и опсег

Видљивост се односи на то ко има право да види класу, функције и особине, а опсег се односи на то гдје те класе, функције и објекти постоје. У Котлину модификатори за видљивост су:

- Подразумијевана видљивост односно public, у Котлину за разлику од Јаве све што се дефинише има видљивост public, дакле доступно је свуда,
- Приватна видљивост (private), означава да су дати чланови доступни само унутар класе односно фајла у ком су дефинисани, ако је ријеч о глобалним функцијама,
- Заштићена видљивост (protected), исто као приватна, разлика је само у томе што су дати чланови доступни и у свим изеденим класама дате класе, ова видљивост се може навести само за чланове неке класе, јер нигдје друго нема смисла,
- Интерна видљивост (internal), све што је маркирано као internal је само видљиво унутар модула у ком се налази, ова видљивост се у пракси ријетко користи.

Опсег индиректно контролише видљливост јер се не може референцирати нешто што не постоји у датом опсегу, али има и значајан утицај на GC, јер од опсега зависи животни вијек овјеката и када се они склањају са heap-а. Разликујемо сљедеће нивое видљивости:

- Глобална, то су глобалне варијебле и фунцкије, ако је ријеч о непромијењивим глобалним варијаблама оне постоје док се извршавају процеси у којима они учествују, а ако је ријеч о промјенљивим гловалним варијаблама, тада саме референце живе до краја, односно док год се апликација извршава, али објекти који се рефенцирају могу бити склоњени са heap-а и биће склоњени ако се на њих више не показује,
- Инстанца, особине дефинисане унутар класе имају видљивост инстанце, сваки објекат неке класе имаће своје инстанце особина, неовисне од других објеката,

- Локалн, свако var/val унутар што није класа има локалну видљивост, нпр. варијабла може бити локална за неку функцију, или за ламбду па када се њихово извршавање заврши она нестаје са сцене.

3.3.3. Апстрактне класе и интерфејси

У јави се често користе апстрактне класе, али и интерфејси који омогућавају прављене "уговора" који класе морају да испоштују, нпр. уговори се користе у MVP архитектуралном стилу, али и за многе друге обрасце. Апстрактне класе у Котлину функционишу на исти начин као у Јави и ове класе су аутоматски open, дакле могу се насљеђивати, једна примјена апстрактних класа јесте за моделовање појава које су по својој природи апстрактне, могуће је нпр. дефинисати класу за мачке и псе те је познато шта то карактерише ове животиње, али није познато шта карактерише једну животињу јер је ово сувише генерички појам, па не би требало програмерима омогућити да креирају објекте типа животиња, за ту сврху се користе апстрактне класе, а нпр. не интерфејси јер иако није могуће направити инстанцу интерфејса, они немају чланове па нису добри за моделовање хијерархије објеката из природе. Они се пак користе за склапање уговора или дефинисање неке функционалности која може да постоји између објеката који имају скроз различиту природу, или једноставно за сврхе маркирања. Интерфејси у Котлину су готово идентични по функционалности онима из Јаве, при чему је једина разлика у синтакси. У котлину се експлицитно мора нагласити да је нека функција у интерфејсу апстрактна, у супротном је потребно функцију дефинисати. Јава овај концепт гдје функције односно методе унутар интерфејса могу имати конкретну имплементацију подржава тек од верзије 8. Како се Котлин може компајлирати и у Јава 6 бајткод, а посоји 100% интероперабилност имзеђу два језика, могуће је ову функционалност користити у пројектима који су идаље везани за верзију 6, директно из Котлина.

3.3.4. Класе података

Једна од главних примједби Андроид апликационих програмера на рачун Јаве јесте велика церемонијалност, гдје се чак за дефинисање једноставног JavaBean-а мора написати много линија кода. Дизајнери Котлина су се потрудили да се ријеше овог проблема, а један од начина јесте и преко класа података (енг. Data classes), које нуде исту функционалност као многе библиотеке написане за Јаву, али уз једноставнију синтаксу јер су дио Котлин језика. Наводе се слично као обичне класе, само испред кључне ријечи class се дода кључна ријеч data, те опет није потребно наводити тијело. Предност класа података у односу на обичне класе је што Котлин самостално дода имплементације за стандардне Јава методе:

- equals(), пореде се вриједности свих особина класе,
- hashCode(), генерише се цијели број који се може користити за HashSet или HashMap,
- toString(), за сврхе дебаговања, враћа вриједности свих особина класе,

Поред ових функција Котлин генерише и имплементацију за copy() функцију, која се користи за креирање копије објекта неке класе података, при чему је могуће измјенити вриједности индивидуалних особина коришћењем именованих параметара, као што се види са слике.

Оно што се губи употребом класа података је немогућност да се означе као open што значи да је немогуће наслиједити овакву класу, бочни ефекат је то да методе унутар класе

података не могу бити апстрактне. Такође иако је могуће навести особине ван конструктора, оне се неће користити при генерисању equals() и copy(). [9]

На Андроид платформи, класе података се уобичајено користе за моделовање ентита базе података, или за моделовање одговора добијених преко неког REST API-ја.

3.3.5. Насљеђивање

Као и већина других ОО програмских језика, Котлин подржава насљеђивање, дијете класа „насљеђује“ особине, конструкторе, функције и др. родитељске класе које имају одговарајућу видљивост (нису приватне). У Јави свака класа насљеђује директно или индиректно класу Object, а у Котлину је то класа Any. Да би се могла нека класа извести из неке друге потребно је навести : након чега слиједи име родитељске класе и позив конструктора коме се морају прослиједити одговарајући параметри, могуће је имати само једну родитељску класу као у Јави. Додатно да би се омогућило насљеђивање потребно је класу маркирати са кључном ријечи open, јер су подразумијевано све класе у Котлину затворене за извођење, што је у контрасту са Јавом, гдје се експлицитно мора забранити извођење. Ова одлука је донешена како би програмери боље размислили прије него што се користе насљеђивањем, јер често се деси да у свом коду нарушавају Лисков принцип супституције којим се каже да сваки објекат неке изведене класе може да се користи умјесто објекта родитељске класе, ово се често крши при извођењу класа из неке библиотеке. Додатно да би се омогућило преклапање неке функције у изведеној класи, потребно је ту функцију маркирати као отворену те је потребно навести кључну ријеч override у изведеној класи гдје се та функција преклапа. Када се нека фунцкија маркира као отворена, она је отворена кроз цијелу хијерархију и то се не може промјенити. Могуће је и особине преклопити, ово је понекад потребно за оне особине које се наводе као параметри у главном конструктору, јер компајлер не дозвољава да се особина изведене класе користи за маркирање истоимене особине из родитељске класе.

Да би се провјерило да ли нека референса показује на објекат изведене класе у Јави се користи кључна ријеч instanceof, а у Котлину је то as. Једна од предности Котлина у односу на Јаву када је у питању полиморфизам јесу тзв. паметни кастови (енг. Smart casts), чија је примјена приказана на слици (слика). У Јави и након што је са сигурношћу утвђено да animal показује на објекат типа Dog, потребно је експлицитно кастовати дати објекат у инстанцу класе Dog, иако би компајлер требао да зна да је ријеч о тој изведеној класи у датој грани, Котлин ово ријешава поменутим паметним кастовима гдје компајлер када је сигуран да је ријеч о баш том изведеном типу аутоматски кастује објекат у одговарајући тип тако да се над њим могу позивати функције које постоје само за Dog објекте. Наравно, у else гранама не може се са сигурношћу знати ког типа је неки објекат па се не врши аутоматско кастовање, а то би и требао бити знак програмерима да ако компајлер није у стању да аутоматски закључи тип ни они не би требали да са сигурношћу тврде да је објекат одређеног типа.

3.3.6. Класе енумерације и затворене класе

Класе енумерације се користе за моделовање неког броја стања и раде у Котлину исто као што раде у Јави, разлика је само у томе што се дефинишу тако што се испред класе дода кључна ријеч enum. Класе енумерације имају конструкторе и функције, као и све остале класе. За свако стање које се моделује, односно за сваку енумерациони константу се

генришу двије особине, а то су name и ordinal. Када се користи when гранање у Котлину потребно је обрадити сваки могући сценарио, зато се мора навести else грана, један од изузетака гдје ово није потребно јесте код класа енумерација, јер ако се корз гранања наведу сва стања не могу постојати нека друга поред тих, али ово само важи ако је наведено свако могуће стање. Алтернатива класама енумерације у Котлину јесу затворене класе, које могу бити родитељске класе само ако се изведена класа налази у истом фајлу или унутар саме затворене класе као угњеждена класа. Главна разлика затворене класе у односу на неку класу енумерације је што свако стање у енумерацији представља singleton, дакле у сваком тренутку постоји тачно једна инстанца, са друге стране код затворених класа може постојати N објеката изведене класе у било ком тренутку, а како се све изведене класе морају налазити унутар истог фајла when израз не мора ни у овом случају да има else грану и поред овог паметни кастови такође важе. Затворене класе се наводе тако што се дода кључна ријеч sealed испред дефиниције класе, типично се у тијело смијештају све изведене класе које су углавном класе података или singleton тј. објекти.

Енумерације се користе на истим мјестима гдје би се оне користиле у Јави, а затворене класе се користе за моделовање стања података, нпр. на Андроид платформи када се дохваћају подаци преко мреже, резултат извршавања може бити тај да се добију назад валидни подаци, или да дође до неке врсте грешке при дохваћању података. Како су у MVI архитектуралном обрасцу стања кључан дио архитектуре, тада се за моделовање ове архитектуре користе затворене класе.

4. АНДРОИД РАЗВОЈНО ОКРУЖЕЊЕ

За развој модерних Андроид апликација користи се Android Studio интегрисано развојно окружење, које је базирано на JetBrains IntelliJ IDEA интегрисаном развојном окружењу. Android Studio је замјенио дотадашње Eclipse Android Developer Tools (ADT) окружење у децембру 2014. године када је изашла верзија 1.0. Пружа подршку за писање Андроид апликација у Јава и C++ програмским језицима, а од верзије 3.0 и за писање Андроид апликација у Котлину. Подршка за Котлин је изванредна јер компанија која стоји иза Котлина уједно стоји и иза развојног алата на ком је базиран Android Studio. Подршка за Јаву није на истом нивоу као и у стандардном IntelliJ IDEA окружењу, јер су тренутно подржани Јава 7 и дио унапријеђења из Јава 8, док IntelliJ IDEA подржава и тренутно најновију берзију Јаве, а то је Јава 13. Постоје и алтернативна окружења за писање Андроид апликација, а то су нпр. Visual Studio који има подршку за писање Xamarin C# апликација и Visual Studio Code који се често корсити за писање хибридних и веб апликација, али како ово нису примарна ријешења за писање нативних апликација у Котлину неће се даље разматрати.

Android Studio нуди мноштво функционалност, а неке од најважнијих су:

- Подршка за рефакторинг Андроид специфичног кода,
- Андроид Емулатор (енг. Android Virtual Device – AVD),
- Подршка за писање Андроид апликација за све врсте уређаја: мобилни уређаји, таблет уређаји, паметни сатови, паметни ТВ итд.,
- Подршка за дизајнирање и визуелну инспекцију Андроид апликација,
- Интеграција са ProGuard,
- Уграђена подршка за Google Cloud Platform и за Firebase,
- Подршка за екстензије писане за IntelliJ IDEA,
- Доступност на свим великим платформама: Windows, OSX, Linux.

4.1. Креирање и импортовање Андроид апликације

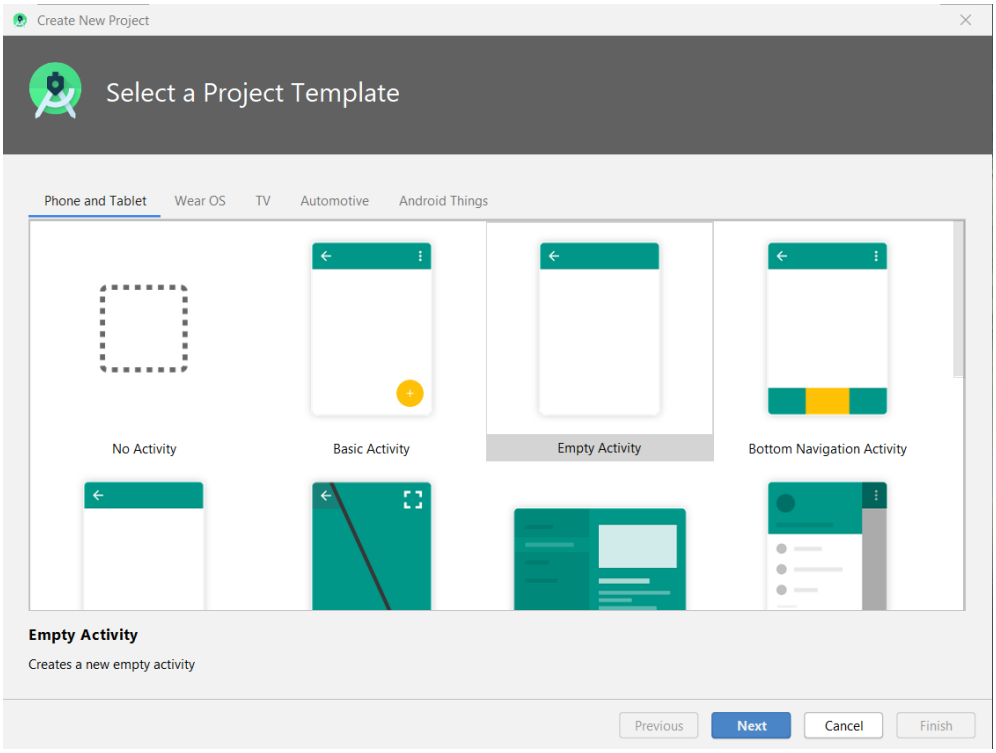
Прије креирања новог Андроид пројекта потребно је одредити:

- Јединствени идентификатор апликације, на једном уређају не могу се истовремено инсталирати двије апликације које имају исти идентификатор, нити се могу дистрибуирати двије апликације преко Google Play Store са истим идентификатором, а исте рестрикције важе и за друге канале дистрибуције. Са становишта апликације идентификатор мора да буде валидан назив за Јава/Котлин пакет. Препорука је купити неки домен, те као први дио назива пакета користити реверзни домен, а као други дио сегмент који идентификује баш ту апликацију, како се не би дошло у колизију са другим апликацијама унутар исте фирме које дијеле домен. Ова одлука је веома битна, јер након избора идентификатора тешко га је промјенити јер нови идентификатор се сматра скроз другачијом апликацијом,
- Директоријум у ком ће се смјестити пројекат, пожељно је да то буде лако доступно мијесто на диску корисника,
- Програмски језик апликације, могуће је одабрати Јава базиран пројекат или пројекат који поред Јаве подржава и Котлин. Тренутно није могуће одабрати опцију за

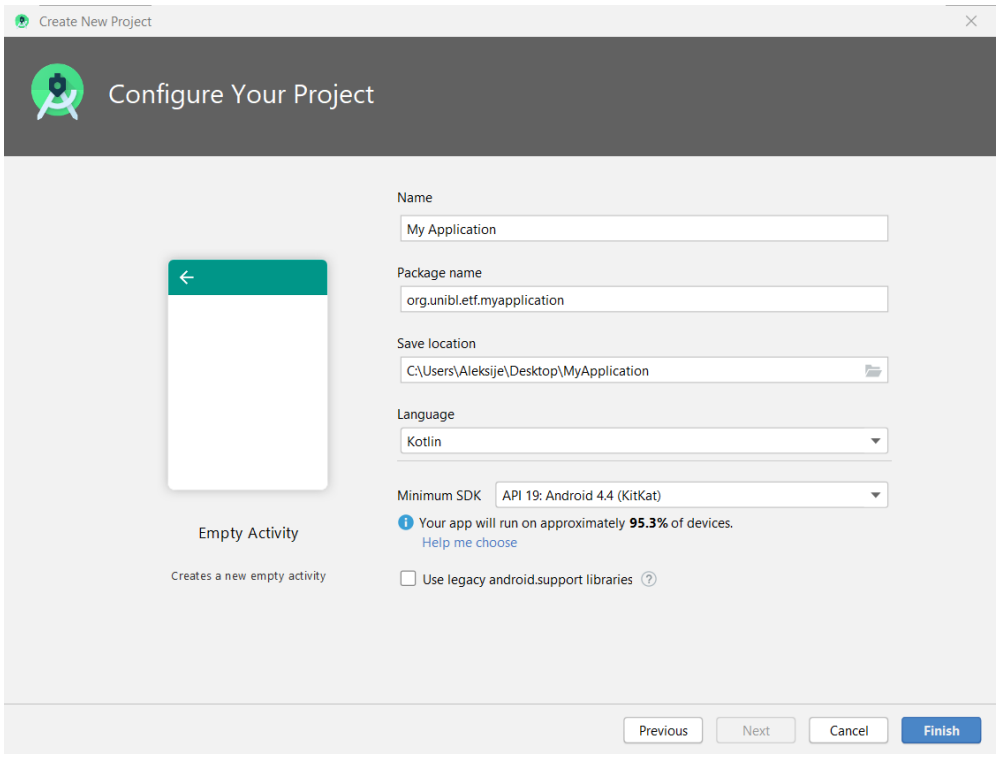
пројекат који је само базиран на Котлину јер велики број API-ја још увијек има само Јава имплементацију, али ово не представља велики проблем како се котлин извршава на Јава ВМ,

- Избор најниже подржане верзије Андроид ОС, што је новија минимална верзија то је једноставније тестирање апликације, а и потребно је писати мање кода уопште јер се не мора размишљати о подршци уназад толико, с друге стране ако је потребна подршка за што већи број корисника онда је пожељно одабрати старију верзију као минималну, избор зависи од потреба програмера или клијента, али у пракси се неће одабрати ниво API испод 14.

За креирање нове Андроид апликације, Android Studio нуди project wizard до ког се може доћи преко менија кликом на File > New > New Project… или преко почетног дијалога, ако тренутно није ни један пројекат отворен, кликом на "Start a new Android Studio project" дугме, након чега се отвара project wizard. Прво је потребно одабрати темплејт, са списка доступних, који су приказани на слици. Идеја је да се одабере темплејт који представља најбољу почетну тачку у зависности од каквог типа апликације односно пројекта програмер жели да направи. Тако да су у оптицају темплејти који нуде имплементирану одређену врсту навигације или подршку за неки специфичан тип уређаја. У пракси се пак већина програмера одлучи за празан темплејт, тј. активност (енг. Empty Activity) јер је за прилагођавање потребно најмање времена, а остали темплејти се углавном користе за сврхе експериментисања и истраживања, више него почетне тачке за праве пројекте. Након избора одговарајућег темплејта, слиједи подешавање основних карактеристика пројекта, овај дијалог приказан је на слици 4.2.



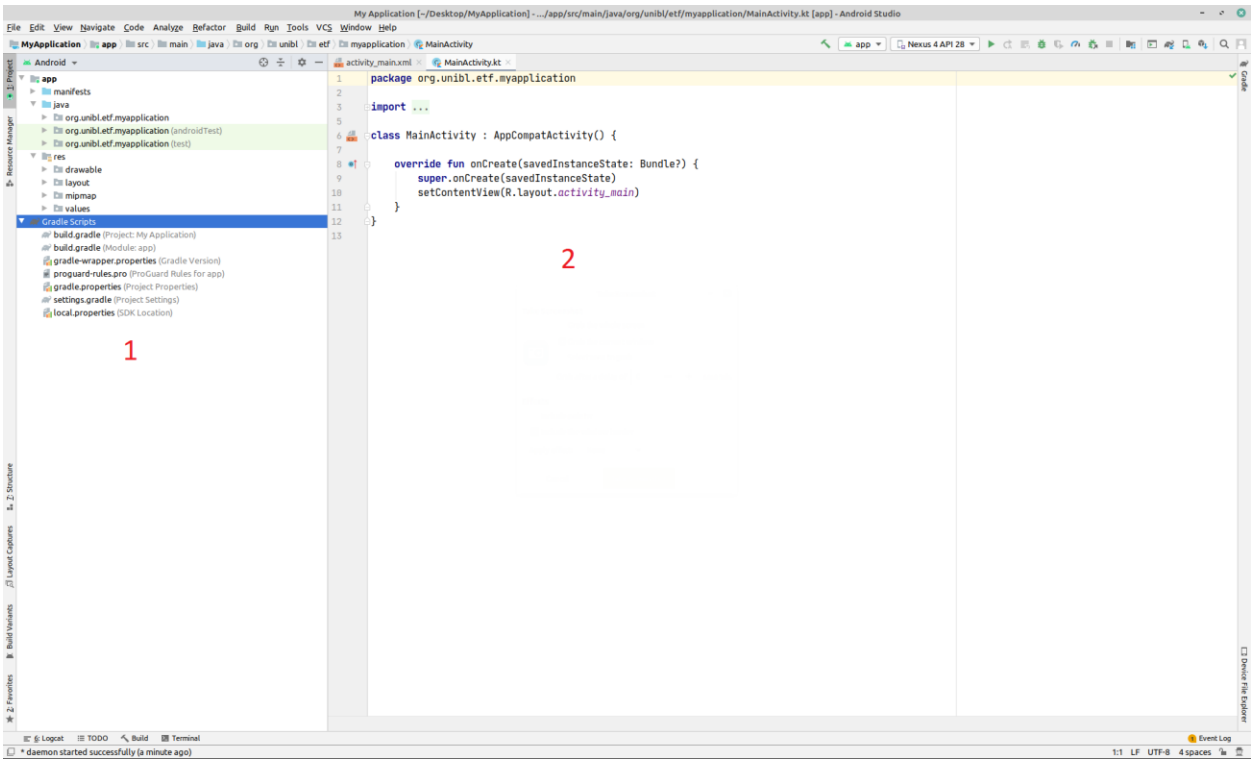
Слика 4.1 Избор темплејта за апликацију



Слика 4.2 Подешавање детаља пројекта

У поље name уноси се назив апликације, назив се смијешта у app_name string ресурс и користи се у манифесту као лабела за назив апликације и главне активности у апликацији. У поље package name уноси се идентификатор апликације, ако је програмер већ раније правио неке пројекте, тада ће окружење аутоматски попунити ово поље. Поље save location представља локацију на диску гдје ће се смјестити пројекат, при чему се не може навести фолдер у коме се већ нешто налази. Након чега је из падајућег менија потребно одабрати програмски језик у ком ће апликација бити написана, први избор је Котлин, а алтернативно могуће је одабрати Јаву. Након тога слиједи падајући мени у ком је потребно навести минималну Андроид верзију која је потребна да би корисник могао инсталирати дату апликацију. Након уношења вриједности потребно је само кликнути дугме Finish и основни пројекат ће бити изгенерисан.

По генерисању добија се почетно окружење као на слици 4.3. Дио означен са један представља фајл (пројекат) менаџер и у њему су приказани сви фајлови релевантни за пројекат, у горњем лијевом углу могуће је прилагодити начин приказивања фајлова, при чему су најчешћи прикази Android и Project. У Android приказу посебно су одвојене gradle скрипте од фајлова који чине саму апликацију (изворни код и ресурси). Сегмент означен са два представља радну површину, односно дио окружења у ком се пише код. Поред ова два сегмента окружење нуди још мноштво опција и пар сегмената, који се неће разматрати.



Слика 4.3 Почетно стање окружења

4.2. Емулатор, дебаговање и тестирање апликације

За покретање написане апликације програмери имају двије могућности, прва је да користе емулатор и друга је да користе физички уређај. Оба приступа имају своје предности и мане, емулатори су знатно спорији и не могу емулирати баш све функције уређаја, тако је нпр. немогуће тестирати апликације које броје кораке, али имају ту предност да се на њима може тестирати понашање апликације на различитим уређајима, који имају различите величине екрана, различиту верзију Андроид ОС, те различите хардверске могућности, а физички уређаји омогућавају тестирање свих функционалности апликације те дају много реалнију представу о томе како би се апликација извршавала у продукцији. С друге стране употреба емулатора је нужна због економичности, наиме сувише би много коштало да се сваком програмеру који ради на неком пројекту обезбједи више уређаја за тестирање.

За креирање емулатора користи се AVD (Android Virtual Device) Manager, који нуди опцију за креирање емулатора на основу постојећих профила, базираних на Гугл уређајима, али и могућност измјене постојећег или креирање скроз новог профила/уређаја. При креирању профила потребно је одабрати уређај, након чега се врши избор верзије ОС која ће се користити на емулатору, те се коначно подешавају неки детаљи специфични за тај емулатор, као што су назив, оријентација и слично. Како је брзина емулације директно везана и за резолуцију уређаја, програмери често користе уређаје са нижом резолуцијом за нека иницијална тестирања, уређаје попут Nexus 4.

Како би се користио мобилни уређај за покретање апликације, потребно је у подешавањима омогућити развојне опције, односно дебаговање преко УСБ уређаја, често је за активацију потребно додирнути ознаку верзије система седам пута, након чега се корисник сматра за апликативног програмера те ће у подешавањима моћи пронаћи мени са

развојим опцијама, гдје може укључити опцију „откалњање грешака путем УСБ уређаја“ чиме је уређај спреман за кориштење за рад покретања апликације. У зависности од ОС могуће је да постоје додатни кораци како би се омогућила употреба уређаја. Ако је све подешено како треба, тада се приказује прикључени уређај у падајућем менију уређаја на којима се може покренути апликација, као на слици 4.4.

Слика 4.4 Мени за покретање апликације

За покретање апликације потребно је кликнути зелену стрелицу, прво дугме до падајућег менија за избор уређаја након чега се извршавају gradle build скрипте и апликација се инсталира на уређају. Извршавање gradle build скрипти код озбиљнијих апликација може потрајати дужи временски период, па да се не би након сваке мале промјене морало изнова покретати апликација, могуће је користити друге двије опције. Друга опција до менија се користи када је дошло до измјена у коду и у ресурсима које апликација користи, окружење ће покушати да дода измјене и након чега ће рестартовати одговарајућу активност, за случај да су промјене само у коду, може се користити трећа опција, гдје окружење покушава да дода измјене у коду, без да се рестартује активност.

За дебаговање апликације, потребно је назначити одговарајуће breakpoints-е у коду, уграђени дебагер омогућава инскепкцију вриједности варијабли те пролазак кроз инструкције једну по једну. Да би се апликација дебаговала потребно је кликнути на дугме са зеленом бубом, а ако је апликација већ покренута потребно је кликнути на дугме са зеленом бубом и стрелицом. За провјеру перформанси апликације и оптерећења меморије користе се опције за profiling.

Тестирање апликације се врши писањем одговарајућих тестова. Уобичајено се користи Junit библиотека, али постоје и неке библиотеке специфичне за Котлин које чине тај процес једноставнијим, као што је Kluent. Окружење има подршку за покретање тестова, генерисање извјештаја на основу успијешности извршавања тестова те тачно маркира оне тестове који се или нису иврши или услијед грешке или нису прошли. Иако окружење нуди богату подршку за тестирање, ипак сам тај процес је спор, поготово када је потребно провјерити апликацију прије него што се она пусти у продукцију. За те сврхе могуће је користити фарме за тестирање гдје се апликација извршава односно тестови извршавају на мноштву уређаја, подршка за овакав вид тестирања је уграђена преко Firebase.

5. АРХИТЕКТУРАЛНИ СТИЛОВИ ЗА АНДРОИД АПЛИКАЦИЈЕ

Развој Андроид апликације без праћења одређених смјерница се брзо покаже као контрапродуктиван приступ. Наиме иако је иницијални развој текао брже, свака измјена постојећих фунцкионалности или проширивање апликације са новим функционалностима је мукотрпна. Праћењем архитектуралних смјерница постижемо раздвајање на компоненте према њиховој улози, па је нпр. могуће измјенити изглед корисничког интерфејса без измјене логике која је у позадини. Такође омогућавамо писање тестова којима се развојни програмери осигуравају да измјене нису увеле нове грешке.

Са временом опробане су многе архитектуралне смјернице за писање Андроид апликације: MVC, MVP, MVVM, MVI и друге. Неки од њих су се показали више, а неки мање успјешним. Гугл је дуго времена избјегавао да се изјасни о томе које су препоручене смјернице, али са појавом *Android Architecture Components*, гдје постоји *ViewModel* класа јасно је да се преферирају стилови MVVM и MVI.

5.1. MVC

Модел-Приказ-Управљач (енг. Model-View-Controller – MVC) је архитектура која се заснива на раздвајању компоненти према улози. Разликујемо три врсте компоненти:

- Модел, обухвата сву пословну логику која се односи на рад са подацима, као што је чување, дохваћање и обрада података из базе података или са мреже,
- Приказ, компонента која се бави приказом података за корисничи интерфејс,
- Управљач, представља посредника између модела и приказа, ослушкује и прихвата корисничке захтјеве на основу којих манипулише са моделом те кориснику, у случају да је дошло до промјене на моделу приказује резултат његове интеракције тако што га враћа до приказа.

Модел нема представу о томе на који начин се подаци приказују кориснику, а с друге стране приказ се не бави са вриједностима података који се приказују, већ искључиво приказом. Управљач је овде у улози оркестратора података. MVC архитектура је данас популарна за развој веб апликације, али се не користи за писање Андроид апликација.

За правилно тестирање Андроид апликације потребно писати све три врсте тестова, мале, средње и велике. Мали тестови су јединични тестови, којима се тестирају појединачне класе унутар апликације, средњи тестови су интеграциони тестови којима се тестира интеракција између класа унутар истог модула или између повезаних модула. Велики тестови су тестови корисничког интерфејса, којима се тестира интеракција са апликацијом. Гугл предлаже писање тестова у складу са Андроид тест пирамидом приказаној на слици 4.1, од којих су 70% тестова мали, 20% тестова су средњи и 10% тестова су велики. Предност малих тестова је што није потребна Андроид виртуелна машина или мобилни уређај како би се извело тестирање, па се овакви тестови извршавају много брже од остала два типа. Такође јединични тестови не захтјевају никакве Андроид специфичне библиотеке већ се користе стандардне библиотеке као што су JUnit, Mockito и Kluent.

Међутим овај архитектурални стил не може да се адекватно примјени на Андроид апликације, јер није могуће правилно извршити расподјелу на компоненте према њиховој

улози. Наиме *Activity* који би требао да представља управљач, уједно мора да обавља задатак измјене приказа података на *.xml layout*-у што значи да су му потребне референце на елементе који су приказани на овом погледу, чиме он ефективно има обе улоге. Ово представља додатни проблем јер сада није могуће писати стандардне јединичне тестове за управљаче јер они имају референце на код који је Андроид специфичан, па је потребна Андроид виртуелна машина, а тиме је добит од писања јединичних тестова изгубљена. [12]

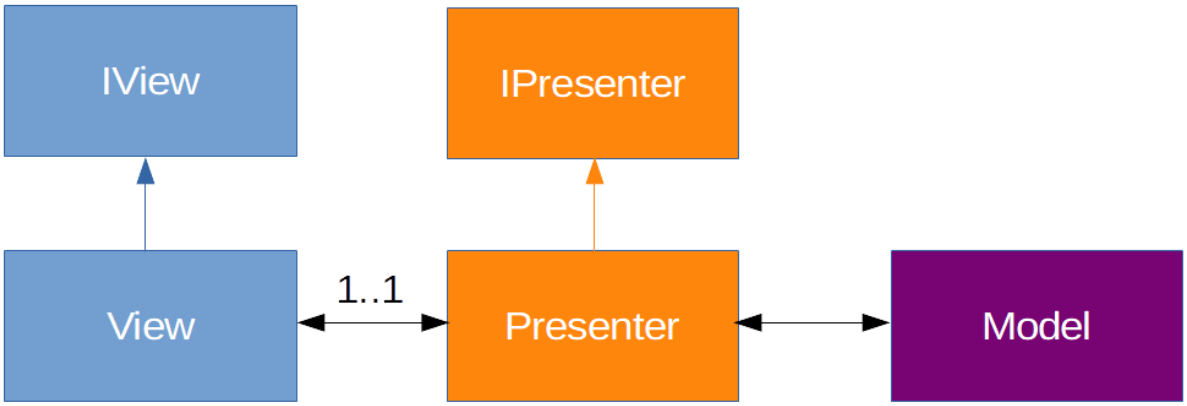
Слика 5.1 Пирамида Андроид тестова

5.2. MVP

Модел-Приказ-Презентер (енг. Model-View-Presenter – MVP) је архитектурална смјерница гдје постоји подјела на три компоненте са уско дефинисаним улогама, како би се избјегли проблеми које сусрећемо у MVC. Разликујемо:

- Модел, задужен је за пословну логику апликације,
- Приказ, бави се приказивањем корисничког интерфејса и обавјештавањем презентера о корисничким акцијама,
- Презентер, дохвата податке од модела, управља логиком корисничког интерфејса и приказом, одлучује шта да се прикаже кориснику и како да се одреагује на корисничке акције које су прослијеђене од приказа.

Интеракција се врши преко приказа који ослушкује корисничке акције, при чему се за приказ користе Activity или Fragment, нпр. акција може да буде клик на дугме или унос текста. Након корисничке акције, приказ обавијести презентера о акцији, а презентер у складу са акцијом може да ажурира модел. Ако је дошло до ажурирања модела, тада презентер треба да ажурира приказ са свјежим подацима. Овим приступом су успијешно распоређене појединачне улоге, али остаје проблем јединичног тестирања презентера. Како би презентер могао да обрађује захтјеве које му упућује приказ, потребно је да има референцу на њега, односно на Андроид специвичне библиотеке. Да би се ово избијегло можемо се користити тзв. слабим референцама, гдје конкретни приказ и презентер имплементирају одговарајуће интерфејсе, а њихова веза је моделована преко уговара односно класе *Contract*. Ово је генерално добар приступ са становишта софтверског инжењерства јер ове класе сада нису уско везане, тј. отклоњене су референце на Андроид библиотеке, па самим тим и презентер се може тестирати помоћу јединичних тестова. Приказ MVP структуре дат је на слици 5.2.



Слика 5.2 MVP архитектурални стил

Један од проблема који сусрећемо код овог архитектуралног стила јесте да је често ниво комплексности који се захтјева од развојног програмера сувише велики за једноставне пројекте. У таквим ситуацијама могуће је не користити посебну класу за уговор, али и не имплементирати интерфејсе за презентер, чиме добијамо један крњи MVP приступ, али идаље сасвим добар за једноставније апликације. Други чест проблем јесте тенденција да презентер постане сувише комплексан јер већина кода заврши у њему, да би се овај проблем избјегао потребно је добро раздвојити класе и водити се принципом да једна класа рјешава један проблем. [13]

5.3. MVVM

Модел-Приказ-МоделПриказа (енгл. Model-View-ViewModel – MVVM) је архитектурални образац код кога опет постоји подјела на три компоненте, са уско дефинисаним улогама, један је од препоручених образаца са Андроид апликације. Разликујемо сљедеће три компоненте:

- Приказ, бави се приказом корисничког интерфејса и обавијештава друге компоненте о корисничким акцијама,
- МоделПриказа, опслужује податке за приказ,
- Модел, пословна логика апликације

На први поглед не постоји јасна разлика између MVVM и MVP архитектуралних смјерница, јер VM обавља сличну функцију као и презентер. Разлика је у томе што је експлицитно наглашено да VM не смије имати референцу(е) на приказе, односно ако је правилно имплементирана класа која служи као VM неће имати референце на Андроид специфичне библиотеке осим оних из пакета *com.android.arch.**. Дакле VM се бави само опслуживањем података, а не одлучује о томе на који начин ће се подаци приказивати корисницима. Модел, познатији као модел података (енг. Data Model) има улогу да доставља адекватне податке за VM, такође је задужен за извршавање CRUD операција над подацима који су негдје у позадини, било да је ријеч о подацима из базе података или подацима који се дохватају са интернета, при чему VM не треба да брине о томе на који начин је дошао до података. Уобичајено се у Андроид апликацијама за моделе користе Котлин класе података, при чему се треба испоштовати принцип да једна класа испуњава једну улогу. Коначно задатак приказа јесте да презентује податке кориснику, он може имати референцу на један или више VM, те прати податке које он опслужује кориштењем Observer софтверског обрасца.

Главна предност MVVM у односу на друге архитектуралне смјернице јесте у томе што је директно подржан од стране Гугла кроз њихове библиотеке, па је самим тим најлакши за имплементирати на правилан начин, а уз то је и најбоље документован и постоји велики број примјера како званичних тако и оних које је дала Андроид заједница. У поређењу са MVP, потребно је писати мање кода. [14] На слици 5.3 је приказана MVVM шема.

Слика 5.3 MVVM шема

5.4. MVI

Модел-приказ-намјера (енг. Model-View-Intent – MVI) је архитектурална смјерница која се често користи у Јаваскрипт програмском језику и на вебу. Настала је у склопу *cycle.js* библиотеке, а користи је још у популарна *Redux* библиотека. Разликујемо:

- Модел, је контејнер за стања, стање учитавања података, стање грешке, стање приказа података итд.,
- Намјера, представља жељу корисника да се изврши одређена акција, за сваку корисничку акцију, приказ добија нову намјеру, које се прате помоћу презентера и као резултат добијамо ново стање у моделу,
- Приказ, бави се приказом корисничког интерфејса и прослијеђивањем корисничких акција.

MVI користи реактивно програмирање, гдје апликације на акције реагују преласком у ново стање. Ново стање се често репрезентује као промјена у корисничком интерфејсу, примјера ради ако је корисник притиснуо дугме да се подаци учитају са интернета, тада апликација прелази у стање учитавања и на корисничком интерфејсу се приказује *progress bar*. Битно је да модели буду непромјењиви, како би се остварио ток података у једном смјеру и уједно затвореност тока података. Једна од предности оваквог приступа јесте да се апликација увијек налази у тачно одређеном стању те да како не постоје бочни ефекти могуће је много лакше имплементирати вишенитност.

Са друге стране иако је ријеч о напредном приступу који има предности у односу на раније наведене, његова мана лежи у чињеници да је релативно нов, па постоји мање материјала и документације потребних за правилно разумијевање, а и по својој природи је сложенији, јер захтјева познавање парадигме реактивног програмирања, те не представља добар избор за једноставне апликације. [15]

6. ПРОЦЕС РАЗВОЈА АНДРОИД АПЛИКАЦИЈА

Андроид апликација састоји се од компонената које се извршавају на Андроид ОС-у, међусобно комуницирају и дијеле ресурсе. Оваква архитектура омогућава апликацијама да користе компоненте других апликација, ако те апликације то дозвољавају нпр. приликом дијељења садржаја огласиће се свака апликација која подржава дати формат, као и могућност вишеструке употребе неке од компоненеата. Овим се редукује употреба меморије и других ресурса уређаја те се ефикасније извршавају апликације. Основне компоненте Андроид апликација су: Activity, Service, Broadcast Receiver и Content Provider. Једна апликација не мора имати све наведене компоненте, већ само оне које су потребне.

Активност (енг. Activity) је дио корисничког интерфејса и често полазна тачка када је ријеч о једноставнијим апликацијама. Уобичајено активност се простире преко цијелог екрана, остављајући простора за статусну линију (енг. Status bar) и навигациони мени, али неки уређаји могу приказивати више активности истовремено, нпр. помоћу split-screen режима или помоћу multi-window режима. Активност представља основни градивни блок корисничког интерфејса, с тим да се данас тежи ка апликацијама које имају једну активност, те мноштво фрагмената, па би активност представљала само контејнер за фрагменте, овако нешто у пракси још није могуће. Све активности наслијеђују класу AppCompatActivity, а ова класа наслијеђује класу Activity. Технички је могуће само директно наслиједити Activity, али Гугл се противи томе јер је AppCompatActivity специфично развијен да се добро понаша (конзистентно) на различитим верзијама Андроид ОС.

Сервис је компонента која се извршава у позадини апликације и није повезана са корисничким интерфејсом. Сервис може да се извршава и када апликација није у фокусу на екрану уређаја, нпр. за репродукцију музике или за ослушкивање захтјева са мејл сервера. Дијеле се на локалне и удаљене сервисе. Локални сервиси се извршавају у истом процесу као и апликација, а удаљени сервиси се извршавају у одвојеном процесу и користе се за комуникацију између процеса.

Broadcast Receiver је компонента која представља страну претплатника у огласивач/претплатник комуникационом систему, који користе како апликације тако и ОС за комуникацију.

Content provider је компонента која омогућава апликацији да сачува податке на уређај и да их дохвати из одговарајућег складишта. Такође, представља механизам помоћу којег апликације међусобно комуницирају.

Све ове компоненте као и полазна тачка апликације се дефинишу у Андроид манифесту. Манифест се генерише приликом креирања пројекта, за једноставније пројекте нису потребне велике модификације, осим додавања по које пермисије, нпр. пермисија за кориштење интернета, док сложенији пројекти имају велике манифесте, који имају често и преко 1000 линија кода. Манифест је XML фајл, коријенски елемент је <manifest> који саджи декларације за просторе имена који се користе и неке атрибуте. Најзначајнији атрибут у <manifest> јесте package којим се наводи гдје ће gradle build tool да изгенерише класе које се користе у пројекту, попут R. Унутар коријенског елемента налази се неки број елемената, као што су разне пермисије, али и елемент <application> који је најзначајнији. Већина елеманата унутар манифеста се дефинише унутар <application>. Атрибутима овог елемента се дефинипе подразумијевано понашање активности, тема која се користи, назив

апликације, лабела апликације, иконица, начин исписа текста[9] итд. Елементи унутар <application> углавном представљају садржај апликације (компоненте), дефинишу се са одговарајућим таговима приказаним у табели 6.1, апликација може да има већи број сваке од ових компоненти.

Табела 6.1 Компонте апликације и одговарајући елементи у манифесту

Компонента	Елемент
Activity	<activity>
Service	<service>
Content Provider	<provider>
Broadcast Receiver	<receiver>

Сваки од ових елемената мора имати android:name атрибут којим се дефинише одговарајућа класа која представља конкретну имлементацију дате компоненте. Компонента може имати и дјечији елемент <intent-filter> којим се дефинишу услови под којима се дата компонента покреће/извршава.

Поред ових компоненти постоје и друге које се користе у Андроид апликацијама, као што су фрагменти, погледи и сл.

6.1. Android Jetpack

Android Jetpack је име бренда, њега чине библиотеке, алати и смјернице које помажу апликативним програмерима да пишу апликације брже. Ове компоненте смањују boilerplate код, помажу при праћењу најбољих пракси и олакшавају комплексне задатке, стављајући фокус на писање кода битног за конкретну апликацију. Библиотеке које су дио Jetpack-a су из androidx.* пакета и нису дио Андроид ОС, што значи да се развијају и унапријеђују неовисно од система и знатно фреквентније. Чине га четири врсте компоненти:

- Foundation: Android KTX, Test, AppCompat итд,
- Architecture: DataBinding, Lifecycles, LiveData, Navigation, Room, ViewModel, WorkManager, Paging итд,
- Behavior: Preferences, Sharing, Notifications, Slices итд,
- UI: Fragment, Layout, WebView, Animation & transitions итд.

Прије него што је постојао Jetpack, постојале су разне библиотеке попут Android Support Library и Android Architecture Components. Већина тога што се налазила у овим библиотека има свој еквивалент под androidx.*, а мијешање међу овим библиотекама углавно није могуће, јер не постоји компатибилност.

Посебан значај за Котлин апликације имају Андроид Котлин екстензије које су дио Android KTX. То су библиотеке које се користе Котлин функцијама екстензијама, екстензијама особина, ламбда функцијама, именованим параметрима, параметрима са

[9] Већина језика се пише са лијева на десно, али неки језици попут арапског се пишу са десна на лијево. Ако развојни програмери желе да њихова апликација буде стварно универзална они ће додати подршку за овакав испис текста на екрану, дефинисањем атрибута android:supportsRtl=”true” у <application>.

подразумијеваним вриједностима и корутинама како би унаприједили постојеће Андроид библиотеке и додатно олакшали писања апликација у Котлину у односу на Јаву.

6.2. Кориснички интерфејс

Кориснички интерфејс Андроид апликација састоји се од неког броја виџета и контејнера који пружају начин за организацију и распоређивање тих виџета на екрану. Сви виџети су изведени из класе android.view.View. Неки од чешће кориштених виџета су: TextView, Button, ImageView, ImageButton, Switch, Checkbox, SeekBar, Toolbar и сл. Поред подразумијеваних виџета програмери могу креирати сопствене или користити виџете које су креирали други у склопу својих библиотека. Неки популарни виџети који нису дио подразумијеваних су и RecyclerView и CardView. Сваки виџет има неки скуп атрибута који су за њега дефинисани, а постоје и атрибути који су универзални за све виџете. Ако се за дефинисање корисничког интерфејса користи XML код тада су атрибути буквално и атрибути у XML коду. Атрибути се могу и подешавати директно из апликативног кода кориштењем одговарајућих особина у Котлину, односно гетера и сетера у Јави. Најважнији атрибути виџета су:

- Идентификатор, који служи за референцирање датог виџета из layout-a, за сврхе позиционирања, али и за референцирање из Котлин/Јава кода,
- Величина, маргина и padding, којима се спецификују редом величина виџета, маргина односно удаљеност виџета од осталих виџета и ивице екрана те padding као простор између садржаја виџета и његових ивица,

За дефиниње величина у Андроиду могу се користити различите јединице, слично као и у CSS, али се у пракси користе само dp (density pixel), апстрактна величина која је базраиана на физичкој величини екрана и sp (scale independent pixel) који се користи за фонтове и омогућава додатно скалирање у зависности од кориснички подешених опција за величину фонта, ако корисник не користи неке посебне опције, понаша се исто као dp. Када програмер наведе величину нпр. у пикселима, компајлер ће га упозорити да то не треба да ради јер се не може виџет скалирати, како величина једног пиксела није иста на различитим уређајима. Дизајнирање корисничког интерфејса могуће је радити директно у XML коду, али и кориштењем графичког дизајнера унутар окружења, који се користи са drag & drop функциналношћу. Прије пар година се графички дизајнер ријетко користио те и сада иако је знатно унапријеђен већина програмера идаље директно дизајнира своје корисничке интерфејсе пишући XML код.

Постоје разни контејнери за организацију виџета, који се користе у зависности од сценарија. Сваки од њих има неки основни скуп правила за организацију виџета, а у пракси се највише користи ConstraintLayout, јер је најуниверзалнији и уједно најнапреднији. Сви контејнери су изведени из класе android.view.ViewGroup, а ViewGroup је изведена из View па све оно што важи за обичне виџете важи и за контејнере.

Контејнери се дијеле у двије групе: они који организују виџете који су познати за вријеме писања апликације и они који организују виџете који су познати тек за вријеме извршања апликације, а да при томе њихов број може да варира у сваком тренутку. Главни представник прве врсте контејнера јесте ConstraintLayout, а главни представник друге врсте контејенера јесте RecyclerView.

6.2.1. ConstraintLayout

ConstraintLayout се појавио 2016. године и све што се може постићи помоћу њега може се постићи и помоћу неких старијих контејнера попут LinearLayout и RelativeLayout. Ови старији контејнери постоје још од верзије 1.0 Андроид ОС, развијени су у периоду када није постојао графички дизајнер корисничког интерфејса, па самим тим неки од њих раде лоше преко графичког дизајнера, а неки раде добро. Посљедица ове лоше интеграције је да и данас велики број програмера уопште не користи графички дизајнер. Поред тога није се водило много рачуна о перформансама када су написани ови контејнери, па они понекад могу бити спорији у егзекуцији, што се нпр. види приликом скроловања екрана. ConstraintLayout је развијен да буде компатибилан са графичким дизајнером и да понуди боље перформансе тако што поједностављује хијерархију контејнера, односно његовом употребом није потребна више тако дубока хијерархија. Додатно RelativeLayout, LinearLayout и други класични контејнери су дио Андроид фрејмворка, то значи да је могуће да се другачије понашају на различитим верзијама Андроид ОС, како ово отежава одржавање компатибилности уназад, Гугл ријетко мијења имплементације ових класа. Што значи да су унапријеђења ријетка. Насупртом томе, ConstraintLayout се излаже као дио посебне библиотеке, која се мора додати у пројекат преко одговарајуће овисности, што значи да Гугл може да га унапријеђује неовисно о ОС, па програмери могу лакше да се адаптирају на измјене и унапријеђења без страха да ће то нарушити компатибилност. Када се користи неки контејнер или виџет који долази из библиотеке односно није дио фрејмворка потребно је навести пуну путању до класе као назив XML елемента.

ConstraintLayout фунцкионише по принципу навођења ограничења за елементе, при чему сваки елемент има четири ограничења: почетно (енг. Start constraint), крајње (енг. End constraint), горње (енг. Top constraint) и доње (енг. Bottom constraint). Почетно и крајње ограничење се називају тако јер постоје језици код којих се текст почиње писати са лијева на десно, као што је српски, али и језици који се пишу са десна на лијево, па почетак код првих означава лијеву страну, а код других десну страну, а крајње ограничење означава десну односно лијеву респективно. Ограничења се манифестују преко маргина, при чему се урачунава и одређен bias, а то је вриједност помоћу које се подешава на коју страну виџет нагиње, нпр. ако виџет има ограничење за неки виџет изнад и виџет испод он је по правили центриран између њих, под условом да је одабрана иста вриједност за маргину, јер је bias подешен на 50%, али могуће је модификовати ту вриједност тако да је он ближи једном односно другом од та два виџета, слично постоји и bias, за хоризонтална ограничења. Поред ових једноставних ограничења ConstraintLayout омогућава и уланчавање елемената за рад њиховог позиционирања, као и кориштење баријера који служе као невидљиви виџети наспрам којих се може неки елемент позиционирати и још се нуди опција за поравнање текста на основу baseline неког виџета. [16]

6.2.2. LinearLayout, RelativeLayout, TableLayout и FrameLayout

LinearLayout представља Андроид верзију box модела[10] сви садржавајући елементи се смијештају у једну колону или једну врсту, што је подразмијеван приступ, у зависности од вриједности android:orientation=”horizontal/vertical” атрибута. Поред тога могуће је неком

[10] На вебу се користи CSS Flexbox [17]

елементу додијелити додатну простор науштрб осталих помоћу android:layout_weight атрибута. Ако један виџет дефинише layout_weight од 2, а други од layout_weight од 1, онда ће првом бити додијељено 2/3 простора, а другом 1/3 простора. Овај контејнер је дизајниран за једноставније корисничке интерфејсе, али је могуће помоћу њега направити и нешто сложенији интерфејс, помоћу угњеждавања, чиме се губи на перформансама, па се он не користи у те сврхе.

RelativeLayout омогућава позиционирање виџета релативно у односу на друге. Може се користити за изградњу комплексних интерфејса и по начину функционисања доста је сличан ConstraintLayout, али је дио фрејмворка и има лошије перформансе, па се ријетко користи у модерним апликацијама.

TableLayout пружа сличну функционалност на Андроиду као што то пружају табеле у HTML-у. Када се Андроид појавио CSS још увијек није био популаран на вебу, али су се зато користиле табеле за позиционирање елемената, па је овај контејнер створен да понуди исту ту функционалност на Андроид платформи. За врсте табеле се дефинише TableRow контејнер. Слични интерфејси се могу створити са ConstraintLayout и кориштењем баријера, али за сложеније табуларне интерфејсе се идаље користи TableLayout.

FrameLayout организује елементе тако што их преклапа једне преко других, користи се за стварање занимљивих визуелних ефеката, резервисање простора за неки елемент који је познат само за вријеме извршавања апликације и за оквиривање неког елемента.

6.2.3. RecyclerView

За ситуацију када је потребно приказати колекцију неких елемената, при чему знамо тачно шта су ти елементи (листа контаката, пјесме, цитати, књиге, листа задатака итд.), али не знамо унапријед величину те колекције, може бити да је по покретању апликације та колекција празна, а да се у неком временском тренутку у њу дода велики број елемената користи се RecyclerView. При чему се за поједине елементе који се приказују користе стандарни контејнери, као што су CardView и ConstraintLayout.

Главни проблем са приказивањем велике колекције елемената јесте меморијско заузеће, чак и једноставнији виџети заузимају око 1 KiB меморије, са стандардним контејнерима за приказ колекције од пар хиљада елемената било би потребно неколико MiB, што је незгодно јер апликације имају право да заузму ограничен простор у извршној меморији уређаја. Ипак сваком тренутку се на екрану уређаја може приказати само неки ограничен број виџета, па нема потребе да постоји виџет за сваки елемент колекције, већ само за оне видљиве. Ово је приницп који се користи за RecyclerView, с тим да се поред виџета за видљиве елементе креирају и виџети за елементе који су одма изнад и испод датих како се не би примјетило штекање приликом скроловања кроз колекцију.

RecyclerView се налази унутар неког контејнера, али је и за појединачне елементе потребно дефинисати одговарајући кориснички интерфејс (layout) у посебном XML фајлу.

RecyclerView ради заједно са другим класама, од којих су значајнији:

- LayoutManager који је задужен за организовање структуре приказа елемената, најједноставнији је LinearLayoutManager, којим се приказује вертикална листа елемената, а постоје и разне друге опције попут GridLayoutManager, StaggeredGridLayoutManager, а може се креирати и сопствена имплементација или користити неко third party рјешење,

- Декоратор елемената који је задужен за изглед елемената и једнсотавна позиционирања нпр. помоћу њега се наводи линија раздвајања (DividerItemDecoration) између елемената,
- Аниматор елемената који је задужен за анимације које се примјењују над појединачним елементима, али и над колекцијом од значаја је за стварање угодне анимације приликом неке модификације колекције,
- RecyclerView.ViewHolder, инстанце ове класе су задужене за мапирање елемената колекције у виџете при чему је при листању екрана потребно наново мапирати елементе у виџете када они треба да постану видљиви,
- RecyclerView.ListAdapter, адаптира податаке за њихов приказ на екрану корисника, назив потиче од адаптер обрасца којим се користи. Конструктору је потребно прослиједити имплементацију DiffUtil.ItemCallback објекат, који има конкретне имплемантације двају Котлин функција: areItemsTheSame() која враћа true ако се ради о истом елементу и areContentsTheSame() која враћа true ако два елемента имају исти визуелни приказ на екрану.

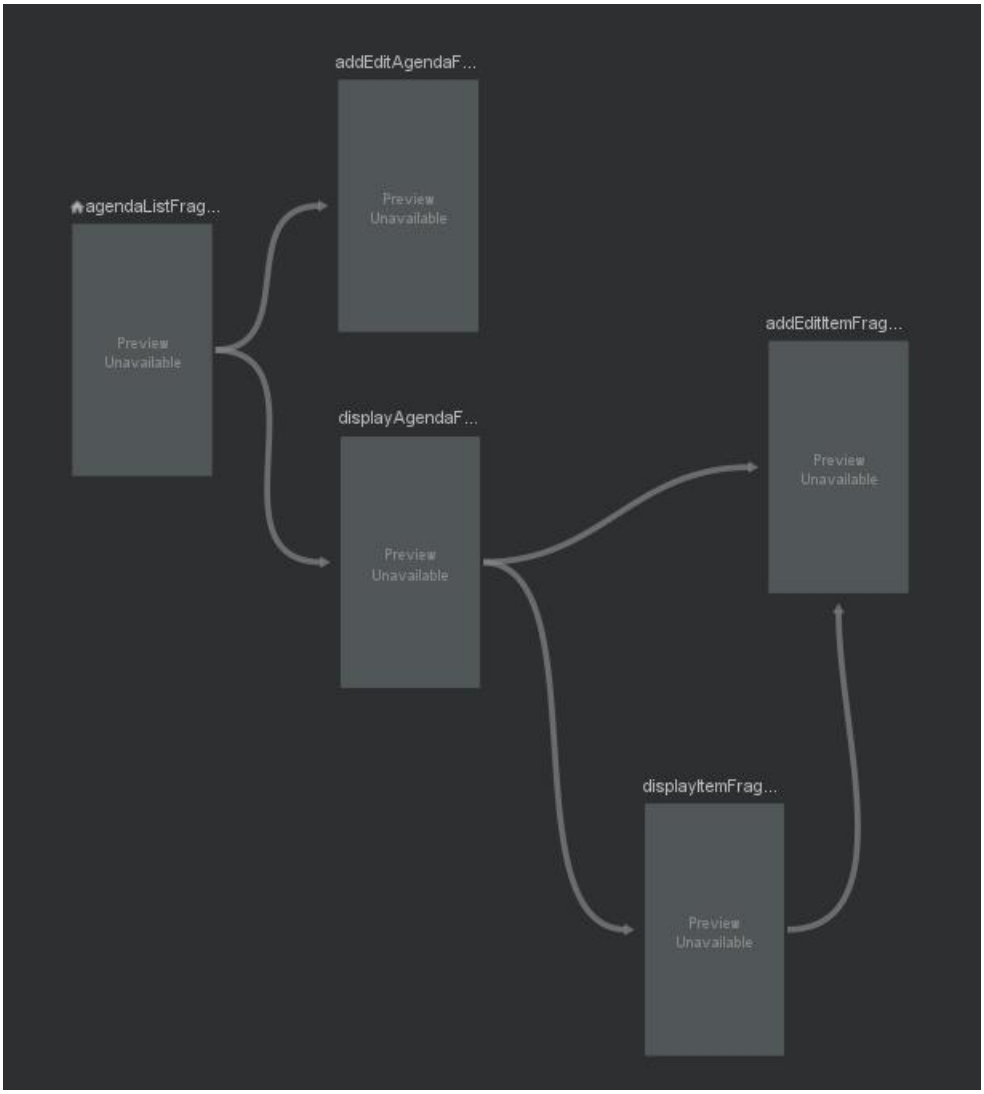
Прије RecyclerView постојали су други контејнери који су се користили у исте сврхе попут ListView који је изведен из Adapter View, али су они замијењени њиме јер је флексибилнији и може се више прилагођавати, све што је потребно јесте користити одговарајући LayoutManager. Једини изузетак је Spinner, којим се имплементира падајућа листа. [16]

6.3. Навигација

У првим верзијама Андроида, ако су програмери жељели да имплементирају више екрана користили би већи број активности при чему се са једне активности прелазило на другу позивом функције startActivity(). Како се Андроид развијао, поред активности могли су се користити и фрагменти, тако да су се за навигацију користили FragmentTransaction-и. Проблем ових приступа је што се директно нарушава приницип раздвајања надлежности, јер једна класа познаје и сувише много детаља о другој класи. Када је потребно урадити било какве измјене у коду, онда то захтјева измјену на више мјеста. Зато је Гугл осмислио Navigation API у склопу Android Jetpack-а који омогућава да се навигација врши без да је потрбно познавати да ли је екран ка ком се иде активност или фрагмент, па ако се некада јави потребе да се имплементација промјени, рецимо да се активност замјени са фрагментом тада се код за навигацију до тог екрана не мора мијењати. Детаљи о имплементацији појединих екрана су скривени од класа помоћу навигационих ресурса. Навигациони ресурси користе идентификаторе за екране и за путање до њих, из кода је само потребно навести идентификатор екрана ка ком желимо да одемо, тако да ако је потребно замјенити имплементацију једног екрана са неким другим, нема додатних измјена у коду.

Навигациони ресурси су у XML формату и дефинишу одговарајуће елементе, али није потребно директно писати код за њих јер развојно окружење нуди одговарајући графички интерфејс помоћу кога је могуће креирати навигациони граф, као што се види са слике 6.1. Навигациони ресурси се користе за навигацију са једног фрагмента на неки други фрагмент, тако да ако апликација има више активности уобичајено је да се за сваку од њих направи посебан навигациони ресурс. Главни елементи су дестинације, који могу бити фрагменти <fragment> или ако је потребно прећи на другу активност онда и активности <activity>. Дестинације имају различите дијечије елементе од који су најзначајнији акције

<action> (стрелице са једне дестинације на другу) којима се наводе све дестинације до којих се може доћи са неког екрана и аргуметни <arguments> који наводе податке које дата дестинација очекује када се до ње долази са неке друге. Навигациони ресури такође олакшавају и имлементацију анимација приликом транзиција са једног екрана (фрагмента) на неки други фрагмент и анимације се додају као елементи унутар акција, при чему разликујемо аниимације при уласку на неки екран, изласку и навигацију кроз граф уназад, али и пренос аргумената кориштењем SafeArgs, умјесто Bundle који су се користили у старијим техникама. Наиме главни проблем са Bundle је што иако подржавају различите типове података, они не намећу тип података, као што то чине Котлин и Јава па ако се прослиједи података који је ниска, а други фрагмент очекује податак који је цијели број то ће изазвати грешку. SafeArgs су аргументи који се наводе приликом навигације са једног фрагмента ка неком другом и са њима је потребно тачно навести који тип података се преноси, у позадини ће развојно окружење изгенерисати класе које служе да се наметну та ограничења за тип података. [16]



Слика 6.1 Навигациони граф

6.4. Складиштење података

За складиштење података на Андроид платформи постоји више опција:

- Чување података на фајл систему,
- Чување података у бази података,
- SharedPreferences, чување примитивних приватних података у key-value паровима,
- Чување података преко мреже.

Опција која ће бити изабрана зависи од различитих потреба:

- Колико података треба да се чува,
- Који тип података треба да се чува,
- Да ли подаци треба да буду приватни или треба да буду доступни и другим апликацијама,
- Потреба за синхронизацијом података, јер поред мобилне апликације постоје и веб апликација, рачунарска апликација и сл.

6.4.1. Чување података на фајл систему

Подаци се на Андроид платформи могу чувати на три различите локације: интерна меморија (енг. Internal storage), екстерна меморија (енг. External storage) и привремена меморија (енг. Removable storage).

Интерна меморија се не односи на читаву меморију уређаја, већ онај дио меморије који је приватан за дату апликацију и невидљив за крајњег корисника. Корисник приступа интерној меморији само кроз апликације. За писање/читање фајлова у/из итнерне меморије користе се функције класе Context, getFilesDir() и getCacheDir(). Фајлови који су смјештени у интерној меморији се бришу заједно са апликацијом, а фајлови у интерној кеш меморији се бришу спорадично од стране уређаја.

Екстерна меморија се односи на дио меморије уређаја којем имају приступ све апликације и крајњи корисник, при чему се додатно дијели на дио који је специфичан за неку апликацију и дио који се дијели између свих апликација, а оба сегмента су видљива крајњем кориснику. За приступање дијелу који је специфичан за неку апликацију користе се функције класе Context: getExternalFilesDir() и getExternalCacheDir(), а за приступање дијелу екстерне меморије који се дијели између свих апликација позивају се функције класе Environment: getExternalStorageDirectory() и getExternalStoragePublicDirectory(), преко њих је могуће манипулисати фајловима других апликација и Гугл је одлучио да се подаци овде не требају више чувати па су ове методе застарјеле (енг. Deprecated) почевши од Андроид верзије 10 и више се не могу користити.

Привремена меморија се односи на microSD картице, USB уређаје или било шта што се може привремено повезати на уређај. За приступање привременој меморији користе се функције класе Context: getExternalFilesDirs() и getExternalCacheDirs(), које враћају низ File објеката, при чему се први односи на екстерну меморију, а други који постоји само ако је прикључен неки уређај показује на директоријум са тог уређаја.

Прије Андроид верзије 10, било је потребно додати одговарајуће пермисије за рад са екстерним фајловима у манифесту апликације, али како се од ове верзије ОС не може више уписивати на дијељене локације тако више и није потребно захтијевати пермисије. [16]

Битно је напоменути да иако званично корисници немају приступ подацима који су смијештени у интерној меморји уређаја, ако они имају root приступ тада могу приступити свим подацима, али се подаци могу криптовати како би се онемогућило њихово читање од неовлштетних лица.

6.4.2. SharedPreferences

SharedPreferences је API који омогућава чување парова кључ-вриједност у меморији телефона, примарно се користи за чување конфигурационих подешавања апликације, као што су филтри при претрази, редослијед сортирања ставки, мјесто до ког је корисник стигао и сл. Андроид платформа нуди подршку за креирање корисничког интерфејса за преференсе и по свом изгледу овај интерфејс је сличан стандардној апликацији за подешавања. Могу се чувати само примитивни типови и ниске, као и скупови ниски Set<String>. За упис у неку preference датотеку користе се функције објекта SharedPreferences.Editor. Нова preference датотека може се креирати или јој се може приступити позивом getSharedPreferences() над класом Context гдје се може експлицитно навести назив датотеке, овај приступ се користи када се датотека треба дијелити у више контекста. Позивом getPreferences() над неком Activity класом, гдје је назив датотеке исти као назив класе, овај приступ се користи за чување конфигурација специфичних за баш ту активност. За подешавање конфигурација које треба да буду видљиве кроз читаву апликацију користи се getDefaultSharedPreferences() функција која враћа подразумијевану preference датотеку за читаву апликацију. За читање из preference датотеке се користе функције попут: getInt(), getLong(), getFloat() и сл. којима се прослиједи кључ и опционално неке подразумијевана вриједност ако кључ не постоји.

6.4.3. Чување података у бази података

За чување сложенијих података или велике количине података локално, најбоља је опција база података и Андроид платформа има уграђену подршку за SQLite RDBMS. Иако је могуће директно радити са SQLite API-јем, у пракси се користи неки виши ниво апстракције попут Room библиотеке. [18]

Room је имплементација објектно-релационог мапера (енг. Object-relational mapper – ORM) и дио је Android Jetpack-a. Постоје и разна third party рјешења која се додуше све мање користе јер нису официјелна и самим тим немају подршку Гугла. Да би се Room могао користити потребно је додати артифакте за ивршну библиотеку и за процесор анотација, при чему је најнижи подржана верзија Андроид API-ја 15. Основне компоненте односно класе у Room су:

- Ентитети (енг. Entities), представљају табеле у бази,
- Data Access Object (DAO), садржи функције које се користе за рад са базом података,
- База података (енг. Database), повезује све ентитете и DAO класе за дату SQLite базу података

У Котлину се за ентитете користе класе података, а да би класа представљала ентитет потребно је додати анотацију @Entity над класом и анотацију @PrimaryKey над једној од особина те класе, при чему се за примарни кључ не смије користити нулабилан тип. Назив класе је уједно и назив табеле у бази података, али га је могуће измијенити навођењем tableName параметра при @Entity анотацији. Особине класе се мапирају у колоне табеле

при чему је њихов назив исти као назив одговарајуће особине, али се може измјенити тако што се особина анотира са @ColumnInfo(name=”naziv_kolone”). Примарни кључ може бити композитан, односно да га чини више особина, гдје се онда особине које чине примарни кључ спецификују при @Entity анотацији са параметром primaryKeys=[“osobina1”, … “osobinaN”]. Могуће је креирати додатне индексе поред примарног кључа, анотирањем особине са @ColumnInfo(index=true).

DAO је API за рад са неком табелом у бази података. Идеја је да се унутар њега наводе све CRUD операције за неку табелу, тако да углавном за сваки ентитет постоји и одговарајући DAO, мада то не мора увијек бити случај, поготово када је ријеч о слабим ентитетима. У Котлину се за DAO користи анотација @Dao којом се анотира нека апстрактна класа или интерфејс, а конкретну имплементацију ће изгенерисати Room процесор анотација. Функције за приступ бази података се не пишу директно, потребно је само анотирати функцију са одговарајућом анотацијом и навести повратни тип података. @Query анотација се користи за SELECT упите, мада је универзална и може се користити за било који тип упита. @Insert користи се за упис података у базу, @Update за ажурирање података из базе и @Delete за брисање података из базе. Све ове функције се извршавају синхроно, што значи да се на резултат њиховог извршавања мора чекати, али тиме би се блокирао кориснички интерфејс док се чека на њихово извршавања, зато Room блокира писање кода који се извршава на главној апликационој нити, већ захтјева да се сав рад са базом података иврша у позадини, како би кориснички интерфејс остао интерактиван све вријеме. Постоје разни реактивни приступи за рјешавање овог проблема. За функције анотиране са @Query може се користити LiveData, тако што се повратна вриједност уоквири са LiveData, нпр. LiveData<List<Student>>. Предност овог приступа је што није потребна никаква додатна библиотека и извршавање се одвија на позадинској нити, али мана је што се не може користити за три остале врсте анотација, те што се резултат извршавања увијек враћа на главној апликационој нити. Котлин приступ за рјешавање овог проблема јесте употреба корутина. Све што је потребно урадити јесте означити неку функцију са suspend и она ће се извршавати на некој позадинској нити, а резултат иршавња могуће је добити како на главној нити, тако и на некој другој нити, ако за то постоји потреба. Корутине се позивају из одговарајућег CoroutineScope, као што је viewModelScope који је исто дио Android Jetpack-a. Иако suspend ради са било којом од анотација, за @Query се чешће користи Flow, што је варијанта са корутинама слична LiveData приступу, али нуди и све остале предности које имају suspend функције. Предност корутина у односу на било који други приступ јесте да су оне дио Котлин програмског језика, па је самим тим рад са њима најлакши, али је мана што су корутине релативно нове те су тек подржане од Котлин верзије 1.3, па ће проћи још неко вријеме док не постану једнако стабилне као остале варијанте. За апликације писане у Јави користи се RxJava, али се ова библиотека може користити и из Котлина, предност ове варијанте је што је најстарија и самим тим стабилнија од осталих, али мана је што захтјева пуно више времена да се схвати и савлада. [18]

За рад са ентитетима и DAO интерфејсима потребна је бар једна апстрактна класа анотирана са @Database која је наслијеђена из класе RoomDatabase. @Database прихвата два обавезна параметра, entities којим се наводи листа ентитета и version којим се наводи тренутна верзија базе података, ова вриједност се инкрементује када год се мијења шема базе података. За кориштење DAO интерфејса, потребно је унутар ове класе дефинисати одговарајуће апстрактне методе које као повратни тип имају тај DAO интерфејс, с тим да није битан назив функција. Уобичајено је да у пројекту постоји само једна класа анотирана

са @Database која повезује све ентитете и DAO интерфејсе, али је могуће да их буде више. Ова класа би требало да буде имплементирана као singleton, што се најлакше постиже коришћењем библиотека попут Koin или Kodein. Како је у питању апстрактна класа, да би се користила конкретна инстанца позивају се функције над класом Room: databaseBuilder() која враћа RoomDatabase.Builder помоћу ког се прави база података на фајл систему уређаја и inMemoryDatabaseBuilder() која враћа RoomDatabase.Builder помоћу ког се прави база података која је у извршној меморији уређаја и користи се само за сврхе тестирања.

Room нуди мноштво опција за рад са базом података и у овом сегменту су обрађене само основне ствари, али ту је и подршка за миграцију базе података, трансакције, подршка за прогресивно учитавање података, претрага пуног текста итд.

6.5. Рад са мрежом

Друштвене мреже, мејл клијенти, стриминг сервиси, временске прогнозе су само неке од апликација које захтјевају мрежну конекцију за рад са удаљеним ресурсима. Андроид платформа пружа апликационим програмерима мноштво опција за обављање овог задатака, могуће је директно користити стандардне Јава сокете или HttpUrlConnection, али се у пракси користе библиотеке које пружају виши ниво апстракције, ту убрајамо [16]:

- OkHttp, најпопуларнија опција за генеричке HTTP захтјеве, редовно се ажурира и добро је подржана од стране програмерске заједнице. Не ради са HttpUrlConnection већ директно користи сокете. Многе друге библиотеке у позадини користе OkHttp и на њега се надграђују,
- Volley, ово је Гуглова библиотека за приступ интернету која у позадини користи HttpUrlConnection и једно вријеме је била јако популарна. Google Play Store као и неке друге Гугл апликације користе Volley,
- Retrofit, библиотека која у позадини користи OkHttp, тренутно је најпопуларнија опција за апликације које требају приступ интернету и у многоме олакшава рад са REST сервисима. Клијентски API се имплементира коришћењем одговарајућих анотација. За конверзију података се могу користити разне опције, а у Котлину је најпопуларнија библиотека Moshi,
- Apollo-Android, GraphQL клијент за Андроид платформу,
- Glide, библиотека која се користи за учитавања слика са интернета, у позадини користи OkHttp, па се добро интегрише са другим библиотекама који користи OkHttp као што је Retrofit итд.

Рад са мрежом подразумијева одговарајуће пермисије које се наводе у манифесту апликације, а то су:

- <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>, пермисија која омогућава провјеру доступности мрежне конекције и
- <uses-permission android:name="android.permission.INTERNET" />, пермисија која је потребна за приступ мрежи.

Такође забрањено је извршавање мрежних захтјева на главној апликационој нити, а ако се то ипак покуша, апликација ће се срушити услијед NetworkOnMainThreadException изуетка. Забрана постоји из разлога, што мрежни саобраћај је често веома спор и

непредвидив и немају сви корисници у сваком тренутку приступ брзој мрежи, па би за њих извршавање на главној нити блокирало кориснички интерфејс на дуже стазе чинећи апликацију недовољно интерактивном. Иако постоји начин да се омогући извршавање на главној нити ово се не препоручује, а већина библиотека свакако има подршку са асинхрону обраду захтјева. У Котлину се у ове сврхе користе корутине, које у многоме олакшавају тај процес за апликативне програмере. Додатно ограничене постоји од Андроид верзије 8+ гдје се забрањује коришћење HTTP протокола директно, умјесто тога обавезно се користи HTTPS протокол, али како још увијек постоје странице које користи само HTTP, а оне често нису под директном контролом апликативних програмера, могуће је омогућити овакву незаштићену комуникацију с тим да се предлаже адекватно обавијештавање крајњих корисника о могућим импликацијама такве једне комуникације. [16]

6.6. Припрема апликације за дистрибуцију

Прије него што се апликација може дистрибуирати крајњим корисницма потребно је да се она дигитално потпише одговарајућим сертификатом, да се минимизује и обфускује те ако је то циљ да се имлементира монетизација.

6.6.1. Дигитално потписивање Андроид апликације

Андроид апликације морају имати јединствен идентификатор који се користи од стране канала за дистрибуцију како би се нека апликација могла разликовати у односу на све остале. Не могу се појавити двије апликације које имају исти идентификатор, поред тога свака апликација мора навести верзију апликације и мора бити дигитално потписана. Ако се не би користио дигитални потпис, онда ништа не би спријечавало трећа лица да направе неку малициозну апликацију и додјеле јој исти идентификатор, инкрементују број верзије и као такву је дистрибуирају. Такође када се нека апликација ажурира потребно је да има исти идентификатор, већу вриједност versionCode-а и да је дигитално потписана од стране истог дигиталног сертификата као и све претходне верзије те апликације. На основу дигиталног потписа могуће је провјерити и да ли је апликација измијењена од дистрибутера, док год се за дигитално потписивање користи сопствени сертификат, а не сертификат дистрибутера, програмер је под контролом над својом апликацијом, у свим другим случајевима се то не може са сигурношћу тврдити. [16]

За апликације које су још у развоју користи се debug.keystore, како се на сваком рачунару користи другачији debug.keystore, тако настају проблеми када се апликација развија од стране више програмера, зато се углавном одабере један debug.keystore који ће да користе сви програмери који су дио тог развојног тима. [16]

За апликацију која иде у продукцију потребно је користити посебан дигитални сертификат, при чему се треба водити рачуна о периоду валидности, јер када тај период истекне не може се апликација више ажурирати. Битно је за нагласити да су продукциони и развојни сертификати самопотписани, тако да се њима осигурава да апликација није на било који начин измијењена, али се не могу користити за потврду идентитета. [16]

Кључеви се могу правити директно из развојног окружења, а могу се и користити нека друга рјешења попут keytool или OpenSSL.

Потребно је осигурати дигитални сертификат, тако да га нико не украде јер би у супротном крадљивац могао да искористи сертификат да објави своју верзију апликације

која је валидна са становишта неког канала дистрибуције и да се не изгуби сертификат или лозинка за његову употребу јер се у супротном губи могућност за ажурирање апликације. [16]

6.6.2. Обфускација Андроид апликације

Обфускација је поступак којим се смањује читљивост изворног кода, тако да није од користи за малициозне кориснике, а да при томе апликација остане потпуно функционална. Наиме, врло је једноставно искористити неки алат за декомпајлирање изворног кода, многи од њих су потпуно бесплатни, тако да је изворни код Андроид апликација на дохват руке ако се не обфускује, чиме програмер излаже своју интелектуалну својину опасности, али и крајњег корисника јер се могу појавити пиратске верзије апликације на тржишту које компромитују приватност. С друге стране, злонамјерни програмери чије апликације треба да на неки начин злоупотребе крајњег корисника се такође могу користити обфускацијом да сакрију свој малициозни код. Не треба се заваравати да је немогуће доћи до изворног кода, али функција обфускатора као алата који врши обфускацију није да у потпуности онемогући реверзни инжињеринг изворног кода, већ да такав поступак учини економски неисплативим за свакога ко би то покушао. [20]

На Андроид платформи стандардни обфускатор је ProGuard. То је бесплатан алат, а постоји и професионална верзија под називом DexGuard која нуди више опција и већи степен сигурности, али и разни други професионални алати чије цијене могу бити јако високе у зависности од потреба апликативних програмера. За већину апликација ProGuard нуди довољан ниво заштите, али за апликаицје које су комерцијално јако успијешне свакако да се исплати уложити у професионални овфускатор.

Неке од техника обфускације су [20]:

- Лексичка обфускација, измјена назива свих класа, функција и варијабли у изворном коду како би се сакрило њихово значење,
- Уметање бескорисног кода, код који ништа не ради се умеће у изворни, постоје алати који уклањају бескористан код,
- Обфускација података, измјена структура података које се користе у апликацији,
- Енкрипција ниски, ниске у изворном коду могу открити много информација чак када су измијењени називи варијабли, критичне секције у коду се могу открити из порука о грешкама, да би се ово онемогућило могуће је енкриптовати ниске, при чему се врши декрипција током извршавања програма, што донекле успорава извршавање,
- Anti-tamper, обфускатор може додати код у апликацију којим се провјерава да изворни код апликације није мијењан, а ако јесте онда може да сруши апликацију прикривеним изузетцима или да пошаље нотификацију развојним програмерима,
- Anti-debug, обфускатор може додати код којим се детектује да се апликација покреће у дебагеру. Ако се користи дебагер, намјерно се измијене вриједности варијабли или се баци неки изузетак са лажним разлогом, због прикривања или се пошаље нотификација развојним програмерима,
- Уклањае мета података и података који се користе од стране дебагера,
- Повезивање или спајање кориштених библиотека у једну,
- Уметање инструкција које се никада неће извршити,
- Обфускација инструкција гранања, дио кода се измијешта у посебне функције или се код који се извршавао у посебној функцији умеће директно у одговарајући блок.

6.6.3. Минимизација Андроид апликације

Прије него што се апликација објави потребно је анализирати колико простора она заузима на диску корисника. Иако су мобилни уређаји данас у стању до похране велике количине података, ипак постоје одређене рестрикције на величину апликације од стране дистрибутера, на Google Play Store апликације не смију бити веће од 100 MiB. Потенцијални корисник можда користи плаћени интернет, па апликације које заузимају много простора могу бити одбојне за њега или можда нема још много простора на диску. Због ових разлога потребно је извршити минимизацију апликације. [16]

Први корак је анализа овисности у апликационом build.gradle фајлу. Овисности које се користе приликом процеса тестирања нису дио APK фајла апликације, па се не требају разматрати, док остале овисности улазе у састав APK фајла. Поред експлицитно наведених овисности, постоје и оне транзитивне овисности, које се довлаче као овисности за неку од експлицитно наведених, нпр. Retrofit користи OkHttp у позадини па је то транзитивна овисност за њега. Неке библиотеке су кориштене у експерименталне сврхе па се могу уклонити, друге су кориштене на једном или два мјеста па је можда боље да се умјесто библиотеке напише сопствена имплементација, ако те неће сувише дуго времена да одузме. За све остале овисности се коментарише њихова декларација па се апликација покуша компајлирати без ње, ако компајлирање прође то значи да је то сувишна овисност, ако не прође онда се не може уклонити. [16]

Код који програмери напишу сами за своје апликације они ће и да користе, у супротном га не би ни писали, али код који долази од библиотека не мора нужно сав да се користи, па је тада згодно да се уклоне оне класе, функције и особине из библиотеке које апликација не треба. У те сврхе се користи ProGuard или R8. Ови алати уклањају код за који процијене да је непотребан за APK, али понекад се деси да погријеше у процијене па је могуће експлицитно укључити неке од класа, функција и особина за које је погрешно процијењено да су непотребни, унутар proguards-rules.pro фајла, који користе оба алата. [16]

За крај потребно је уклонити неискориштене ресурсе из апликације, то је могуће урадити мануелно из развојног окружења или приликом процеса минимизације од стране алата попут ProGuard. Многе од библиотека које апликација користи имају дефинисане string ресурсе за разне језике, али ако апликација користи само подскуп тих језика онда су сви остали сувишни, тако да је могуће експлицитно навести кориштене језике, тада се сви сувишни ресурси уклањају из APK фајла. [16]

Након уклањања неискориштених ресурса, потребно је минимизовати оне који се стварно користе од стране апликације, то су у првом реду слике. Пожељно је да се PNG и JPEG слике замијене са SVG сликама, како слике у векторском формату заузимају мање простора, али имају и ту предност што се скалирају да добро изгледају на било којој величини екрана. Понекад ово није могуће, јер дизајнер није понудио слику у SVG формату, па се тада предлаже да се слике из PNG и JPEG формата конвертују у WebP формат, који нуди бољи степен компресије, а да при томе слика задржи прихватљив квалитет, а ако програмери не желе да користе овај формат могуће је користити алате који врше додатну компресију PNG односно JPEG слика не мијењајући формат. [16]

Процес минимизације могуће је аутоматизовати и прилагодити посебно за одређени уређај коришћењем App Bundles за дистрибуцију апликација. Мана овог приступа је што се мора користити Гуглов сертификат за дигитално потписивање апликације, па програмер није више у контроли над сопственом апликацијом. [16]

6.6.4. Монетизација

Мобилне апликације могу се монетизовати на различите начине:

- Куповина у апликацијама, користи се за продају додатних функционалности, добара у видео игрицама или за перманентно отклањање реклама,
- Модел претплатника, апликација нуди додатне функционалности или додатна добра у видео игрицама која се обнављају периодично када се и врши наплата,
- Рекламе, апликација приказује рекламни садржај за одговарајућу накнаду,
- Плаћене апликације, корисници морају платити тражену цијену прије преузимања
- Е-комерц, апликације се могу користити за продају физичких добара или сервиса

За куповину у апликацијама, модел претплатника и плаћене апликације користи се Google Play Billing библиотека. Потребно је креирати Google Wallet Merchant налог на Google Play Console сервису који се користи за дистрибуцију апликација на Google Play Store-у, анализирање рада апликације и мноштво других ствари везаних за апликацију. Након креирања налога потребно је дефинисати производе који се могу купити кроз апликаицју, интегрисати Billing API у апликацију и додати механизме за откључавање купљених функционалности или добара. Такође могуће је и само апликацију означити као плаћену. Е-комерц је тренутно ограничен на мали број земаља, па се чешће користе властита ријешења у ове сврхе.

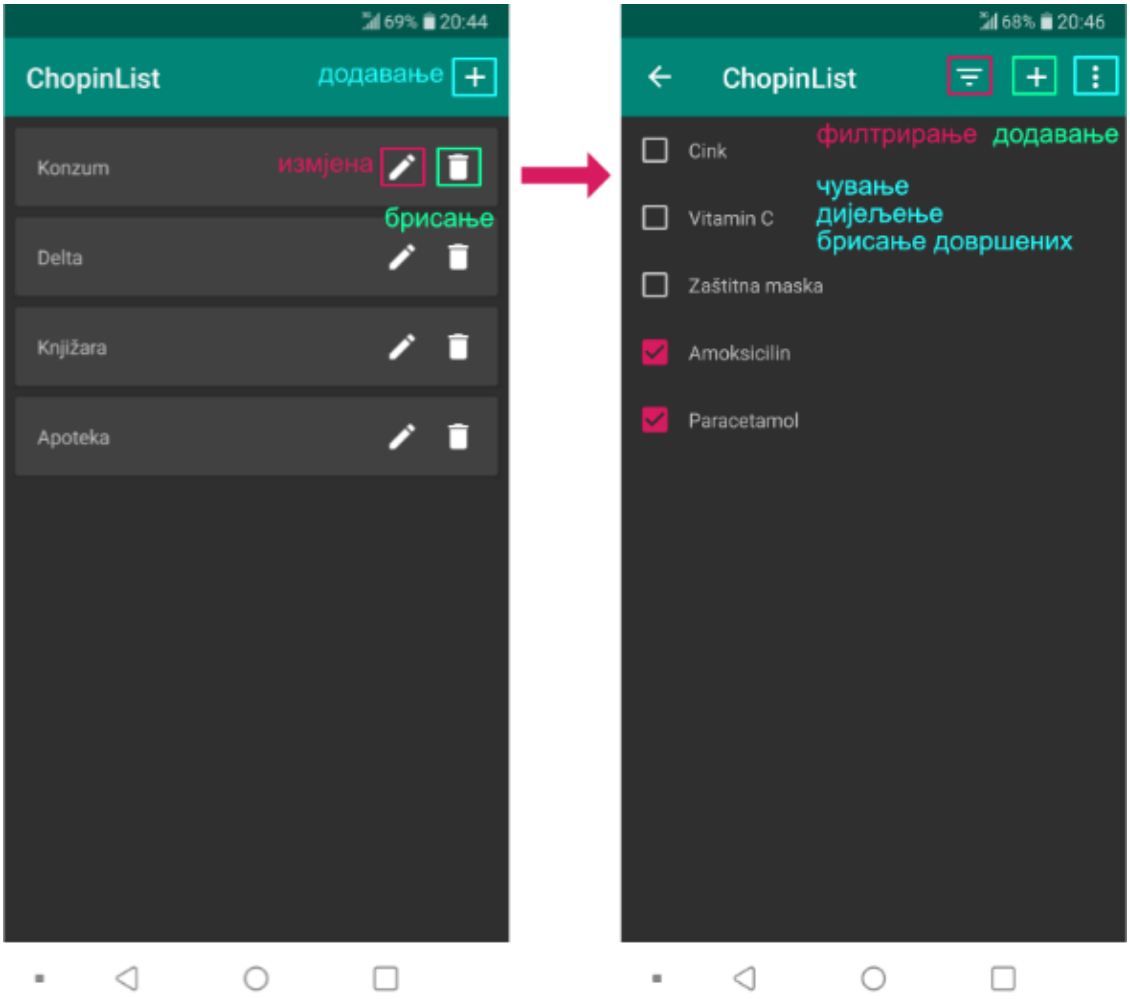
За приказивање реклама највише се користи AdMob сервис. Потребно је додати Google Mobile Ads SDK у пројекат. AdMob је сервис који користи преко милион апликација, али и велики број advertiser-a. AdMob се бави проналаском и приказом релевантних реклама за крајњег корисника, а апликативни програмер је дужан да одабере оне сегменте за приказ реклама који су у складу са његовом апликацијом. Тако апликације које су за дјецу не могу приказивати рекламе на којима је садржај неприкалдан за тај узраст, не поштовање ових рестрикција ће довести до отклањања апликације са Play Store. Добре праксе налажу да се рекламе додају у кориснички интерфејс тако да изгледају да су дио њега и да нису напорне за крајње кориснике. Постоје различите врсте реклама које се могу приказивати као што су банери, interstitial, видео рекламе и сл.

7. ПРАКТИЧНИ ДИО

За демонстрацију имплементиране су двије апликације, прва је базирана на MVI архитектуралном стилу и користи локалну базу података, а друга је базирана на MVVM архитектуралном стилу и представља REST клијент за рад са удаљеним ресурсима.

7.1. ChopinList

Прва апликација се назива ChopinList. То је апликација која омогућава корисницима прављење листи намјерница односно ставки које желе да набаве приликом куповине, приказана је на слици 7.1.



Слика 7.1 ChopinList

Почетни екран (фрагмент) приказује све агенде које је корисник направио до тада, на располагању су му опције за додавање нове агенде, измјену назива постојеће и брисање неке од агенди. Агенда садржи списак ставки, које карактеришу назив, количина, цијена, мјерна јединица и напомена те стање, које може бити довршено и недовршено у зависности од тога да ли је корисник набавио дату ставку до сада. Екран (фрагмент) који приказује све

ставке нуди опције за додавање нових ставки, измјену и преглед постојећих, као и опције за филтрирање ставки гдје се може назначити да се приказују само оне ставке које су довршене или само оне које нису, те приказ свих ставки, поред тога ту је и опција за брисање довршених ставки те коначно за дијељење агенде или чување у HTML формату, гдје се кориснику нуди списак апликација/опција које може искористити у ову сврху.

ChopinList демонстрира рад са разним Android Jetpack библиотекама, као и MVI архитектуралним стилом. Коришћене библиотеке као и плагини приказани су на сликама 7.2-3.

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    implementation 'androidx.cardview:cardview:1.0.0'
    implementation 'androidx.fragment:fragment-ktx:1.2.1'
    implementation "androidx.lifecycle:lifecycle-livedata:2.2.0"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.2.0"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation "org.koin:koin-core:$koin_version"
    implementation "org.koin:koin-android:$koin_version"
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    implementation "com.github.jknack:handlebars:4.1.2"
    kapt "androidx.room:room-compiler:$room_version"
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Слика 7.2 Коришћене библиотеке у app.build.gradle

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'
apply plugin: 'androidx.navigation.safeargs.kotlin'
```

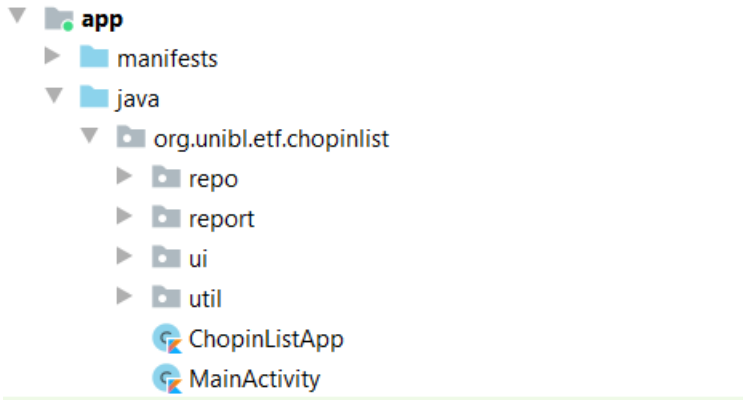
Слика 7.3 Коришћени плагини у app.build.gradle

Већина библиотека је дио Android Jetpack-а који је обрађен у поглављу пет, поред тих библиотека коришћене су и неке 3rd party библиотеке, а то су: Koin за уметање овисности и Handlebars, за генерисање извјештаја у HTML формату. Структура пројекта приказана је на слици 7.4, пројекат је подијељен на неколицину пакета према функцији коју класе у датом пакету обављају. Пројекат користи образац са репозиторијумом, тако да се не приступа директно ViewModel-има већ се приступ врши преко репозиторијума, што значи да ако се у позадини промјени нека логика везана за рад са подацима, сама апликација неће

морати водити бригу о томе јер она се користи само интерфејсом, исто тако ако се дода нова логика нпр. за рад са удаљеним ресурсима неће се морати мијењати ништа на вишем нивоу што се тиче логике. Како постоје два ентита, а то су агенда и ставка тако и постоје:

- Agenda ентитет имплементиран као класа података са одговарајућим DAO интерфејсом и AgendaRepository репозиторијум,
- Item ентитет такође имплементиран као класа података са одговарајућим DAO интерфејсом и ItemRepository репозиторјиум

Ови репозиторијуми се користе из одговарајућих ViewModel-а. На вишем нивоу односно у погледима се користи RecyclerView поглед који омогућава ефикасно приказивање велике листе ставки, као и навигацију такве једне листе. Овај поглед се користи како са приказ листе агенди, тако и за приказ ставки унутар неке од агенди.



Слика 7.4 ChopinList структура

Да би се омогућила тестабилност апликације, али и код одржавао једноставнијим, апликација користи принцип инверзије овисности, односно уметање овисности путем Koin, који захтјева да се дефинише класа за апликацију унутар које се дефинишу све овисности. Koin у позадини умеће све потребне овисности, а све што је потребно да програмер уради јесте да наведе овисности као параметре конструктора, приказ класе апликације дат је на слици 7.5.

```
class ChopinListApp : Application() {
    private val koinModule = module { this: Module
        single { AppDatabase.newInstance(androidContext()) }
        single { this: Scope
            val db: AppDatabase = get()
            AgendaRepository(db.agendaStore()) ^single
        }
        single { Handlebars() }
        single { this: Scope
            val db: AppDatabase = get()
            ItemRepository(db.itemStore()) ^single
        }
        single { ItemRosterReport(androidContext(), get()) }
        viewModel { AgendaRosterMotor(get()) }
        viewModel { (agendaId: String) -> ItemRosterMotor(get(), get(), get(), agendaId) }
        viewModel { (modelId: String) -> SingleAgendaMotor(get(), modelId) }
        viewModel { (modelId: String) -> SingleItemMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin { this: KoinApplication
            androidLogger()
            androidContext( androidContext: this@ChopinListApp)
            modules(koinModule)
        }
    }
```

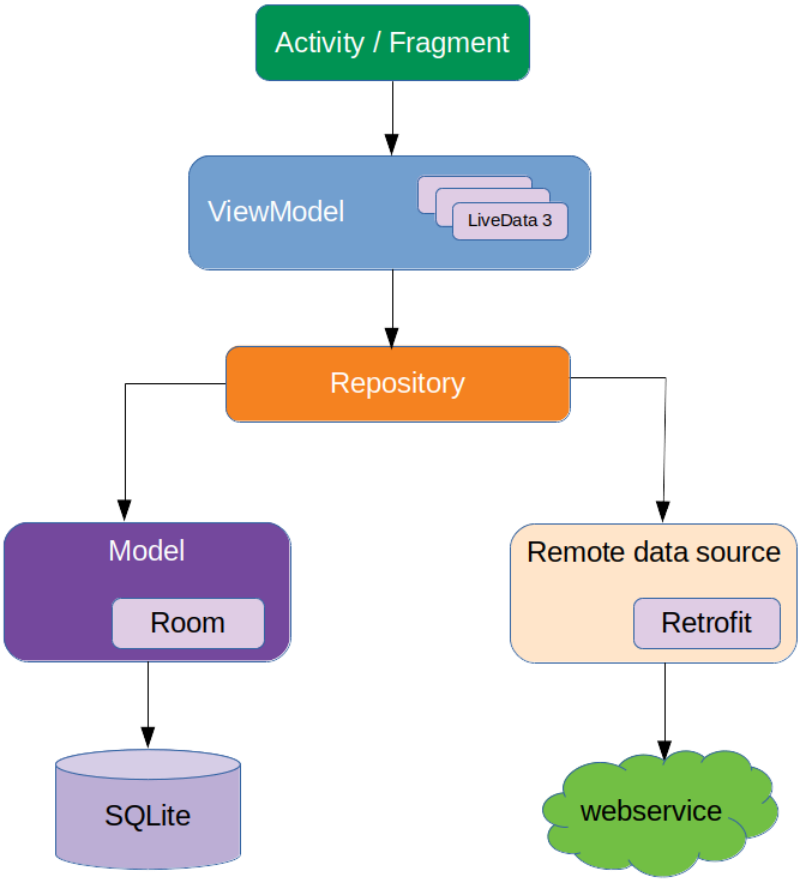
Слика 7.5 Овисности апликације

7.2. QuotesList

Друга апликација назива се QuotesList. То је апликација која кориснику омогућава да након регистрације (пријаве) на систем преглед постојеће и креира нове цитате. Ова апликација демонстрира рад са удаљеним ресурсима и користи MVVM архитектурални стил. Поред Android Jetpack библиотека користе се и неке third party библиотеке: Koin, Retrofit као REST клијент и gson за конвертовање података.

QuotesList се у позадини качи на REST сервис написан помоћу Python програмског језика и микро фрејмворка Flask, приступ сервису је ограничен само на кориснике који имају одговарајућу ауторизацију помоћу JWT токена, који се генерише приликом сваког успијешног пријављивања на сервис и при регистрацији новог корисника. Сваки токен има ограничено трајање те се означава ништавним након истека трајања или након што се корисник одјавио, ради веће сигурности. При слању захтјева ка сервису потребно је послати дати JWT токен, ако је он којим случајем неправилан или је истекао, сервис ће одбити да обради захтјев те ће кориснику упутити одговарајућу нотификацију о насталој грешци. Комуникација је додатно „заштићена“ и употребом HTTPS (HTTP + TLS) протокола. Сервис се сервира преко Heroku и у позадини се користи PostgreSQL база и gunicorn сервер.

Архитектурална шема на којој је ова апликација базирана, приказана је на слици 7.6.



Слика 7.6 Поједностављен приказ архитектуре QuotesList

Рад са апликацијом почиње тако што се корисник пријави или региструје ако већ нема кориснички налог, с тим да ако није доступан интернет тада ће апликација да нотификује корисника о томе јер аутентификација није могућа. REST сервис по успијешној аутентификацији враћа податке о кориснику изузев лозинке, поруку о успијешности и одговарајући JWT токен. Кориснички подаци и JWT токен се кеширају локално, како би се могли користили за рад са сервисом, те да се корисник не би морао сваки пут пријављивати када покреће апликацију. Након тога апликација довлачи све цитате са удаљеног сервиса ако они постоје, те их такође кешира локално, тако да нема потребе да се сваки пут подаци довлаче, већ само након одређеног временског периода. Могуће је додати нове цитате, они се онда смијештају у локалну базу, али се и шаље POST захтјев којим се цитат чува и на серверској страни. Цитати се могу обрисати, тада се они отклањају из локалне базе, али се и шаље DELETE захтјев којим се цитати брише и на серверу. Како је у ова апликација развијена само у демонстративне сврхе, није се водило рачуна о синхронизацији података.

Како се подацима не приступа директно већ само преко репозиторијума, могуће је да се имплементација промјени, а без да се мијења остатак логике. На слици 7.7 је приказана изглед QuotesList апликације. (требам тек додати слику)

8. ЗАКЉУЧАК

Ово је закључак, није још написан.

ЛИТЕРАТУРА

[1] Wikipedia, “Android version history”, https://en.wikipedia.org/wiki/Android_version_history, посјећено: 26.02.2020.

[2] Wikipedia, “Long-term support”, https://en.wikipedia.org/wiki/Long-term_support, посјећено: 26.02.2020. године.

[3] Android developers, “Distribution dashboard”, https://developer.android.com/about/dashboards/, посјећено: 26.02.2020. године.

[4] Digital trends, “What is Android fragmentation, and can Google ever fix it?”, https://www.digitaltrends.com/mobile/what-is-android-fragmentation-and-can-google-ever-fix-it/, посјећено: 26.02.2020. године.

[5] Guiding Tech, “Android Wake-locks, How to detect and Fix them”, https://www.guidingtech.com/45384/android-wakelocks/, посјећено: 27.02.2020. године.

[6] Wikipedia, “Google v. Oracle America”, https://en.wikipedia.org/wiki/Google_v._Oracle_America, посјећено: 27.02.2020. године.

[7] Open Source Initiative, “Apache Licence, Version 2.0”, https://opensource.org/licenses/Apache-2.0, посјећено: 27.02.2020. године.

[8] Yunhe Shi, Kevin Casey, M. Anton Ertl, и David Gregg. 2008. Virtual machine showdown: Stack versus registers. ACM Trans. Archit. Code Optim. 4, 4, Article 2 (January 2008), 36 страница. DOI: https://doi.org/10.1145/1328195.1328197

[9] Mark L. Murphy, *Elements of Kotlin 0.2*, CommonsWare, LLC, United States of America, 2019.

[10] Mark L. Murphy, *Elements of Kotlin Coroutines 0.1*, CommonsWare, LLC, United States of America, 2019.

[11] Josh Skeen, David Greenhalgh, *Kotlin Programming the Big Nerd Rench Guide*, Big Nerd Rench LLC, Arizona 2018

[12] Florina Muntenescu, *Android Architecture Patterns Part 1: Model-View-Controller*, https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6, посјећено: 11.03.2020. године.

[13] Florina Muntenescu, *Android Architecture Patterns Part 2: Model-View-Presenter*, https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5, посјећено: 11.03.2020. године.

[14] Florina Muntenescu, *Android Architecture Patterns Part 3: Model-View-ViewModel*, https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b, посјећено: 11.03.2020. године.

[15] Mark L. Murphy, *Exploring Android 1.0*, CommonsWare, LLC, United States of America, 2019.

[16] Mark L. Murphy, *Elements of Android Jetpack 0.8*, CommonsWare, LLC, United States of America, 2020.

[17] W3C, "CSS Flexible Box Layout Module Level 1", https://www.w3.org/TR/css-flexbox-1/, посјећетно: 23.03.2020. године.

[18] Mark L. Murphy, *Elements of Android Room 0.1*, CommonsWare, LLC, United States of America, 2020.