Usporedba Flutter i nativne Android aplikacije

Kovačević, Matej

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:018131

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-14**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni diplomski studij

USPOREDBA FLUTTER I ANDROID NATIVNE APLIKACIJE

Diplomski rad

Matej Kovačević

Osijek, 2023.

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

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 05.05.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Matej Kovačević
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1133R, 13.10.2020.
OIB studenta:	40867740983
Mentor:	izv. prof. dr. sc. Mirko Köhler
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Ivica Lukić
Član Povjerenstva 1:	izv. prof. dr. sc. Mirko Köhler
Član Povjerenstva 2:	Miljenko Švarcmajer, mag. ing. comp.
Naslov diplomskog rada:	Usporedba Flutter i nativne Android aplikacije
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U diplomskom radu potrebno je usporediti aplikacije razvijene u dva različita razvojna okvira. Aplikacije trebaju biti napisane pomoću Flutter-a i jednog od nativnih Android rješenja. Potrebno je provesti vlastite i standardizirane testove, a rezultate prikazati u radu.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	05.05.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 18.05.2023.

Ime i prezime studenta:	Matej Kovačević
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1133R, 13.10.2020.
Turnitin podudaranje [%]:	6

Ovom izjavom izjavljujem da je rad pod nazivom**: Usporedba Flutter i nativne Android aplikacije**

izrađen pod vodstvom mentora izv. prof. dr. sc. Mirko Köhler

i sumentora ,

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

Potpis studenta:

SADRŽAJ

1. UVOD 1

2. OPIS TRENUTNOG STANJA U OKRUŽENJU MOBILNIH APLIKACIJA I METODE RAZVOJA 2

2.1. Trenutno stanje u razvoju mobilnih aplikacija 2

2.1.1. Aplikacije razvijene u Flutteru 3

2.1.2. Aplikacije razvijene u Kotlinu 4

2.2. Nativni razvoj 4

2.2.1. Tumačenje platforme 4

2.2.2. Ekosistem 5

2.3. Višeplatformski razvoj 6

2.3.1. Virtualni strojevi 6

2.3.2. Generator koda 6

2.4. Hibridni razvoj 7

3. KONCEPT MOBILNE APLIKACIJE 8

3.1. Sučelje aplikacije 8

3.2. Funkcionalni i nefunkcionalni zahtjevi 9

3.2.1. Funkcionalni zahtjevi 9

3.2.2. Nefunkcionalni zahtjevi 9

3.3. Testiranje performansi nativne i višeplatformske aplikacije 10

3.3.1. Testna okolina 10

3.3.2. Provedeni testovi 12

4. RAZVOJNE TEHNOLOGIJE I PROGRAMSKO RJEŠENJE 14

4.1. Tehnologija, programski jezici i alati 14

4.1.1. Android 14

4.1.2. Flutter 20

4.2. Implementacija u Androidu 25

4.2.1. Arhitektura Android aplikacije 25

4.2.1. Velika lista u Androidu 30

4.2.2. Google Maps i ažuriranje lokacije korisnika 35

4.3. Implementacija u Flutteru 37

4.3.1. Arhitektura Flutter aplikacije 37

4.3.2. Velika lista u Flutteru 37

5. MJERENJA I ANDROID I FLUTTER TEHNOLOGIJE 41

5.1. Mjerenja 41

5.1.1. Veličina aplikacije 41

5.1.2. Vrijeme pokretanja 41

5.1.3. Prikaz liste slika 43

5.2. Analiza rezultata 48

5.3. Analiza korisničkog iskustva i subjektivni doživljaj 49

6. ZAKLJUČAK 51

LITERATURA 52

SAŽETAK 55

ABSTRACT 56

ŽIVOTOPIS 57

1. UVOD

Današnji svijet gotovo je nezamisliv bez mobilnih uređaja i aplikacija koje se mogu preuzeti na *online* trgovinama. Svaka aplikacija ima svoju svrhu i nastoji pružiti korisniku što više funkcionalnosti koje pravilno rade. Osim toga da pravilno rade na trenutno konkurentom tržištu iznimno bitno postaje i da imaju što bolje performanse jer to je na kraju ono što ih razlikuje od ostalih sličnih aplikacija. Brze i responzivne aplikacije imaju veću šansu da će ih korisnik zadržati na uređaju, ali i češće koristiti, nego spore i nefunkcionalne aplikacije. Ovaj rad obrađuje na koji način se te performanse mogu testirati i kako se one za aplikaciju s istim funkcionalnostima razlikuje kada se ona napravi sa dvije vrlo poznate tehnologije za razvoj mobilnih aplikacija, a to je Android native korištenjem *Jetpack Compose* skupine alata i *Flutter*. Na temelju definiranih testnih slučajeva, bit će napravljeni testovi performansi za višeplatformsko i nativno rješenje te će se rezultati analizirati i odrediti koji je potencijalno bolji pristup pri razvoju.

U drugom poglavlju govori se o izazovima tijekom izrade mobilnih aplikacija sa stajališta korisnika, klijenta i programera, prikazano je trenutno stanje mobilnih aplikacija i primjer napravljenih nativnih i višeplatformskih mobilnih aplikacija. Treće poglavlje prikazuje funkcionalnosti aplikacije ovog rada i njenu građu, opisuje testne slučajeve i testno okruženje. U četvrtom poglavlju definirane su korištene tehnologije i prikazani su dijelovi implementacije nativne i višeplatformske aplikacije. U petom poglavlju prikazuju se rezultati testiranja i njihova analiza te subjektivni doživljaj stvaranja aplikacija.

2. OPIS TRENUTNOG STANJA U OKRUŽENJU MOBILNIH APLIKACIJA I METODE RAZVOJA

Prema platformi *Statista*, koja sakuplja i obrađuje korisničke podatke diljem svijeta broj instaliranih aplikacija ove godine bit će oko 299 milijardi, a to je veliki rast u usporedbi na otprilike 275 milijardi u prethodnoj godini. Broj objavljenih aplikacija i mobilnih igricu u 2021. bio je skoro 22 milijuna, a taj broj je danas puno veći. Zanimljiva činjenica je da prosječno vrijeme provedeno na mobilnom uređaju u 2021., na analiziranim tržištima, je bilo 4 sata i 48 minuta dnevno. Za usporedbu se može uzeti prosječno gledanje televizije u Americi koji iznosi otprilike 3 sata. Temeljem navedenog može se zaključiti da mobilni uređaji i aplikacije igraju sve važniju ulogu u životima ljudi i već su postali glavni izvor zabave, ali i poslovnih informacija i drugih oblika interakcije. Bitno je napomenuti da se velika većina pozornosti danas usmjerava na mobilne uređaje te će i dalje biti snažni rast ovo tržišta, pa je ključno adaptirati poslovne ciljeve u skladu s tim trendom [1].

2.1. Trenutno stanje u razvoju mobilnih aplikacija

Bez obzira na industrijsko područje kojom se ljudi bave, način kreiranja mobilne aplikacije značajno utječe na uspjeh mobilne aplikacije. U spomenutom velikom broju aplikacija dobar plan i odabir najboljeg načina razvoja odlučuje hoće li aplikacija biti primijećena ili će se izgubiti u zaboravu kao puno drugih svake godine. Osim toga, mobilne aplikacije mogu služiti za donošenje vrlo bitnih poslovnih odluka, stoga je vrlo bitno da se procesu izrade i realiziranju ideje pristupi na najbolji način. Prilikom planiranja izrade mobilne aplikacije treba odlučiti koji pristup će se koristiti u izradi – nativni, hibridni ili križni. U nastavku ovog rada ovi načini bit će detaljnije opisani i oprimjereni. Brojni se problemi i rizici mogu pojaviti tijekom same izrade, ali i planiranja implementacije mobilne aplikacije zbog kompleksnih zahtjeva, . Klijentu, a poslije i krajnjem korisniku najbitnije je da aplikacija sadrži glavne funkcionalnosti, a po mogućnosti i dodatnu vrijednost. Iako su zahtjevi i problemi koje aplikacija rješava najbitniji cilj, korisnici isto tako hoće imati izgled aplikacije ugodan oku sa modernim animacijama i vrhunskim korisničkim iskustvom što podrazumijeva i intuitivnu interakciju s aplikacijom. Ono što definira ove značajke su performanse aplikacije koje se očituju u tome da je aplikacija brza i pri težim logičkim i matematičkim operacijama. Korisniku zapravo i nije bitna metoda i tehnološka pozadina kako je aplikacija napravljena nego korisničko iskustvo i stvarni problem koji je ona u mogućnosti riješiti [2]. Sa sve složenijim problemima, a time i korisničkim zahtjevima sve je teže ispuniti kriterije i funkcionalnosti kako bi se zadovoljio korisnik.

U kontekstu mobilnih aplikacija, programer je glavni član tehničkih timova koji su odgovorni za ispunjenje korisničkih zahtjeva kroz implementaciju i ažuriranje softvera. On u suradnji s ostalim softverskim inženjerima, arhitektima rješenja, analitičarima, dizajnerima i voditeljima projektima pokušava stvoriti proizvod i sklopiti ga u smislenu cjelinu koja će odgovarati potrebama korisnika [3]. Osim što mora biti stručan u tehnologiji kojom se služi za implementaciju, mora imati i izvrsne komunikacijske vještine kojima će ispravno komunicirati buduće korake u projektu.

Programer mora imati dubinsko poznavanje načela kodiranja, dizajniranja arhitekture aplikacije i jasan raspored kako će razviti određene module. Ovisno o edukaciji, programeri aplikacija moraju znati izabrati programski jezik i alat kojim će programirati ovisno o zahtjevima i operativnom sustavu za koji razvijaju [3]. Prodajni odjeli tvrtki pokušavaju prodati što isplativiju opciju vanjskim klijentima, pa se zato vrlo često u slučaju mobilnih aplikacija odlučuju za višeplatformska rješenja koja su puno jeftinija, vremenski i resursno manje zahtjevna opcija za vlasnike poduzeća [4]. Iako brojni programeri sve više preferiraju raditi u višeplatformskim tehnologijama, isto tako moraju biti svjesni mogućih prepreka i ograničenja te na vrijeme ih prezentirati interno u timu pa kasnije i transparentno u komunikaciji s klijentom. Ako je aplikacija namijenjena samo za jednu platformu, može se raditi nativna aplikacija u *Kotlinu* za Android ili *Swiftu* za iOS. Ako je potrebno kreirati aplikaciju za obje platforme opcije su ili dvije zasebne nativne aplikacije ili jedna višeplatformska. Za nativni razvoj potrebne su različite implementacije u drugačijim tehnologijama za jednake aplikacije. Unatoč navedenim prednostima višeplatformskih aplikacija postoje i neki nedostaci kao što je prevelika ovisnost o vanjskim bibliotekama, veća veličina aplikacije, lošije performanse u nekim slučajevima i druge. Prilikom odlučivanja za neki od ova dva načina treba uzeti u obzir dostupan proračun, vrijeme potrebno za izradu te performanse koje su tražene.

Hibridni razvoj stvoren je zadnji, kao posljedica brzog razvoja web tehnologija korištenih u preglednicima. Rezultat takvog razvoja je programsko rješenje za pojedinačnu platformu pa se može svrstati i u višeplatformski, ali način na koji funkcionira se razlikuje od drugih tehnologija koje su ubrojene u višeplatformski.

2.1.1. Aplikacije razvijene u Flutteru

Sa 100 milijuna korisnika u brojnim zemlja, *Google Pay* korisnicima diljem svijeta omogućuje plaćanje, uštedu, upravljanje troškovima i još mnogo toga [5]. Da bi te funkcionalnosti bile moguće napisano je 1,7 milijuna linija koda unutar Android i iOS aplikacija. Velika količina koda teško se održavala jer se *Google Pay* nastavio proširivati u nove zemlje, od kojih svaka zahtjeva

svoje posebne značajke za aplikaciju. Tako su odlučili, unatoč brojim izazovima, prepisati aplikaciju u *Flutteru* i prvo ju objaviti u Indiji, pa američkom tržištu a poslije i u drugim zemljama.

Osim *enterprise* aplikacija *Flutter* se koristi i u svijetu mobilnih igara. Jedna od najpoznatijih igara PUBG MOBILE i tvrtka koja ju je proizvela *LightSpeed & Quantum Studio* razvili su aplikaciju koja omogućuje igračima iz cijelog svijeta da dijele isječke igranja i ostalu interakciju među igračima. Timu se *Flutter* činio kao najbolja opcija zbog brzine razvoja, odličnih performansi i mogućnošću korištenja nativnih plugina u *Dart* kodu.

2.1.2. Aplikacije razvijene u Kotlinu

Jedna od najkorištenijih aplikacija za razmjenu poruka *WhatsApp* razvijena je kao nativna aplikacija za iOS i Android mobilne uređaje. Primjetno je da je ova platforma bila četvrta najpreuzimanija aplikacija na globalnoj razini u trećem tromjesečju 2020., s oko 140 milijuna preuzimanja [6]. Razlog zašto je ovakvu vrstu aplikacije potrebno pisati u nativnom *frameworku* leži u tome da sadrži brojne kompleksne funkcionalnosti kojima se lakše pristupa putem nativnog koda, nego preko biblioteka koje često nisu dovoljno ažurirane. Primjer takvih funkcionalnosti su video i obični pozivi, slanje glasovnih poruka sigurnost i enkripcija pri razmjeni poruka, zahtjevna funkcionalnost pretraživanja i slično. Aplikacija razmjene poruka između običnih korisnika koristi se i u poslovnom svijetu kao kanal komunikacije.

Aplikacija *Spotify* je pravi primjer nativne aplikacije. Pruža digitalnu glazbenu uslugu s tisućama pjesama i *podcast* produkcijskih kuća i medija diljem svijeta. Osim što je pisana nativno, također je jedna od najznačajnijih aplikacija u oblaku. Koristeći se *Google Cloud*-om, *Spotify* je povećao učinkovitost performansi u velikim radnim opterećenjima kako bi pružila nezaboravno iskustvo reproduciranja glazbe svojim korisnicima.

2.2. Nativni razvoj

Nativni razvoje predstavlja izradu mobilne aplikacije prvenstveno za jednu platformu. Aplikacija se razvija koristeći programske jezike i alate koji su specifični tu platformu [7].

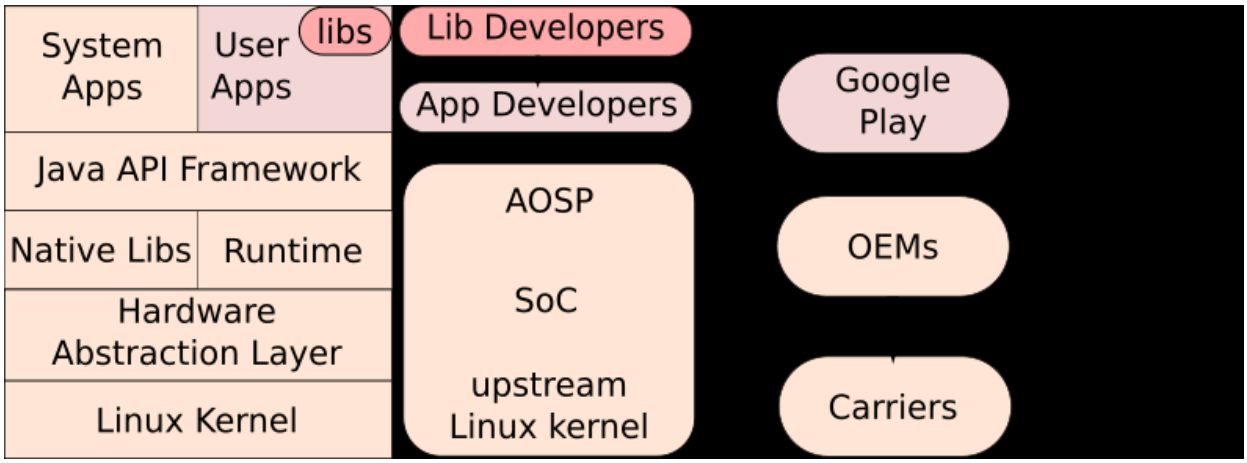
2.2.1. Tumačenje platforme

U IT svijetu, platforma je bilo koji softver ili hardver koji se upotrebljava za udomljavanje usluge ili aplikacije. Platforma aplikacije, primjerice, sastoji se od operacijskog sustava, hardvera, i koordinirajućih programa koji koriste grupu instrukcija za specifični mikroprocesor ili procesor. Platforma stvara temelj koji pomaže uspješno pokretanje objektnog koda. Također, aplikacije mogu igrati dio u opisu platforme. Iako aplikacija može tražiti glavni računalni sustav, kao što je

specifični operacijski sustav i hardver za pohranu ili poslužitelj, aplikacija se može smatrati platformom ako se rabi kao alat za rješavanje smislenog posla [8]. Primjerice, SQL je aplikacija baze podataka, ali se nerijetko upotrebljava kao modul u drugim funkcijama, kao što su, analitika, bilježenje, ERP i CRM sustavi. Tako da se SQL može smatrati platformom.

2.2.2. Ekosistem

Softverski ekosustav je skup softverskih alata, biblioteka i aplikacija koje se koriste za određenu svrhu. Primjeri uključuju baze podataka, operativne sustave, programske jezike i softverske okvire [9]. Naziv potječe iz ekologije i referira se na okoliš u kojem specifična vrsta preživljava i uspijeva. Softverski ekosustav je jako sličan: stvoren je od svih komponenata koje su nužne za pravilan rad aplikacije — uključujući hardver, operativni sustav i drugi softver potreban za njezino pokretanje. U kontekstu mobilnih aplikacija, često pokriva i *deployment* sustav, trgovinu gdje se aplikacija objavljuje i ostali elemente koji čine sustav, a dio su procesa razvoja i objavljivanja aplikacije da bude dostupna na mobilnim uređajima. *Deployment* faza je zadnja u životnom ciklusu razvoja aplikacije jer se s njom stavlja program u produkciju: Nakon planiranja, dizajniranja, implementacije i testiranja produkt se objavljjuje da bude dostupan široj javnosti. Neki od poznatijih *deployment* alata za mobilne aplikacije su *Bitrise, Jenkins, GitHub CI, GitLab CI* i ostali. Android aplikacije se objavljuju na online trgovini *Google Play Store*, a iOS na *Appstore*-u. Ekosustav ne obuhvaća samo proces implementacije, nego sve do korištenja od strane korisnika, pa čak i marketinšku stranu aplikacije. Na slici 2.1 prikazan je Android ekosustav u širem smislu te riječi.



Slika 2.1 Android ekosustav

2.3. Višeplatformski razvoj

Multiplatformski mobilni razvoj je način koji omogućuje pravljenje jedne mobilne aplikacije koja točno radi na nekoliko operativnih sustava. U višeplatformskim aplikacijama, dio ili čak kompletni izvorni kod moguće je podijeliti. To znači da programeri mogu razviti komponente mobilne aplikacije koje rade na Androidu i na iOS-u bez potrebe da ih nanovo programiraju za svaku pojedinu platformu. Iako krajnji rezultat bude jednak, postoji razlika između pristupa koji se nazivaju višeplatformski.

2.3.1. Virtualni strojevi

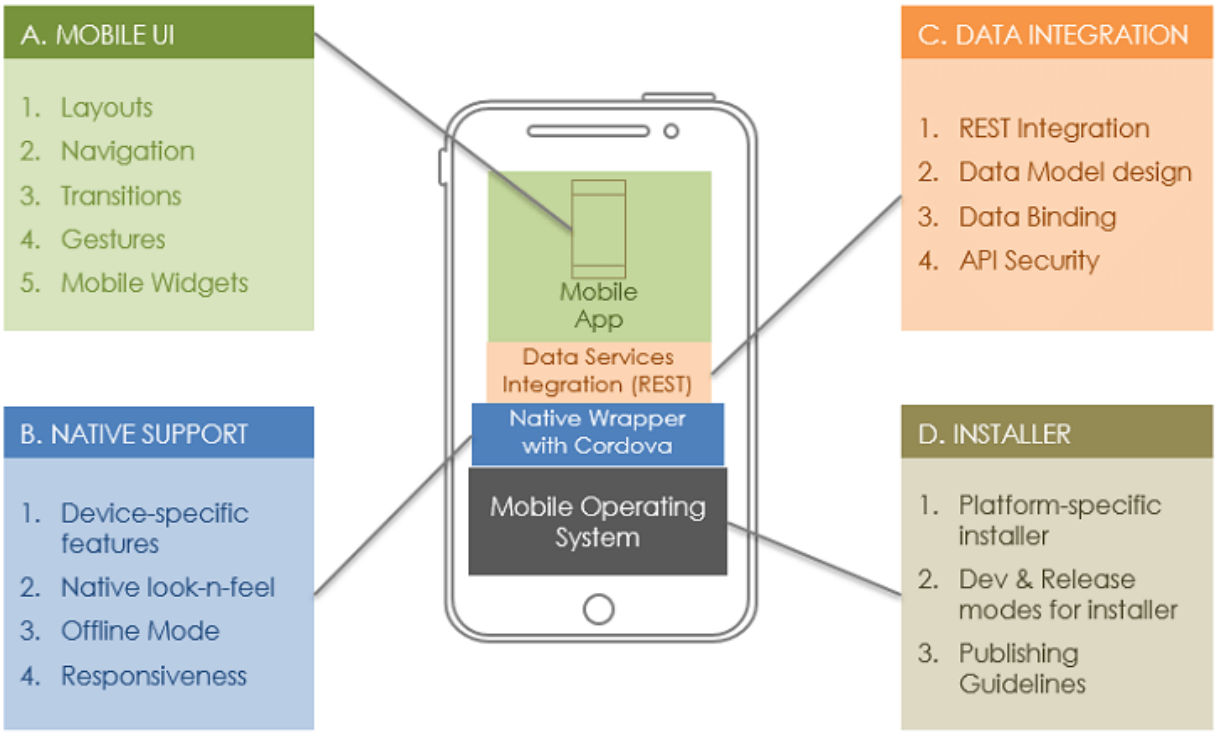
Višeplatformski razvoj funkcionira na način da kompajlira aplikaciju u kod koji može biti pokretan na više platformi. Za te potrebe koriste se virtualni strojevi. Jedan takav je i J*ava Virtual Machine* (JVM) koji pruža *runtime* okoliš za pokretanje *Java* koda ili aplikacija [10]. Transformira Java bajt kod u strojni jezik. U nekim drugim programskim jezicima, prevoditelj proizvodi strojni kod za određeni sustav na kojem se onda može pokretati, ali *Java* prevoditelj proizvodi kod za virtualni stroj poznat kao *Java* virtualni stroj. Brojni programski jezici prevode se u *Java* bajtkod (*Scala , Clojure, Kotlin, Groovy*...), tako da, programska rješenja kreirana pomoću navedenih programskih jezika isto se mogu pokretati u JVM.

2.3.2. Generator koda

Velika prednost multiplatformskog razvoja je to što aplikacije imaju jedinstvenu kodnu bazu (eng. *codebase*). Prioritet ovakvog razvoja je napraviti aplikaciju u jednom programskom jeziku, a pozadinski alati i resursi generiraju nativni kod posebno za svaku platformu. Performanse takvih aplikacija mogu se usporediti s nativnim aplikacijama, a velika je ušteda na količini programskog koda, vremenu, ljudskim i drugim resursima koji bi inače bili potrebniji kada bi se aplikacije razvijale zasebno. Međutim, postoje i neke mane ovog pristupa i nije uvijek tako idealno, a to je da višeplatformska tehnologija ne podržava uvijek sve značajke i funkcionalnosti nativne, te ne može uvijek pratiti sve promjene koje tvrtke objavljuju za svoje nativne tehnologije. Trenutno najkorištenije višeplatformske tehnologije (eng. *cross-platform technology)* su Flutter, *React Native, Kotlin Multiplatform Mobile*. *Flutter* tehnologija se dosta odupire mogućim nedostacima jer je zajednica koja poboljšava ovaj sustav jako razvijena, a i postoji mogućnost da se u izvornom *Dart* kodu koristi kod, odnosno API-iji pojedine platforme.

2.4. Hibridni razvoj

Prilikom pretrage *web* stranica i foruma, može se primijetiti se naizmjenično koriste izrazi "višeplatformski mobilni razvoj" i "hibridni mobilni razvoj", ali to nije sasvim točno. U višeplatformskim aplikacijama, programeri mogu napisati kod jednom i ponovno ga koristiti na različitim platformama. Hibridne aplikacije, s druge strane, kombiniraju izvorne i web tehnologije. U njima je potrebno da se u izvornu aplikaciju piše i kod napisan na programskom jeziku web tehnologija kao što je *JavaScript* , *HTML*, *CSS*. To se može napraviti pomoću okvira, kao što su *Ionic Capacitor* i *Apache Cordova*, koristeći dodatne dodatke za pristup izvornim funkcionalnostima platformi [11]. Jedna od podudarnosti između hibridnog i višeplatformskog razvoja je dijeljenje koda. Međutim, nativne aplikacije imaju puno bolje performanse nego hibridne. Budući da hibridne aplikacije se služe jednom kodnom bazom, neke su funkcionalnosti specifične za određeni operacijski sustav i ne rade dobro na drugima. Na slici 2.2 možete vidjeti na koji način funkcioniraju hibridne aplikacije.



Slika 2.2 Hibridna aplikacija

3. KONCEPT MOBILNE APLIKACIJE

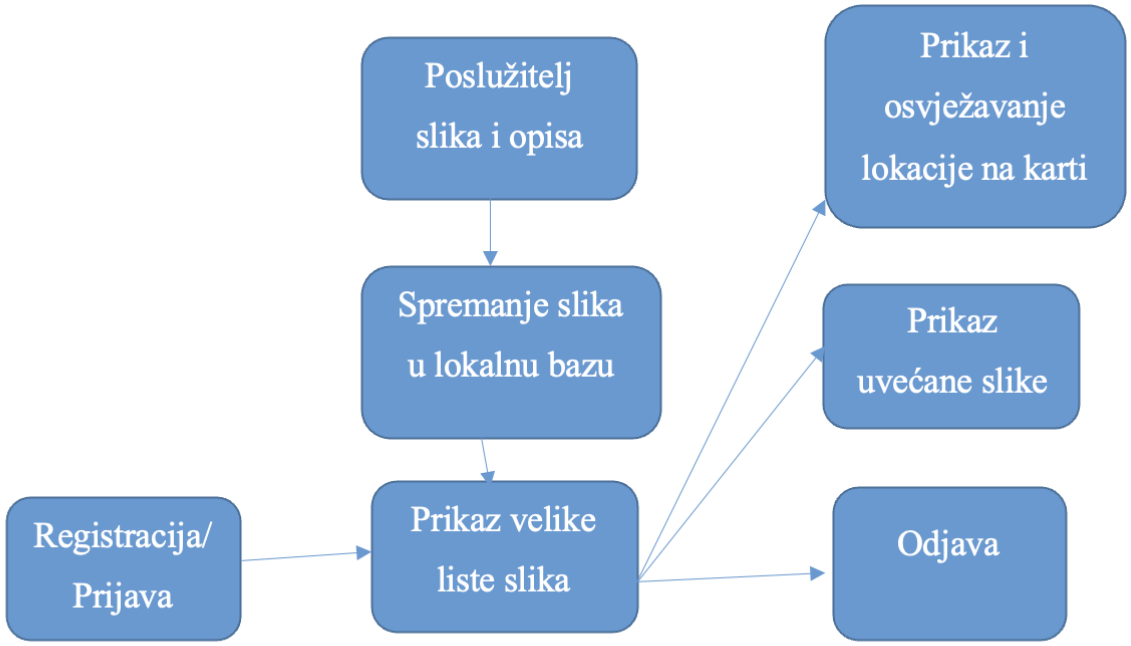
U ovom odjeljku predloženi su prijedlog korisničkog sučelja aplikacije, funkcijski te nefunkcijski zahtjevi programa i pregled alata upotrjebljenih za provođenje testova aplikacije.

3.1. Sučelje aplikacije

Kreirana mobilna aplikacija sastoji se od početnog tzv. *Splash* zaslona koja vodi na zaslon za registraciju u aplikaciju. Osim registracije, korisniku je omogućena i prijava putem zaslona za prijavu čiji su elementi na sučelju gotovo isti kao i na registracijskom zaslonu.

Nakon prijave, korisnika se vodi na zaslon koji prikazuje jako veliku listu slika nasumičnih slika i slučajne opise dohvaćene s poslužitelja https://jsonplaceholder.typicode.com. Klikom na element list korisnik može pogledati uvećanu sliku na novom zaslonu. Slike se spremaju i dohvaćaju iz relacijske lokalne baze podataka.

Osim navedenih zaslona, postoji i zaseban zaslon kojem se pristupa klikom na gornjoj traci aplikacije (eng. *AppBar*). Na otvorenom zaslonu može se vidjeti Google karta koja prikazuje i ažurira trenutnu lokaciju korisnika. Na gornjoj traci zaslona sa listom slika postoji i tipka za odjavu korisnika iz aplikacije koja ga vraća na zaslon za prijavu. Na slici 3.1 može se vidjeti koncept mobilne aplikacije i elementi od kojih se sastoji.



Slika 3.1 Koncept aplikacije

3.2. Funkcionalni i nefunkcionalni zahtjevi

Inženjerska zajednica podijelila je zahtjeve softverskog sustava u dvije glavne kategorije: funkcionalne i nefunkcionalne zahtjeve.

Funkcionalni zahtjevi opisuju funkcionalno ponašanje sustava, dok nefunkcionalni zahtjevi izražavaju kako bi sustav trebao funkcionirati. Poznato je da atribut kvalitete kao što je pouzdanost, mogućnost modifikacije, izvedba ili upotrebljivost je nefunkcionalan zahtjev softverskog sustava [12]. Ključni faktor za uspjeh aplikacije je ispunjavanje nefunkcionalnih zajedno s funkcionalnim zahtjevima.

3.2.1. Funkcionalni zahtjevi

Planiranje funkcionalnih zahtjeva odvija se u fazi projektiranja sustava. Određuje se rad i funkcionalnost softvera zbog koji će se kupiti. Ovi zahtjevi uključuju logiku aplikacije, zahtjeve korisnika, administrativne zadatke, autorizacija korisnika, prava pristupa, zahtjevi za vanjskim resursima i ostalo. Neki od najčešćih funkcionalnih zahtjeva mobilne aplikacije su prijava, registracija i odjava korisnika, dohvaćanje personaliziranih podataka i njihov prikaz u obliku lista, grafova i ostalih načina prikaza podataka, dostupnost podataka *offline* što podrazumijeva spremanje u lokalnu bazu, analitiku korištenja funkcionalnosti aplikacije, navigacija, rukovanje greškama i slično.

Nakon navođenja općenitih primjera funkcionalnih zahtjeva, mogu se navesti primjeri takvih zahtjeva i za aplikaciju razvijenu u ovom radu:

- Korisnik se može registrirati i prijaviti u aplikaciju ako je već registriran
- Korisnik se isto tako može odjaviti iz aplikacije
- Korisnik može pregledavati listu slika, dostupnu i offline
- Korisnik može kliknuti na element u listi kako bi otvorio detalje
- Korisnik može otvoriti kartu na kojoj može pratiti svoju lokaciju

3.2.2. Nefunkcionalni zahtjevi

Iako je važno da aplikacija sadrži sve tražene funkcionalnosti, jednako je važno kako zahtjevi pokreću ostatak razvoja softvera. Konkretno, tijekom faze projektiranja, velik dio aspekata kvalitete sustava je određen. Kvalitete sustava često se smatraju kao nefunkcionalni zahtjevi, koji se nazivaju i atributi kvalitete [13]. To su zahtjevi kao što su upotrebljivost, pouzdanost, cijena,

mogućnost održavanja, vrijeme razvoja i ostali faktori koji su zajedno s funkcionalnostima ključni za uspjeh proizvoda. Mobilne aplikacije takvu vrstu zahtjeva prikazuju u obliku veličine instalirane aplikacije, vremena pokretanja aplikacije, memorijsko opterećenje spremanja podataka u lokalnu memoriju i slično. Za zahtjeve je iznimno bitna skalabilnost, da aplikacija podržava korištenje od strane više korisnika s puno podataka, pouzdanost da funkcionalnosti uvijek daju isti izlaz za iste korake koje napravi korisnik te brzi odziv pri interakciji s korisnikom.

Ako se uzmu obzir spomenute vrste nefunkcionalnih zahtjeva, za aplikaciju kreiranu o ovom radu mogu se navesti sljedeći nefunkcionalni zahtjevi:

- Korisnik treba vidjeti prikaz liste slika nakon prijave unutar trideset sekundi
- Prilikom listanja ne smije se događati vizualno zastajkivanje tzv. „*Jank*“
- Na većini uređaja različitih veličina aplikacija treba prikazivati sve elemente i imati sve funkcionalnosti
- Sustav ne smije ostati bez dovoljno RAM memorije tijekom korištenja
- Aplikacija mora učitati Google kartu unutar par sekundi i prikazati lokaciju korisnika
- Prilikom pritiska na element u listi uvijek se mora otvoriti slika s detaljima
- Vrijeme od pritiska na ikonu aplikacije do njenog otvaranja mora biti manje od 1 sekunde

3.3. Testiranje performansi nativne i višeplatformske aplikacije

Testiranje performansi mobilnih aplikacija je važan proces koji osigurava da aplikacija radi prema očekivanjima i da pruža nesmetano korisničko iskustvo. Neki od koraka koje treba izvršiti pri testiranju performansi su definiranje ciljeva, određivanje testnih slučajeva, odabrati prave alate i okruženje za testiranje, izvršiti testove na pravim uređajima te na kraju i analizirati dobivene rezultate.

3.3.1. Testna okolina

Kako bi se prepoznale i riješili ključni problemi s performansama u aplikaciji potrebno je provjeriti što se događa tijekom određenih radnji pomoću alata za uklanjanje pogrešaka (eng. *Debugging tools*). Android Studio razvojno okruženje nudi nekoliko alata za profiliranje koji pomažu u pronalaženju i vizualiziranju mogućih problema [14]:

- *CPU profiler* pomaže u praćenju problema s izvedbom u vremenu izvršenja.

- *Memory profiler* pomaže u praćenju alokacija memorije.
- *Network profiler* prati upotrebu mrežnog prometa.
- *Energy profiler* prati upotrebu energije koja može doprinijeti pražnjenju baterije.

Navedeni alati poslužit će u testiranju nativne Android aplikacije, a za praćenje performansi Flutter aplikacije iskoristiti će se alati pod nazivom *DevTools* koji imaju sljedeće mogućnosti [15]:

- Pregledavanje rasporeda i stanja korisničkog sučelja *Flutter* aplikacije.
- Dijagnosticiranje problema s izvedbom korisničkog sučelja (UI *jank*) u *Flutter* aplikaciji.
- Profiliranje CPU-a za *Flutter* ili *Dart* aplikaciju.
- Profiliranje mreže za *Flutter* aplikaciju.
- Otklanjanje pogrešaka na razini izvornog koda (*source-level debugging*) *Flutter* ili *Dart* aplikacije.
- Otklanjanje problema s memorijom u naredbenoj liniji *Flutter* ili *Dart* aplikacije.
- Pregledavanje općih zapisnika i dijagnostičke informacije o pokrenutoj naredbenoj liniji *Flutter* ili *Dart* aplikacije.
- Analiziranje koda i veličina aplikacije.

Ovi alati također nude opciju uvoza i izvoza rezultata praćenja performansi. Osim navedenih alata koristi se i cloud servis *Firebase* i njegova usluga *Performance Monitoring* usluga koja pomaže dobiti uvid u karakteristike performansi *Apple*, Android i web aplikacija [16]. Pomoću ovog alata skupljaju se podaci o izvedbi aplikacije, a onda se isti mogu pregledavati i analizirati na *Firebase* konzoli te s obzirom na rezultate poboljšati ili popraviti određene aspekte programa. Potrebno je testirati aplikaciju na što više uređaja kako bi rezultati bili što efektniji. Za potrebe ovog rada aplikacija je testirana na dva prava mobilna uređaja *Samsung Galaxy S8* i *Samsung Note 20 Ultra*, a specifikacije navedenih mogu se vidjeti u tablici 3.1.

Tablica 3.1 Opis uređaja za testiranje

	Galaxy Note 20 Ultra 5G	Samsung Galaxy S8
Android verzija	13	9
Model	SM-N986B	SM-G950F
Procesor	Samsung Exynos M5 @ 2.73 GHz	Samsung Exynos M2 @ 2.31 GHz
Broj jezgri	8	8
RAM	12 GB	4 GB

Rezolucija zaslona	1080 x 2316 px	1080 x 2220 px
Gustoća prikaza	372	425

3.3.2. Provedeni testovi

Sastavljeni su testni slučajevi koji se fokusiraju na najbitnije stavke s obzirom na zahtjeve mobilne aplikacije, ali i na testove koji nisu do sada provedeni.

Veličina aplikacije

Sve aplikacije imaju veličinu u megabajtima koja određuje količinu pohrane koja joj je nužna da bi funkcionalno radila na mobilnim uređajima. Kako je prostor za pohranu dosta ograničen, važno je da je aplikacija razumne veličine.

Prosječna veličina aplikacije razlikuje na dva operacijska sustava, za iOS je oko 35 MB, a za Android oko 12 MB i očito prosječne veličine variraju ovisno o kategoriji aplikacije kojoj su dodijeljene [17]. Malo je vjerojatno da bi korisnici željeli instalirati nešto što se čini kao vrlo velika aplikacija. Korisniku kojem nestaje prostora na uređaju, poželjet će obrisati aplikacije sa svog uređaja, a najveće aplikacije prve su na udaru.

Postotak korištenja procesora

Optimiziranje korištenja CPU-a aplikacije ima mnoge prednosti, poput osiguranja bržeg i glađeg korisničkog iskustva i očuvanja trajanja baterije uređaja. Ako se koristi prevelik postotak procesora, drugi procesi u uređaju mogu biti negativno pogođeni. CPU će se mjeriti kroz cijelu aplikaciju, ali posebnu pozornost obraćat će se na neke specifične radnje definirane u testnim slučajevima.

Mrežni promet

Obavlja se vremensko mjerenje mrežnog prometa s obzirom da aplikacija dohvaća veliku količinu podataka odjednom s poslužitelja. Obavljalo se i mjerenje CPU i alokacija memorije, ali ti rezultati ovise i o trenutnim uvjetima mreže i poslužitelja.

Količina korištene memorije

Korišteni RAM tijekom interakcije s aplikacijom je iznimno bitan jer gubitak i curenje memorije mogu dovesti do smrzavanja zaslona pa čak i rušenja. Mjerenje količine iskorištene memorije radi se na početku i na kraju, ali i tijekom samog korištenja unutar aplikacije. Pomoću dostupnih

grafikona omogućuje se praćenje dodjele memorije. Uređaji nižih performansi posebno su osjetljivi s obzirom na manju količinu RAM memorije koju posjeduju. Posebno je bitno obratiti pozornost na količinu korištene memorije pri učitavanju karte i traženju lokacije korisnika.

Vremena odziva

Prema provedenim ispitivanjima korisnici su spremni čekati da aplikacija odgovori na korisnički zahtjev prema sljedećim parametrima: polovica sudionika (45%) odabrala je vrijeme odgovora od 3 do 5 sekundi, dok je druga četvrtina ispitanika spremna čekati dulje od 5 sekundi. 13% ispitanika bi čekali samo 1 ili 2 sekunde za odgovor [18]. Koliko su korisnici spremni čekati na određene zadatke ovisi od slučaja i funkcionalnosti koju koriste, ali svakako je potrebno uzeti u obzir ovu procjenu tog vremena jer svo duže vrijeme koje je korisnik morao čekati uvelike narušava iskustvo korištenja aplikacije. Provedena su mjerenja pokretanja aplikacije te vrijeme dohvaćanja podataka s poslužitelja, spremanje i dohvaćanje iz lokalne baze.

Kodiranje i razvoj

Osim opipljivih mjerenja provedenih korištenjem raznih alata, dat će se subjektivni doživljaj u razvoju aplikacija kao što su primjerice koliko kompleksno je bilo implementirati pojedine funkcionalnosti, koliko koda je potrebno te kakav je sveukupni doživljaj tijekom razvoja pojedine aplikacije.

4. RAZVOJNE TEHNOLOGIJE I PROGRAMSKO RJEŠENJE

U ovom poglavlju predstavljene su tehnologije i programski jezici korišteni za nativni i višeplatformski razvoj potrebni za razvoj aplikacije. Prije prikaza same implementacije bitno je upoznati se sa teorijskom pozadinom uporabljenih tehnologija i alata koji su se koristili u testovima. Nakon opisa tehnologija prikazana su rješenja u svakoj od platformi.

4.1. Tehnologija, programski jezici i alati

4.1.1. Android

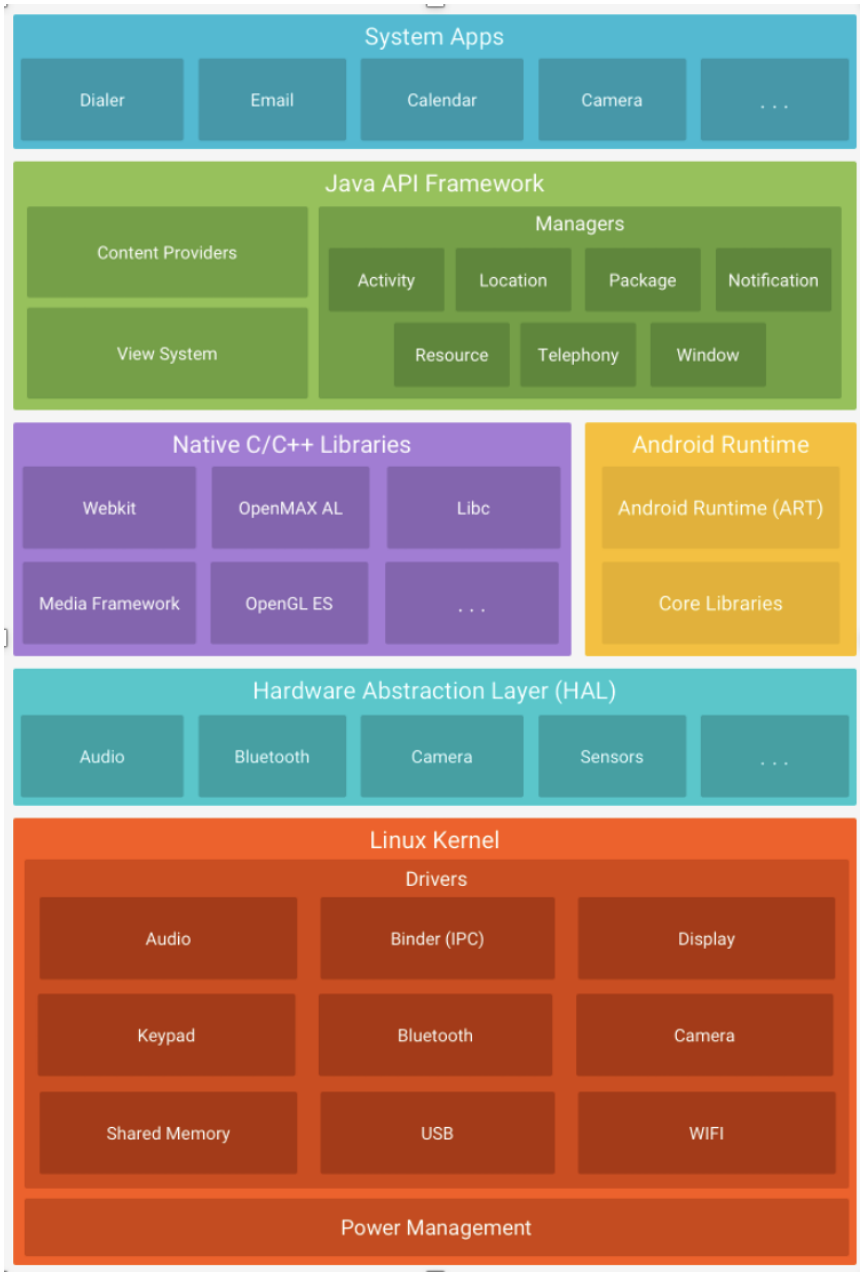
Mjesto najdominantnijeg operacijskog sustava za mobilne uređaje još uvijek zauzima Android. Njegova najveća konkurencija *Apple*-ov iOS koji zauzima oko 29% tržišta u trenutnoj godini(*StatCounter,* 2023). Android je i dalje najpopularniji operacijski sustav s 71% udjela u svjetskom tržištu mobilnih uređaja [19].

Android je operacijski sustav baziran na *Linux* jezgri, razvijen pretežno za uređaje sa pametnim zaslonima osjetljivim na dodir, poput tablet računala i pametnih mobilnih uređaja. Naknadno operacijski sustav prilagođen je i za pametne televizore, za automobilska sredstva te za pametne satove. Za svaki od tih tipova uređaja sustav je prilagođen jedinstvenom korisničkom sučelju. Android je operacijski sustav stvoren 2003. godine u Kaliforniji, razvijen od strane tvrtke *Android Inc*., međutim, 2005. godine tvrtku počinje posjedovati Google. Prvi put ova je platforma predstavljena široj javnosti 2007. godine, a nakon toga u devetom mjesecu 2008. godine kreiraju se prvi uređaji utemeljeni na Androidu. Od prvog dana Android je zamišljen kao programska podrška otvorenog koda pa je tako kompletni kod dostupan pod Apache licencom od listopada 2008. godine kad su razvijeni prvi uređaji. Google kreira paket softvera pod svojim vlasništvom koji je povezan s Androidom, a obuhvaća programe za razne usluge, na primjer, trgovina aplikacijama *Google Play* te Google karte ili *Gmail,*. Početkom 2011. godine postaje najprodavaniji mobilni operacijski sustav za pametne telefone, a kasnije i za tablete. 2 milijarde korisnika mjesečno aktivno koristi ovaj operacijski sustav, a više od 3,5 milijuna aplikacija trenutno se nalazi u Google Play trgovini [20]. Zbog svog otvorenog izvornog koda Android omogućava brojne pogodnosti kao što je povezivanje s ostalim uređajima koristeći razne komunikacijske komponente (primjerice, LTE, Bluetooth, UMTS i ostalo), internetski preglednik, podrška za povezivanje vanjskih uređaja za upravljanje (tipkovnica, računalni miš itd.), SMS i MMS servise za primanje i slanje poruka, bežično povezivanje na internetsku mrežu, spajanje na Google T V i kontroliranje svih funkcija i mnoge druge. Android aplikacije su najvećim

dijelom pisane u *Java* ili *Kotlin* programskom jeziku, ali od konferencije Google I/O 2019 glavni programski jezik za razvoj mobilnih aplikacija na Androidu postaje *Kotlin* [21] koji u kombinaciji sa *Android Software Development Kit* (SDK) omogućava kreiranje mobilnih aplikacija.

4.1.1.1. Arhitektura

Android je softver baziran na modificiranim verzijama Linux *kernela* i drugih programa otvorenog koda. Ono što se pojavljuje na površinskoj razini na operacijskom sustavu, je samo prividna jednostavnost jer postoji više slojeva koji izgrađuju Android operacijski sustav. Slojevi se neprestano usavršavaju zahvaljujući otvorenom kodu i ulaganjem brojnih tvrtki [22]. Za razvoj uspješnih aplikacija potrebno je razumijevanje cjelokupne arhitekture Androida koja se sastoji od segmentiranih slojeva koji zajedno funkcioniraju kao cjelina. Zahvaljujući povezanoj arhitekturi i slojevima na niskoj razini, programeri se mogu usredotočiti na poslovne zadatke koristeći tu strukturu, a ne moraju brinuti o apstrakciji koja je u pozadini. Iako slojevi međusobno skladno funkcioniraju, njihova implementacija je neovisna i zato je moguće po potrebi primijeniti ažuriranja na svaki sloj zasebno i tako ispraviti moguće greške ili uvesti poboljšanja.



Slika 4.1 Android arhitektura

Prema slici 4.1 imamo 5 skrivenih slojeva android arhitekture.

Na najnižoj razini nalazi se Linux *kernel*, koji rukuje svim upravljačkim programima koji su potrebni tijekom pokretanja na Android uređajima, a to su upravljački programi za zaslon, programi za kameru, audio programi, USB upravljački programi i memorijski upravljački programi i ostali. Na vrhu Linux *kernela* su neke izvorne biblioteke koje uređaju daju upute za rukovanje različitim tipovima podataka na odgovarajući način. To su neke biblioteke temeljene na *Javi*, kao i osnovne biblioteke C/C++ i one koje omogućuju Android razvoj, kao što su *Libc*, *Graphics*, SSL (*Secure Socket Layer*), *Media*, *OpenGL* (*Open Graphic Library*) i ostali. *Android Runtime Environment* gradivni je dio Androida. Sadrži module kao što su temeljne biblioteke i

DVM (*Dalvik Virtual Machine*). Biblioteke, zajedno sa Android *runtime*-om osnova je okvira i pokreće aplikacije. DVM se koristi za optimiziranje upotrebe memorije, trajanja baterije, i performansi. *Android Application Framework* daje sučelja, klase, i uslužne programe koji se upotrebljavaju za razvoj Android aplikacija. U ovaj je okvir uključen i *Android Hardware Abstraction Layer* (HAL), koji aplikaciji daje mogućnost komunikacije s upravljačkim programima specifičnim za hardver te upravljanje resursima i korisničkim sučeljem. Aplikacije su vršni sloj na Android arhitekturi od kojih neke dolaze već instalirane na uređaju, kao što su kamere, glazba, galerije, a neke su preuzete iz trgovine *Google Play Store*-a, primjerice društvene igre, profesionalne aplikacije itd. Ako se želi postaviti prave temelje za razvoj na Android platformi, prvi korak bi trebao biti skupljanje osnovnog razumijevanja njegove osnovne arhitekture.

4.1.1.2. Android Studio

Android Studio službeno je integrirano razvojno okruženje (IDE) za razvoj Android aplikacija [23]. Zahvaljujući brojnim alatima iz *IntelliJ* IDEA, *Android Studio* nudi mnoge značajke koje povećavaju efikasnost pri razvoju Android aplikacija, kao što su:

- Sustav kreiranja temeljen na *Gradle*-u
- Brz emulator s mnogim funkcionalnostima
- Jedinstveno okruženje
- Učitavanje promjena na aplikaciji bez ponovnog pokretanja
- Sveobuhvatni alati i *framework* za testiranje
- *Lint* alati za hvatanje upotrebljivosti, performansi, kompatibilnosti verzija
- NDK i C++ podrška
- Olakšana integracija sa *Google Cloud Messaging* i *App Engine* zbog ugrađene podrške za *Google Cloud Platform*

Alati ovog razvojnog okruženja koji su značajno pomogli u analizi i testiranju aplikacije su *Debug and profile tools*. S njima je moguće otkloniti pogreške i poboljšati izvedbu koda, pratiti korištenje CPU i grafičke memorije te ostala obilježja performansi.

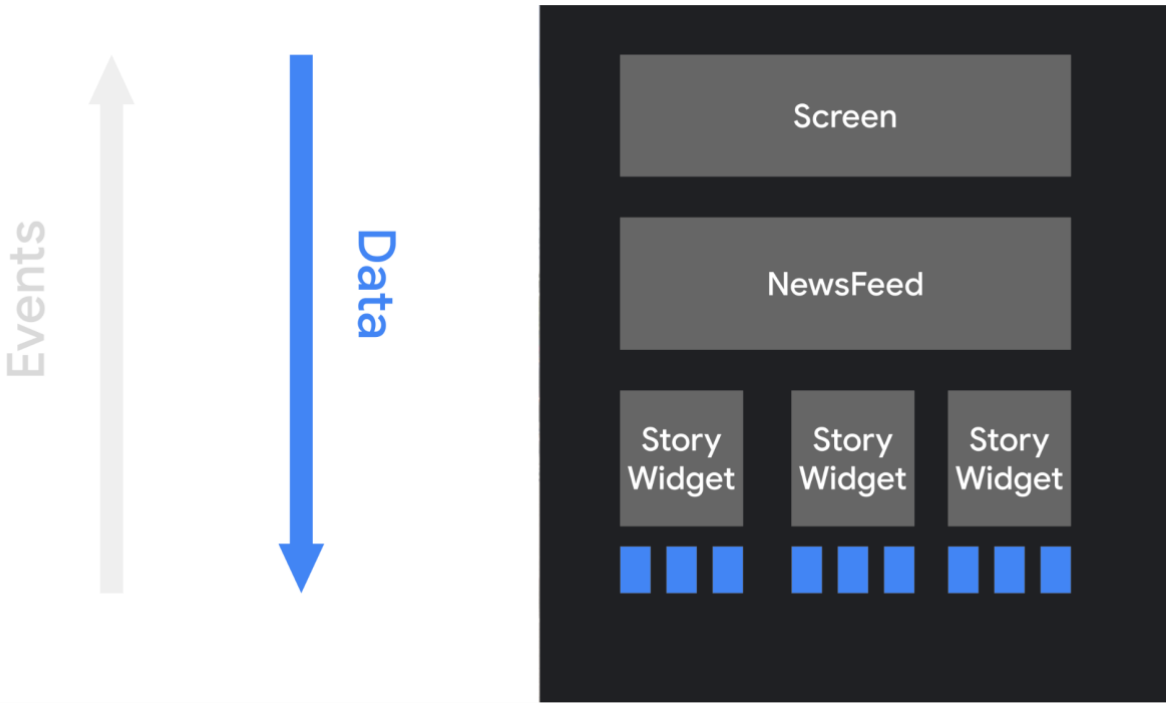
4.1.1.3. Kotlin programski jezik

Programski jezik u potpunosti kompatibilan s Android operacijskim sustavom. Razvijen od strane *JetBrains* tvrtke i od 2019. je službeno glavni programski jezik za razvoj Android aplikacija. Iako su još uvijek neke aplikacije pisane u *Javi* većina ih se prepisuje u Kotlin te i sami platformski programeri koji rade u *Google*-u nastoje pisati nove biblioteke u Kotlinu. Kotlin pruža brojne

prednosti, a neki od njih su: manje koda i veća čitljivost, abrojne alate i u potpunosti je integrirano u *Android Studio*, pruža podršku Android *Jetpack* skupini biblioteka, dodatne značajke i biblioteke kao što su korutine, lambda funkcije i imenovani parametri i ostale moćne funkcionalnosti. Isto tako može se koristiti u kombinaciji s *Javom* bez prepisivanja koda, *Kotlin* multiplatform omogućuje korištenje poslovne logike u iOS aplikacijama i može se također razvijati i web aplikacije. *Kotlin* ima veliku podršku koja svakim danom raste diljem svijeta. Prema Googleu, više od 60% od 1000 najboljih aplikacija u trgovini Play koristi *Kotlin* [21].

4.1.1.4. Stanje korisničkog sučelja

Jetpack Compose je moderan deklarativni UI alat za Android. *Compose* olakšava pisanje i održavanje korisničkog sučelja aplikacije korištenjem deklarativnog API-ja koji omogućuje generiranje korisničkog sučelja aplikacije bez imperativne izmjene prikaza sučelja [24]. U ovom radu umjesto starog načina definiranja izgleda zaslona u XML jeziku za označavanje podatka je korišten novi UI skup alata *Compose*. Posljednjih godina cijela se IT industrija koja se bavi *frontend* tehnologijama počela premještati na deklarativni tip korisničkog sučelja, a to doprinosi jednostavnosti za inženjering povezan s kreiranjem i ažuriranjem korisničkih sučelja. Deklarativni način renderiranja funkcionira tako da se konceptualno generira kompletni zaslon od nule, a potom se primjenjuju samo nužne promjene. Tim pristupom izbjegava se kompleksnost ručnog ažuriranja hijerarhije prikaza sa stanjem, kao što je to slučaj kada se UI definira imperativno u XML gdje svaki element ima *get* i *set* metode za interakciju. Regeneracijom cijelog zaslona je veliki izazov jer to možebitno skupo, u smislu računalne snage, vremena, i trošenja baterije. Međutim, kako bi se smanjio takav trošak, *Compose* inteligentno bira koja područja korisničkog sučelja, ovisno o tome ako im se stanje promijenilo i treba li ih ponovno kreirati u bilo kojem trenutku što se naziva rekompozicija. U *Composeu* UI elementi (*widgeti*) su bez stanja i nemaju funkcije *get* i *set* za interakciju jer zapravo nisu objekti. Oni se ažuriraju tako što se poziva ista funkcija koja ih može sastaviti, ali s različitim argumentima. To olakšava pružanje stanja modulima arhitekture kao što je *ViewModel*. Zatim, *composables* su odgovorni za transformaciju trenutnog stanja aplikacije u UI svaki put kada se vidljivi podaci ažuriraju. Zatim su *composable widgeti* odgovorni za transformiranje trenutnog stanja aplikacije u zaslon i elemente na njemu.



Slika 4.2 UI sloj

Kada korisnik ima interakciju s korisničkim sučeljem, događaji se pokreću sa akcijama kao što je pritisak na gumb *onClick*. Ti događaji pokreću poslovnu logiku aplikacije, koja potom može mijenjati stanje aplikacije. Kada se stanje promijeni, funkcije koje stvaraju *widgete* mogu se sastaviti ponovno i pozivaju se s novim podacima. Primjer pozivanja funkcije koja mijenja stanje *widgeta* može se vidjeti na slici. Framework poziva rekompoziciju samo na funkcijama kojima se stanje, odnosno u ovom slučaju varijabla *name* promijenila jer regeneriranje cijelog korisničkog sučelja je relativno skup proces i utječe na performanse. Bitno je razdvojiti poslovnu logiku od UI-a i izdvojiti logiku na pozadinsku nit te rezultat u obliku stanja proslijediti na UI sloj kako aplikacija ne bi počela zastajkivati.

```
@Composable
fun Greeting(names: List<String>) {
    for (name in names) {
        Text("Hello $name")
    }
}
```

Slika 4.3 Prikaz *Jetpack Compose* funkcije

Compose ima sljedeća obilježja za koja programeri trebaju biti svjesni kada ga koriste [24]:

- *Compose* funkcije izvršavaju se bilo kojim redoslijedom – ako prepozna da neki UI elementi imaju viši prioritet izgradit će prvo njih
- *Compose* funkcije mogu se pokretati i paralelno – uzimaju prednost višejezgrenosti
- Rekompozicija preskače što je više moguće compose funkcija i lambda funkcija – regenerira samo elemente koji se moraju ažurirati s trenutnim stanjem
- Rekompozicija se može otkazati jer je optimistična – ako se rekompozicija nije dovršila, a stanje se već opet promijenilo ona se može otkazati i započeti rekompoziciju s novim parametrima
- *Compose* funkcija može se izvoditi često, jednako često kao i svaki kadar animacije – ako se dohvaćaju neki podaci tu funkciju treba premjestiti na pozadinsku nit u poslovnu logiku tj. *ViewModel* u MVVM arhitekturi kako ne bi došlo do zastajkivanja zaslona jer se odvija prečesto

4.1.2. Flutter

Flutter je razvojni okvir objavljen u svibnju 2017. godine od strane Google-a za izradu aplikacija za Android, iOS, macOS, Linux, Windows i web [25] i od tada *Flutter* zajednica iznimno brzo raste. *Flutter* je alat za više platformi koji je dizajniran da omogući upotrebu izvornog koda u operacijskim sustavima kao što su Android i iOS, a u isto vrijeme omogućuje aplikacijama API sučelje s usluga nativne platforme. Razvojni programeri pomoću ove tehnologije mogu isporučiti aplikacije visokih performansi koje izgledaju i koriste se prirodno na različitim platformama uz minimalne razlike. Prilikom razvoja, *Flutter* aplikacije se izvode u VM-u koji nudi hot reload, odnosno ponovno učitavanje promjena bez ponovnog kompajliranja. Za objavljivanje, aplikacije se kompajliraju direktno u strojni kod, bilo da su to ARM ili Intel x64 procesori, te u *JavaScript* ako se fokusira na web. *Framework* ima dopuštenu BSD licencu, što znači da je otvorenog koda, i ima uspješan doprinos trećih strana koji nadopunjuju korištenje glavnih biblioteka.

Flutter se temelji na četiri glavne komponente:

- programski jezik *Dart* i platforma
- Sloj najniže razine koji implementira crtanje *widgeta* na zaslon i komunikacija s API-jima platforme, primjerice rad s datotekama, rad s mrežom,
- Bibiloteke pisane u *Dartu* za komunikaciju sa slojem nižih razina

- Osnovni *widgeti* pruženi od strane platforme koji su glavne komponente korisničkog sučelja

Podsustav koji omogućuje iscrtavanje grafike naziva se *Skia* i on u teoriji može iscrtavati 3D modele, ali je *Flutter* pretežno korišten za izradu 2D aplikacija. *Skia* upravlja grafičkom karticom (GPU) i ovisno o naredbama iz višeg sloja komunicira sa grafičkom karticom šta treba prikazati na korisničkom sučelju.

4.1.2.1. Dart jezik

Dart je jezik namijenjen za razvoj brzih aplikacija na bilo kojoj platformi [26]. Dizajniran je za tehničku uvjete i posebno je prilagođen razvoju klijenta, pri čemu daje prioritet razvoju (hot reload moguć ispod sekunde) i omogućuje proizvodnju visoke kvalitete u velikom rasponu kompilacija (web, mobilni i desktop). Ovaj jezik također je preduvjet za programiranje u *Flutter* tehnologiji. Podržava većinu osnovnih zadataka razvojnog programera kao što su analiza, formatiranje i testiranje koda. Dart jezik je *typesafe* što znači da koristi statičku provjeru tipa kako bi osigurao da vrijednost varijable uvijek odgovara statičkom tipu varijable. *Dart* je dosta fleksibilan, pa dopušta uporabu dinamičkog tipa u kombinaciji s provjerama vremena izvođenja, a to može biti korisno tijekom eksperimentiranja ili varijable za koje nismo sigurni kojeg su tipa pa mogu biti dinamičke. Jezik je i *null safety*, isto kao i *Kotlin*, što znači da vrijednosti ne smiju biti *Null* osim ako izričito programer to ne naglasi u kodu. Zaštita od *Null* vrijednosti provedena je prije samog pokretanja kroz statičku analizu koda. *Dart* VM omogućava prevođenje u trenutku izvođenja (JIT) s inkrementalnom rekompozicijom (hot reload), praćenje uživo skupa mjernih podataka (pokrećući *DevTools*) i široku podršku za otklanjanje grešaka. *DevTools* i njegovi alati su temeljni elementi korišteni u ovom radu za mjerenje performansi aplikacije i usporedbu s nativnom Android aplikacijom. Kompajler *Dart ahead-of-time* (AOT) može kompilirati u izvorni x64 ili ARM strojni kod kada je aplikacija spremna za objavu u trgovini ili se implementira u produkcijsku pozadinu. *Dart Web* omogućuje izvođenje *Dart* koda na webu kompiliranjem *Dart* koda u *JavaScript* kod, koji se zatim pokreće u pregledniku.

4.1.2.2. Arhitektura

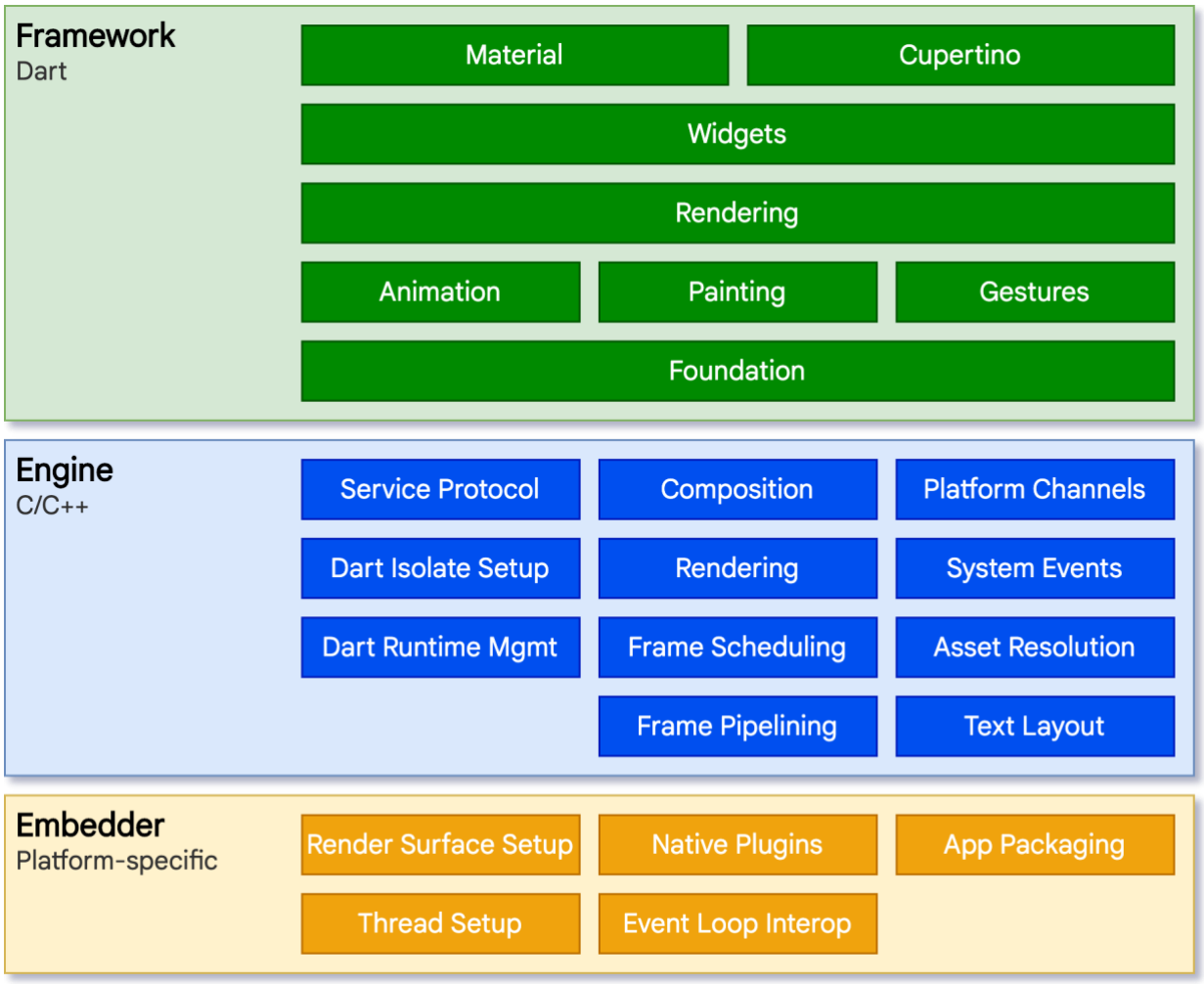
Flutter je dizajniran kao nadogradiv sustav sa puno slojeva. Postoji kao niz nezavisnih biblioteka od kojih svaka ovisi o prvom sloju. Svi slojevi su kreirani da mogu biti promijenjeni i nisu usko povezani s ostalim slojevima [27].

Kao i bilo koja druga izvorna aplikacija Flutter ima *Embedder* specifičan za platformu i pruža ulaznu točku; rukuje s temeljnim operacijskim sustavom za pristup uslugama kao što su prikazivanje korisničkog sučelja, unos i pristupačnost; i upravlja razmjenjivanju poruka. *Embedder* je napisan na jeziku koji odgovara pojedinoj platformi: *Objective-C*/*Objective-C*++ za iOS i macOS, Java i C++ za Android, i C++ za Windows i Linux.

Core je *engine*, napisan pretežno u C++ i podržava varijable potrebne za podršku svim *Flutter* aplikacijama. Omogućuje iscrtavanje novih zaslona kada je to potrebno. Omogućuje implementaciju *Flutter*-ovog primarnog API-ja niske razine, uključujući grafiku (putem *Skia*), mrežni ulaz i izlaz, raspored teksta, arhitekturu dodataka, podršku za pristupačnost, i Dart runtime i skup alata za kompajliranje.

Kroz *dart:ui* biblioteku, *Engine* je izložen gornjem sloju odnosno *Flutter frameworku*. On ugrađuje temeljni C++ kod u *Dart* klase koje programeri onda mogu koristiti bez znanja o pozadinskoj apstrakciji.

Najčešće programeri imaju interakciju s *Flutter*-om kroz *framework* koji pruža reaktivan, moderan, pristup napisan *Dart* jezikom. Uključuje bogat skup temeljnih biblioteka, platformi, rasporeda i puno podslojeva. Tu su temeljne klase za iscrtavanje, animacije, nakon toga slijedi sloj za renderiranje elemenata na sučelje koji dinamički može mijenjati elemente ovisno o stanju, te na kraju *widgete* koji predstavljaju elemente na sučelju zajedno sa *Material* i *Cupertino* bibliotekama koje daju skup kontrola za određenu platformu. Iako *Flutter framework* sadrži dosta biblioteka, većina programera poseže za dodacima specifičnih za platformu koji su implementirani u obliku vanjskih biblioteka, primjerice za biblioteka za kameru, zahtjeve na poslužitelj, animacije i slično.



Slika 4.4 Slojevi *Flutter* Arhitekture

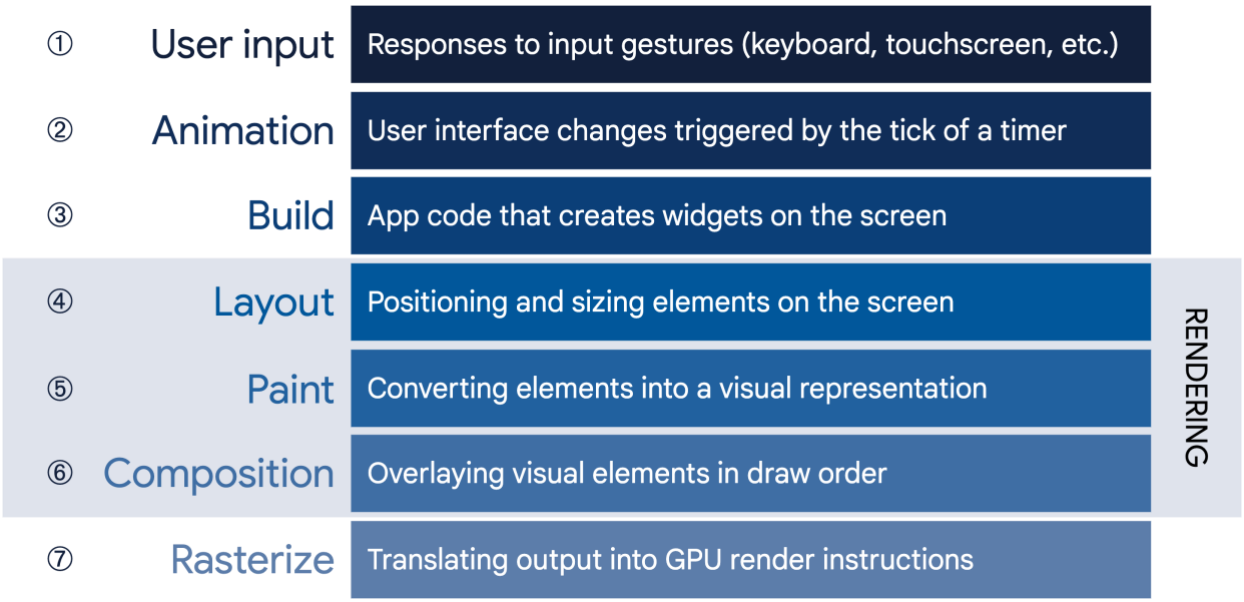
4.1.2.3. Stanje korisničkog sučelja

Flutter je reaktivni, pseudo-deklarativni *framework* korisničkog sučelja, u kojem se stanje sučelja osigurava preslikavanjem iz stanja aplikacije, a *framework* ima cilj ažurirati sučelje tijekom izvođenja ako se stanje aplikacije promijeni. Takav model inspiriran je *Facebook*-om i njihovim vlastitim *frameworkom React* koji zaobilazi tradicionalno načelo dizajna [27].

Većina *frameworka* korisničko sučelje opisuje jednom i potom posebno ažurira dijelove korisničkim kodom prilikom izvođenja, kao odgovor na događaje (eng. *evente*). Jedan od problema ovog pristupa je da ako aplikacija postaje složenija, programeru postaje teže ažurirati elemente korisničkog sučelja jer se stanja kaskadno nižu. *Flutter*, kao i drugi reaktivni okviri, ima alternativni pristup ovom problemu eksplicitnim odvajanjem njegovog osnovnog stanja od korisničkog sučelja. S API-jima se stvara samo opis korisničkog sučelja, a *framework* se brine o korištenju te jedne konfiguracije za ponovno stvaranje ili ažuriranje korisničkog sučelja prema

nužnosti. *Widgeti* su nepromjenjive klase koje se koriste za konfiguriranje stabla objekata u *Flutter*-u. U svojoj srži, *Flutter* je niz alata za učinkovito prolaženje po promijenjenim dijelovima stabala, pretvaranje stabala objekata u stabla objekata niže razine i širenje promjena kroz ta stabla. *Widget* prikazuje svoje korisničko sučelje pozivanjem metode *build*(), koja je funkcija koja pretvara stanje u korisničko sučelje. Kad god se od funkcije pokrene *build*(), *widget* bi trebao vratiti novo stablo, neovisno što je taj isti *widget* prethodno vratio. *Framework* obavlja kompliciraniji posao kako bi odredio koje funkcije izgradnje treba pozvati s obzirom na stabla objekta renderiranja. *Flutter* može ponovno kreirati samo dijelove korisničkog sučelja, na svakom prikazanom okviru, gdje se stanje mijenjalo pozivanjem metode *build*() *widgeta*. Važno je da se metode izgradnje brzo vrate, a težak računalni posao treba obaviti na neki asinkroni način, u pozadinskoj niti i potom spremiti kao dio stanja koje će koristiti metoda izgradnje. Iako je ovaj način malo naivan, ova automatizirana usporedba je dosta efektivna jer omogućuje interaktivne aplikacije sa visokim performansama. U *Flutteru* razlikuje se *StatelessWidget* koji nemaju promjenjivo stanje, primjerice ikona ili oznaka te *StatefulWidget* koji se trebaju mijenjati s obzirom na interakcije korisnika ili drugih čimbenika, primjerice, ako *widget* sadrži brojač i on se smanjuje kad korisnik pritisne gumb, vrijednost brojača je stanje tog *widgeta* i treba se ažurirati njegov dio korisničkog sučelja. Iz ovog opisa može se zaključiti da *Flutter* ima isti deklarativan pristup kao i *Jetpack Compose* kod native Android aplikacije, renderira samo dijelove, odnosno *widgete* koji su promijenili svoje stanje u odgovoru na neku interakciju.

Višeplatformski okviri obično rade kreiranjem sloja apstrakcije preko temeljnih izvornih Android i iOS biblioteka korisničkih sučelja i pokušavaju poboljšati nedostatke prikaza svake platforme. Taj način dodaje dodatne troškove koji mogu biti intenzivni, pogotovo tamo gdje postoji velika interakcija između logike aplikacije i korisničkog sučelja. *Flutter* reducira te apstrakcije koristeći vlastiti skup svojih *Widgeta*. *Dart* kod koji iscrtava *Flutter*-ove elemente kompilira se u izvorni kod koji koristi *Skia* za renderiranje. *Flutter* također izgrađuje vlastitu kopiju *Skia*-e kao dio *engine*-a, omogućujući programeru da nadogradi aplikaciju kako bi ostao u tijeku s najnovijim poboljšanjima performansi čak i ako uređaj nije ažuriran s novom verzijom Androida. To vrijedi i za *Flutter* na drugim izvornim platformama, kao što su Windows, iOS, ili macOS. Glavno načelo koje *Flutter* primjenjuje za renderiranje je brzina. Tok podataka se može vidjeti na slici 4.5.



Slika 4.5 Tok podataka pri interakciji korisnika

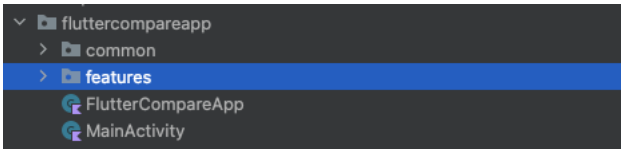
4.1.2.4. Razvojno okruženje

Flutter aplikacije se mogu izraditi koristeći bilo koji uređivač teksta u kombinaciji s *Flutter* SDK alatima naredbenog retka. Preporučena razvojna okruženja su *Android Studio*, *VS Code*, *IntelliJ*. Ta okruženja pružaju dovršetak koda, pomoć pri uređivanju *widgeta*, isticanje sintakse, otklanjanje pogrešaka i podršku za pokretanje. Moguće je instalirati i vanjske *plugin*-e koji imaju dodatne funkcionalnosti za ugodnije rukovanje kodom i aplikacijom pri razvoju.

4.2. Implementacija u Androidu

4.2.1. Arhitektura Android aplikacije

Arhitektura nativne aplikacije u Androidu podijeljena je na dijelove prikazane na slici 4.6.



Slika 4.6 Osnovna podjela u projektu

Prikazana arhitektura na najvišoj razini apstrakcije je prema slici *feature-first* i zahtijeva da se stvori nova mapa za svaku novu značajku koja se dodaje aplikaciji. A unutar mape mogu se dodati slojevi kao podmape [28]. Za usporedbu može se uzeti *layer-first* pristup prema slici gdje za svaku

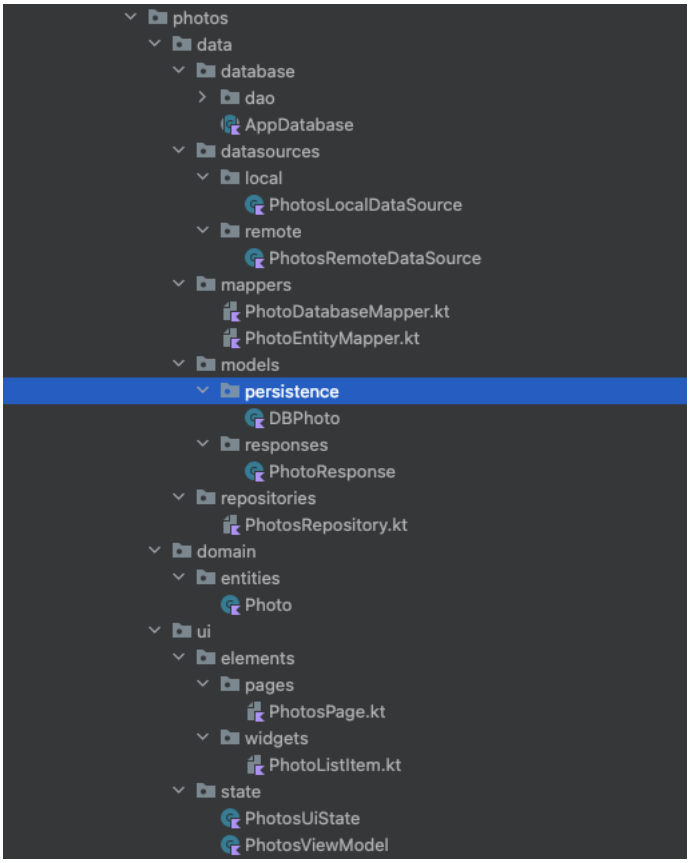
značajku njene datoteke raspoređujemo po slojevima koji su zajednički za sve. Ovakav način je dosta lagan za koristiti u praksi, ali nije dovoljno skalabilan ako aplikacija raste. Prilikom rada na značajki skače se na različite direktorije koji su dosta udaljeni jedno od drugog i ako odlučimo obrisati neku značajku to postaje puno teže jer ćemo morati provjeravati svaki sloj i brisati povezane datoteke. Ovaj pristup može dobro funkcionirati ako je aplikacija manje veličine i neće se dalje nadograđivati.

Slika 4.7 Layer-first arhitektura projekta

Feature-first raspoređuje svoje datoteke prema značajkama, a ne slojevima. Isto tako po potrebi se može imati zajednički *feature* pod nazivom *common* gdje se nalaze neki elementi koji se provlače kroz više značajka. U ovom pristupu ako hoćemo ažurirati pojedinu značajku možemo se fokusirati samo na jedan direktorij, te u slučaju da hoćemo obrisati neku od značajki to možemo napraviti brisanjem samo jednog direktorija. Prema tome, može se zaključiti da ovaj način ima puno više prednosti od prethodnog.

Slika 4.8 Feature-first arhitektura

Svaka značajka unutar svog direktorija ima direktorije organizirane prema preporučenoj arhitekturi od strane Android dokumentacije i možemo je vidjeti na slici 4.9.



Slika 4.9 Arhitektura projekta

Svaka aplikacija mora imati dva sloja, a to su sloj korisničkog sučelja i sloj podataka (eng. *data layer*), dok je *domain* sloj opcionalan. Sloj korisničkog sučelja pokazuje podatke aplikacije na zaslonu. Podatkovni sloj sadrži poslovnu logiku aplikacije i dostavlja podatke aplikacije [29]. Dodatni sloj domene se koristi za pojednostavljenje i mogućnost ponovnog iskorištavanja koda prilikom komunikacije UI i sloja podataka.

Prvi sloj najbliži korisniku je sloj korisničkog sučelja ili prezentacijskog sloja. Uloga sloja je prikazati podatke na zaslonu. Prilikom promijene podataka, bilo zbog interakcije korisnika (kao što je na primjer pritisak na gumb) ili nekog vanjskog dohvaćanja (kao što je, primjerice, odgovor mreže), korisničko sučelje se treba ažurirati kako bi prikazalo promjene. Ovaj sloj se sastoji od dvije podgrupe [29]:

Elementi korisničkog sučelja izgrađeni pomoću *Compose* funkcija koji prikazuju podatke na zaslonu. Isto tako ovaj sloj sadrži navigaciju, teme i ostale elemente koji čine sučelje. Direktorij je podijeljen na zasebne *widget* funkcije koji se uklapaju u zaslon *Compose* funkcije.

Držači stanja (kao što je klasa *ViewModel*) koja drži podatke i izlaže ih korisničkom sučelju te rukuju poslovnom logikom aplikacije. Ovaj sloj reagira na interakcije korisnika i poziva metode iz podatkovnog sloja te dobiva povratno podatke koje dostavlja i prikazuje relevantne promijene na sloj korisničkog sučelja. Podatke na korisničko sučelje daje u obliku stanja prema kojem UI može renderirati zaslon sa tim stanjem.

Podatkovni sloj aplikacije sadrži poslovnu logiku. Poslovna logika je ono što aplikaciji daje pravu vrijednost. Sastoji se od metoda koje određuju kako aplikacija dohvaća, pohranjuje te vrši promijene nad podacima.

Podatkovni sloj sadržava spremište od kojih svako može sadržavati više izvora podataka. Repozitorij klasa kreira se za svaku različitu vrstu podataka kojima se rukuje u aplikaciji. Tako u ovom primjeru postoji *PhotosRepository*, a repozitorij koji sadrži podatke o korisniku bi bio *UserRepository*. Repozitorij klase odgovorne su za izlaganje podataka ostatku aplikacije, centraliziranje pohrane i promjena podataka, rukovanje različitim izvorima podataka i držati poslovnu logiku kako bi *ViewModel* bio razumljiviji. Klase izvora podataka (*DataSource*) imaju zadatak za rad sa isključivo jednim izvorom podataka koji može biti lokalna baza podataka, poslužitelj, datoteka i slično. Ove klase su poveznica između aplikacije i vanjskog sustava koji sadrži podatke i operacije nad njima. U ovom sloju se također nalaze modeli te njihovi pretvarači (*Mapper*-i) koji pretvaraju modele iz izvora u neki drugi oblik pogodan za daljnju pohranu ili korištenje i obrnuto. Isto tako, lokalna baza podataka i pristupna klasa za bazu se nalaze u ovom sloju jer rade operacije nad podacima aplikacije.

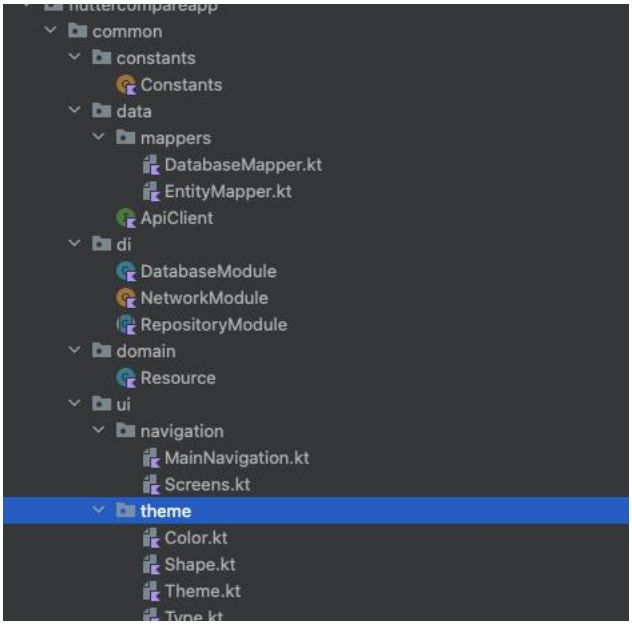
Domenski sloj je opcionalni i nalazi se između podatkovnog sloja i korisničkog sučelja.

Odgovoran je za enkapsulaciju kompleksne poslovne logike ili jednostavne poslovne logike koju više *ViewModela* ponovno koriste. Sloj nije obavezan jer nemaju sve aplikacije složene zahtjeve. Treba se koristiti samo kada je to potrebno u već opisane dvije situacije. U ovom sloju se nalaze i entiteti koji predstavljaju modele pogodne za prikaz na korisničkom sučelju.

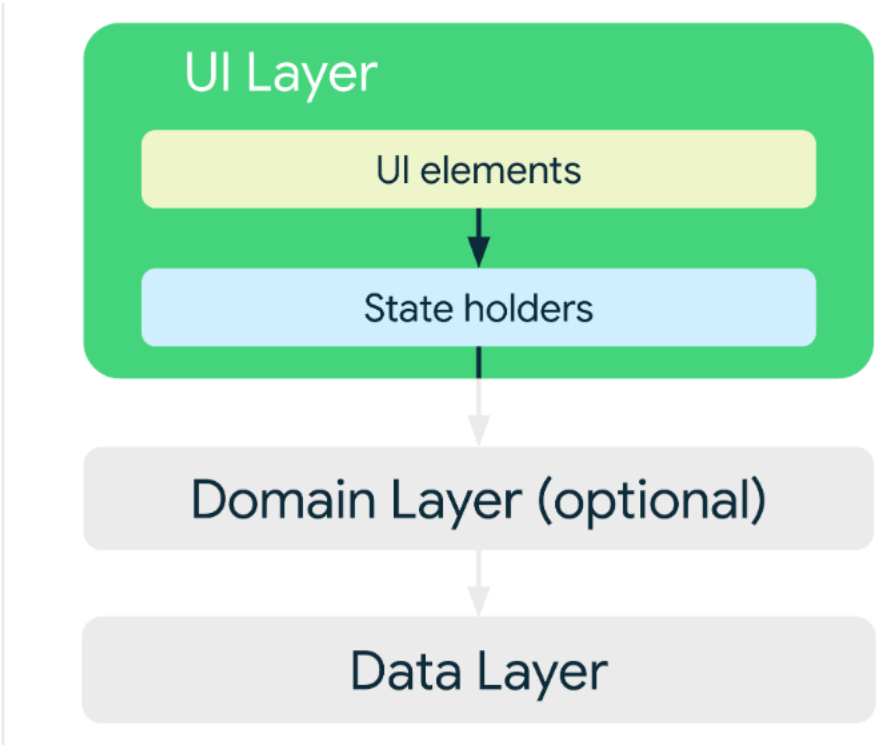
Kako bi svi slojevi međusobno komunicirali preporučen način povezivanja je koristeći ubrizgavanje ovisnosti (eng. *Dependency injection*) koji omogućava klasama da definiraju svoje ovisnosti bez da ih konstruiraju. Ovakav način omogućava brzo prebacivanje između proizvodnih i testnih implementacija te smanjuje složenost i dupliciranje koda koji bi bio kada bi se klase instancirale jedne unutar drugih. Preporučena biblioteka za ovu funkcionalnost je *Hilt* koji automatski konstruira objekte prolazeći po stablu ovisnosti, stvara spremnike ovisnosti za klase *frameworka* i pruža jamstva za vrijeme kompajliranja ovisnosti.

Dobra arhitektura omogućuje lakše održavanje, kvalitetu i robusnost aplikacije te skaliranje jer više timova može raditi na istom kodu uz minimalna isprepletanja. Kod je puno razumljiviji i strukturiraniji pa novim članovima tima je lakše ispratiti komponente aplikacije. Isto tako, jedan od bitnih prednosti je sposobnost lakšeg testiranja jer su komponente izdvojene u zasebne cjeline i slojeve pa je moguće testirati metode svakog sloja posebno.

Direktorij *common* je zajednički za sve značajke u aplikaciji i tu se nalazi ubrizgavanje ovisnosti, zajedničke bazne klase koje ostale u aplikaciji nasljeđuju ili koriste na neki drugi način, teme, boje aplikacije i ostalo.



Slika 4.10 Raspored datoteka po slojevima



Slika 4.11 Osnovna podjela slojeva u arhitekturi projekta

4.2.1. Velika lista u Androidu

U ovom odlomku prikazan je proces dohvaćanja i prikazivanja slika na listi.

```
@Composable
fun PhotosPage(
    authViewModel: AuthViewModel = hiltViewModel(),
    photosViewModel: PhotosViewModel = hiltViewModel(),
    onPhotoClicked: (photo: Photo) -> Unit,
    onMapsClicked: () -> Unit,
) {

    LaunchedEffect( key1: true) { this: CoroutineScope
        photosViewModel.getPhotos()
    }

    Scaffold(
        topBar = {
```

Slika 4.12 Pozivanje dohvaćanja slika iz UI sloja

Arhitektura aplikacije MVVM (Model – View – ViewModel), stoga se u ViewModelu drži stanje aplikacije. Iz priložene slike može se vidjeti da se pri kreiranju Compose elementa poziva dohvaćanje slika pomoću metode iz ViewModela. LaunchedEffect se izvršava samo jednom iako bude više renderiranja ako je key1 nepromijenjiv.

```
  private val repository: PhotosRepository,
: ViewModel() {

  var uiState by mutableStateOf(PhotosUiState())
      private set

  Matej Kovacevic 2
  fun getPhotos() {
      viewModelScope.launch { this: CoroutineScope
          uiState = uiState.copy(
              isLoading = true,
              error = null
          )

          when (val result =
              repository.refreshPhotos()) {
              is Resource.Error -> {
                  uiState = uiState.copy(
                      isLoading = false,
                      error = result.message
                  )
              }
              is Resource.Success -> {
                  uiState = uiState.copy(
                      photos = result.data ?: emptyList(),
                      isLoading = false,
                      error = null
                  )
              }
```

Slika 4.13 Promijena stanja u *ViewModel*-u

ViewModel dalje propagira zahtjev na repozitorij i čeka odgovor kako bi ažurirao stanje sa rezultatom ako je uspješno ili greškom ako se dogodi. Za pozivanje asinkronih funkcija kao što je *refreshPhotos* koriste se *korutine* koje omogućuju da se blok koda odvija asinkrono na drugoj niti i da se asinkroni kod poziva u sinkronom obliku. Pozivanje *korutine* u *ViewModel* klasi najbolje je ograničiti na *ViewModel* područje (*viewModelScope*) i poziva se metodom *launch*.

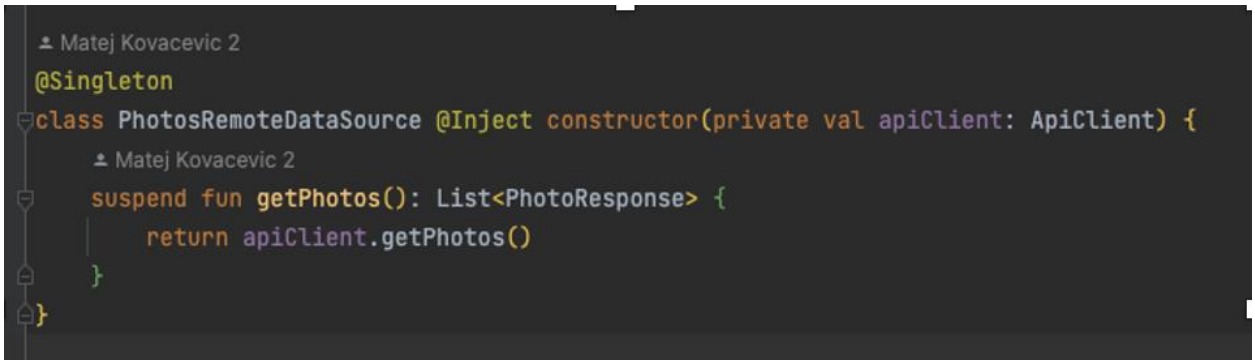
Slika 4.4 Dohvaćanje i spremanje slika u podatkovnom sloju

ViewModel poziva metodu *refreshPhotos* koja povlači slike sa poslužitelja korištenjem *RemoteDataSource* i sprema ih u lokalnu bazu podataka preko *LocalDataSource*. Nakon spremanja podataka u lokalnu bazu, isti se dohvaćaju i mapiraju u domenski oblik pogodan za korištenje u sloju korisničkog sučelja. Ova funkcija označena je sa *suspend* ključnom riječi, što znači da može biti pauzirana i nastavljena u bilo kojem trenutku te može izvršavati zahtjevnu i dugotrajnu operaciju bez da blokira glavnu nit jer se može pozivati samo iz *korutine*. Svaka akcija obavijena je sa praćenjem vremena potrebnog za izvršavanje korištenjem *Performance Monitoring* od *Firebase* platforme.

Slika 4.5 Poziv na poslužitelj

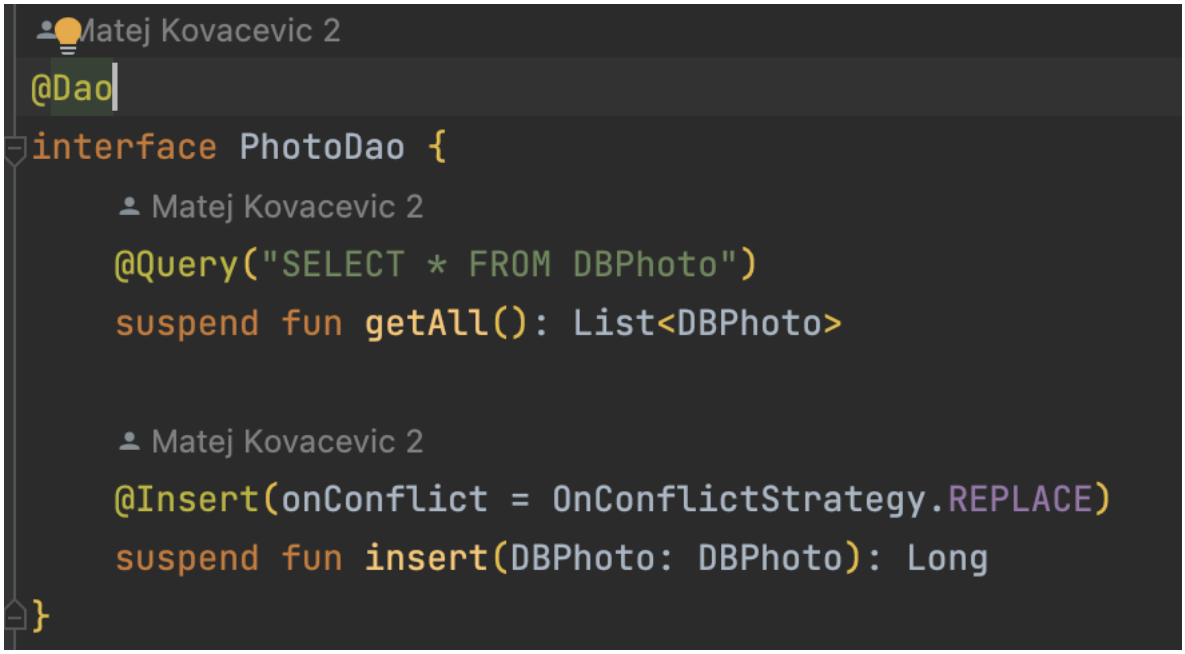
S poslužitelja su podaci o slikama dohvaćeni korištenjem *Retrofita*. *Retrofit* je brza i lagana biblioteka koja se koristi za dohvaćanje i slanje podataka na poslužitelj koji ima REST arhitekturu. *Retrofit* predstavlja apstrakciju na HTTP pozive i rješava greške i prije nego što se uspostavi interakcija s poslužiteljem.

PhotosRemoteDataSource pruža još jedan dodatan sloj između repozitorija i API poziva na poslužitelj, što je vidljivo na slici 4.16.



Slika 4.16 Udaljeni izvor slika

Razlog zašto postoji dodatan sloj između repozitorija i pristupnih točaka za lokalnu bazu i poslužitelj je u tome što postoje lokalni i udaljeni podaci koji moraju biti sinkronizirani pa je ovakva arhitektura povoljna za takve funkcionalnosti.



Slika 4.7 Metode za pristupanje lokalnoj bazi

Kako bi korisnik mogao pregledavati listu slika i kada nije spojen na mrežu, aplikacija za lokalno spremanje koristi *Room* biblioteku koja daje sloj apstrakcije preko *SQLitea* kako bi se omogućilo lakše korištenje i pristup bazi podataka i pritom iskorištava puna snaga *SQLitea*. *Room* omogućuje provjeru SQL upita tijekom kompajliranja, pojednostavljenju migraciju tablica u bazi te već neke dostupne funkcije kao što su dodavanje, ažuriranje, brisanje bez pisanja SQL upita.

PhotosLocalDatasource nudi još jedan dodatan sloj između *Room*, odnosno lokalne baze podataka i repozitorija, što možemo primijetiti na slici 4.18.

```
@Singleton
class PhotosLocalDataSource @Inject constructor(private val photoDao: PhotoDao) {

    Matej Kovacevic 2
    suspend fun insertPhoto(photoResponse: PhotoResponse): Long {
        return photoDao.insert(photoDatabaseMapper(photoResponse))
    }

    Matej Kovacevic 2
    suspend fun getPhotos(): List<DBPhoto> {
        return photoDao.getAll()
    }
}
```

Slika 4.18 Lokalni izvor podataka

Nakon uspješnog dohvaćanja slika mijenja se stanje definirano u *ViewModel*-u i ažurira se *Compose widget* koji o tom stanju ovisi. *Widget* koji ovisi o dohvaćenim slikama je *LazyColumn* i on se adekvatno ažurira novim podacima, možemo vidjeti na slici 4.19.

```
LazyColumn(
    modifier = Modifier.weight(1f)
) { this: LazyListScope
    items(
        items = photosViewModel.uiState.photos,
        key = { it.id }
    ) { this: LazyItemScope  photo ->
        PhotoListItem(photo = photo, onPhotoClick = { it: String
            onPhotoClicked(photo)
        })
    }
```

Slika 4.9 Widget koji prikazuje slike u listi

4.2.2. Google Maps i ažuriranje lokacije korisnika

Google Maps platforma je grupa API-ja i SDK-ova koji programerima i inženjerima omogućuju instaliranje Google karata u web-stranice i mobilne aplikacije ili učitavanje podataka s Google karata [30].

Za potrebe ovog projekta korištena je Google karta i GPS senzor kako bi se dobila trenutna lokacija korisnika u intervalima od pola sekunde. Takva funkcionalnost zajedno s učitavanjem mape iziskuje veće korištenje CPU i RAM-a što je poželjno za potrebne testove.

Jetpack Compose ima posebnu biblioteku *Maps Compose* za *Map*-s SDK. To je skupina funkcija otvorenog koda koje se mogu sastaviti i tipova podataka koji se mogu koristiti s *Compose*-om za izradu aplikacije. Kako izgleda *Composable* funkcija koja prikazuje *Widget* kartu, može se vidjeti u sljedećoj slici 4.20.

```
GoogleMap(
    uiSettings = MapUiSettings(myLocationButtonEnabled = true),
    properties = MapProperties(isMyLocationEnabled = true),
    modifier = Modifier.fillMaxSize(),
    cameraPositionState = cameraPositionState,
)
```

Slika 4.20 UI element za prikaz Google karte

Widget prima atribute koji omogućuju prikaz lokacije korisnika na mapi, određuju veličinu mape korištenjem *Modifier* element koji omogućuje uređivanje i određuje ponašanje karte. Osim toga postoji i *cameraPositionState* promjenjiva varijabla koja postavlja položaj na karti. Tip podatka ove varijable je *rememberSaveable* koja omogućava pohranjivanje objekta u memoriji i držanje stanja tijekom rekompozicije i konfiguracijskih promjena (promjena iz portret u pejzažni način rada). Kada se pozicija kamere promijeni *GoogleMap widget* se ponovno kreira sa novom vrijednošću pozicije.

```
val cameraPositionState = rememberCameraPositionState { this: CameraPositionState
    position = CameraPosition.fromLatLngZoom(LatLng( latitude: 0.0, longitude: 0.0), zoom: 14f)
}
```

Slika 4.21 Varijabla za poziciju na Google karti

Kako bi se mogla pratiti lokacija korisnika potrebno je zbog zaštite privatnosti korisnika, zahtijevati dozvole za lokaciju [31]. Korisnika može dati dopuštenje za precizno dohvaćanje

lokacije (*ACCESS_FINE_LOCATION*) ili lokaciji koja otprilike odgovara položaju korisnika (*ACCESS_COARSE_LOCATION*).

```
    }
    val launcher = rememberLauncherForActivityResult(
        ActivityResultContracts.RequestMultiplePermissions()
    ) { it: Map<String, Boolean>
        val isNotGranted = it.values.contains(false)
        if (!isNotGranted) {
            fusedLocationClient.requestLocationUpdates(
                locationRequest,
                locationCallback,
                Looper.getMainLooper()
            )

        }
    }
```

Slika 4.22 Traženje dopuštenja lokacije i ažuiriranje trenutne lokacije

Ako je pristup lokaciji dopušten, omogućena je metoda za ažuriranje lokacije, odnosno ažuriranje položaja kamere ovisno o lokaciji dobivenoj u povratnoj funkciji (*callback*) *locationCallback,* kao što je to navedeno u sljedećem kodu.

```
launcher.launch(
    arrayOf(
        Manifest.permission.ACCESS_FINE_LOCATION,
        Manifest.permission.ACCESS_COARSE_LOCATION
    )
```

Slika 4.23 Traženje dopuštenja lokacije

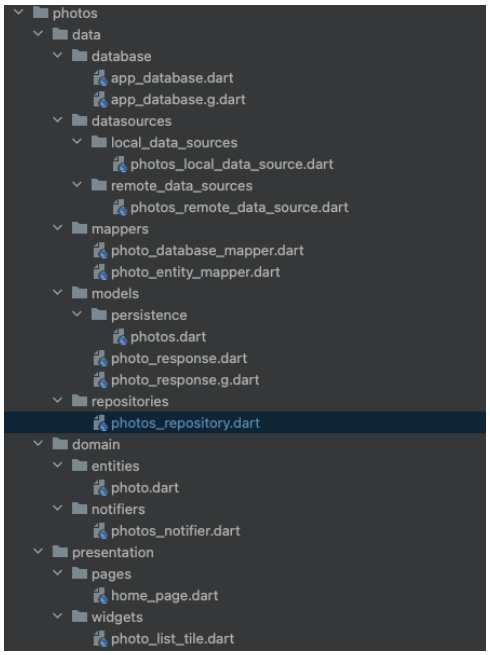
```
val locationCallback = remember {
    object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult) {
            super.onLocationResult(locationResult)
            cameraPositionState.position = CameraPosition.fromLatLngZoom(
                LatLng(
                    locationResult.lastLocation.latitude,
                    locationResult.lastLocation.longitude
                ), zoom: 14f
            )
        }
    }
}
```

Slika 4.24 Ažuriranje pozicije kamere na karti

4.3. Implementacija u Flutteru

4.3.1. Arhitektura Flutter aplikacije

Arhitektura upotrebljena u *Flutteru* je slična kao i u aplikaciji napravljenoj u Androidu. Aplikaciju čini već tri navedena sloja (podatkovni sloj, domenski sloj i UI sloj). Arhitektura *Flutter* aplikacije prikazana je na slici 4.25.



Slika 4.25 Arhitektura *Flutter* projekta

4.3.2. Velika lista u Flutteru

Iz priložene arhitekture može se zaključiti da je arhitektura gotovo identična arhitekturi nativne aplikacije. Isto postoje tri sloja, pri kojem u podatkovnom sloju postoji repozitorij koji dohvaća slike iz udaljenog izvora (*PhotoRemoteDataSource*) i sprema ih u lokalnu bazu podataka preko klase koja predstavlja lokalni izvor (*PhotosLocalDataSource*). Pozivanje navedenih metoda te mjerenje vremenskih intervala pomoću *Firebase Perfomance monitoring* alata može se vidjeti na slici 4.26.

```dart
@override
EitherFailureOr<List<Photo>> refreshPhotos() async => execute(() async {
      Trace getPhotosNetworkTrace = FirebasePerformance.instance
          .newTrace(FirebaseConstants.getPhotosFromNetworkTrace);
      await getPhotosNetworkTrace.start();
      final photoResponses = await _apiClient.getPhotos();
      await getPhotosNetworkTrace.stop();

      Trace saveToDatabaseTrace = FirebasePerformance.instance
          .newTrace(FirebaseConstants.saveToDatabaseTrace);
      await saveToDatabaseTrace.start();
      await _photosLocalDataSource.savePhotos(photoResponses);
      await saveToDatabaseTrace.stop();

      Trace readFromDatabaseTrace = FirebasePerformance.instance
          .newTrace(FirebaseConstants.readFromDatabaseTrace);
      await readFromDatabaseTrace.start();
      final localPhotos = await _photosLocalDataSource.getPhotos();
      await readFromDatabaseTrace.stop();
      final domainPhotos = localPhotos.map(_photoMapper).toList();
      return Right(domainPhotos);
    });
}
```

Slika 4.26 Dohvaćanje i spremanje slika u podatkovnom sloju

Može se primijetiti da je kod i postupak jako sličan Android programskom kodu i da bi mjerenja trebala biti validna. U ovom slučaju jedina značajnija razlika je u tome što se koristi pomoćnu funkcija za enkapsuliranje asinkronih poziva *execute()* koja vraća uspješan rezultat ili grešku s porukom. Za lokalnu bazu podataka koristi se biblioteka *drift_sqflite* (nekadašnji *moor*) koji je po sintaksi, načinu korištenja i performansama najsličniji *Room*-u u nativnom Androidu.

```dart
mixin ErrorToFailureMixin {
  EitherFailureOr<T> execute<T>(
    EitherFailureOr<T> Function() function, {
    String? failureTitle,
  }) async {
    try {
      return await function();
    } catch (err, stackTrace) {
      log(err.toString());
      log(stackTrace.toString());
      return Left(Failure.generic(
        title: failureTitle,
        error: err,
        stackTrace: stackTrace,
      ));  // Failure.generic, Left
    }
  }
}
```

Slika 4.7 Ekstenzija za pozive dohvaćanja s poslužitelja

Dio koji se razlikuje od nativnog androida je *state management.* U nativnoj aplikaciji koristio se model MVVM gdje se logika upravljanja stanjem držala u *ViewModel* koji je na događaje korisnika kao što je pritisak na gumb nakon dohvaćanja podataka iz viših slojeva prezentirao podatke u obliku stanja na korisničko sučelje. Prema slici, u Flutter aplikaciji postoji slični princip korištenjem *Riverpod* biblioteke koja sadrži *Provider*-e koji se ponašaju kao *ViewModel*-i. Međutim postoji puno vrsta *Provider*-a, a najsličniji *ViewModelu* je *StateNotifier*. On u sebi drži stanje liste slika koje prezentira korisničkom sučelju kako bi se *Widget* stablo moglo ponovno izgraditi s tim novim stanjem. Isto tako sadrži i metodu *getPhotos*() koja se može pozivat iz koda korisničkog sučelja ili u ovom slučaju pri inicijalizaciji *StateNotifier*-a. Ova metoda poziva dohvaćanje slika iz repozitorija i ažurira korisničko sučelje sa novom listom ako je dohvaćanje uspješno ili s porukom greške ako nije.

Stanje na korisničkom sučelju sluša se pomoću metode watch na ref objektu koji omogućava *Widget*-ima da komuniciraju s *Provider*-om. Može se vidjeti ako su uspješno dohvaćeni podaci da se izvršava kod unutar dana grane.

```dart
import 'package:fluttercompareapp/features/photos/domain/entities/photo.dart';

final photosNotifierProvider =
    StateNotifierProvider<PhotosNotifier, BaseState<List<Photo>>>((ref) {
  return PhotosNotifier(
    ref.watch(photosRepositoryProvider),
    ref,
  )..getPhotos();  // PhotosNotifier
});  // StateNotifierProvider

class PhotosNotifier extends BaseStateNotifier<List<Photo>> {
  final PhotosRepository _photosRepository;

  PhotosNotifier(this._photosRepository, super.ref);

  void getPhotos() => execute(
        _photosRepository.refreshPhotos(),
      );
}
```

Slika 4.28 Promjena stanja sučelja u *Notifier*-u

Ono što je velika prednost *Riverpoda* osim različitih vrsta *Provider*-a za razne svrhe, je to što ga je moguće koristiti za ubrizgavanje ovisnosti. Na slici 1 se može vidjeti kako se repozitorij može ubrizgati u *StateNotifier* pomoću *ref.watch* metode bez potrebe korištenja neke druge biblioteke za tu istu funkcionalnost.

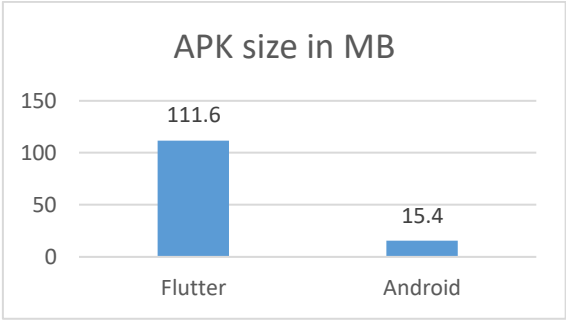
5. MJERENJA I ANDROID I FLUTTER TEHNOLOGIJE

S obzirom na anketu stranice *Statista*, 42% programera koristi *Flutter* kao višeplatformsku tehnologiju u usporedbu sa 38% *React Native* i 11% *Xamarin* [32]. Obzirom da je najkorištenija tehnologija među višeplatformskim, uključen je u mnoge rasprave vezane za usporedbu s nativnim aplikacijama. Ovim radom subjektivne doživljaje programera i ostalih sudionika u IT industriji potkrijepit će se činjenicama i mjerenjima na obje platforme uz pomoć sofisticiranih alata.

5.1. Mjerenja

5.1.1. Veličina aplikacije

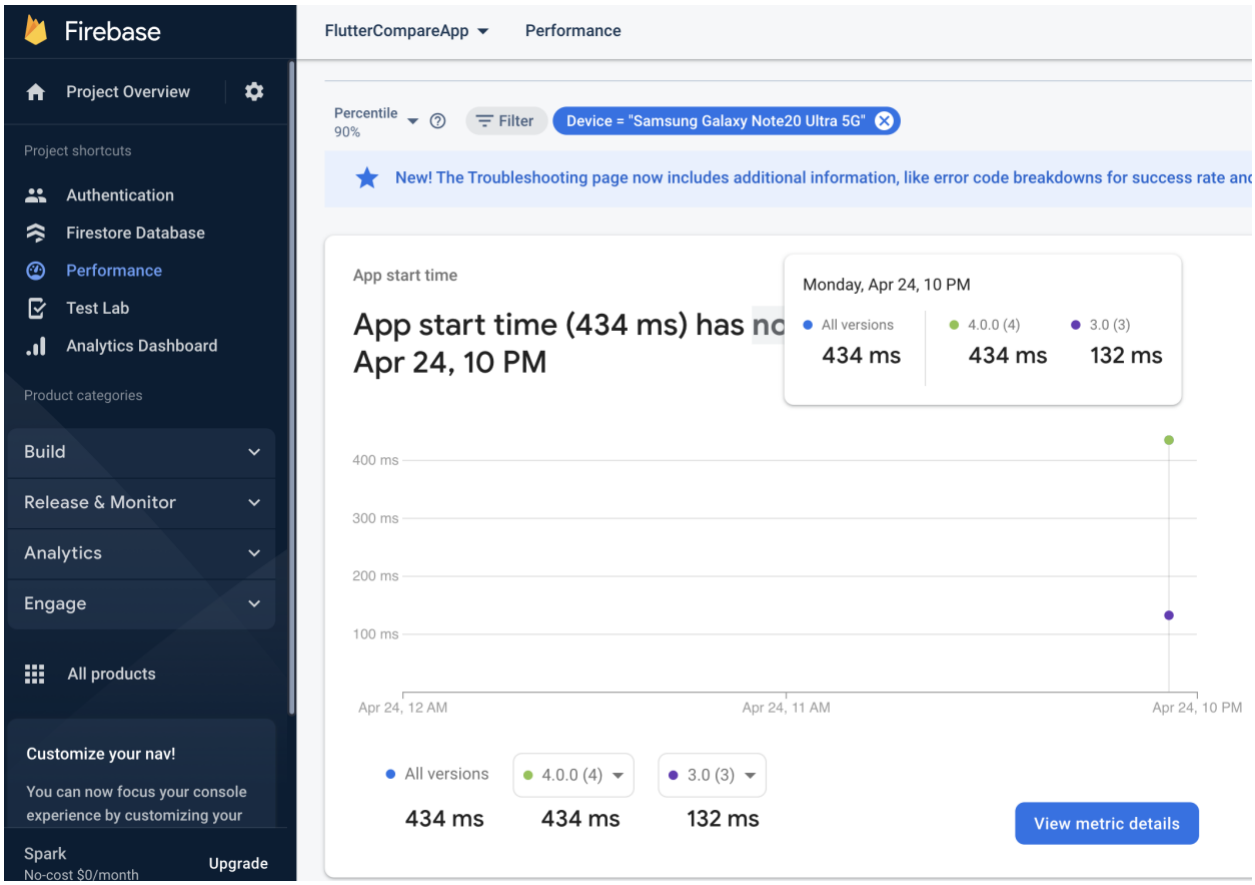
Prema rezultatima veličina aplikacija pri izgradnji prikazanih na slici 5.1, vidljivo da je veličina *Android* aplikacije pri izgradnji skoro 8 puta manja nego aplikacija u *Flutter* tehnologiji što je velika razlika koja u kompleksnijim aplikacijama može biti nezaobilazni problem.



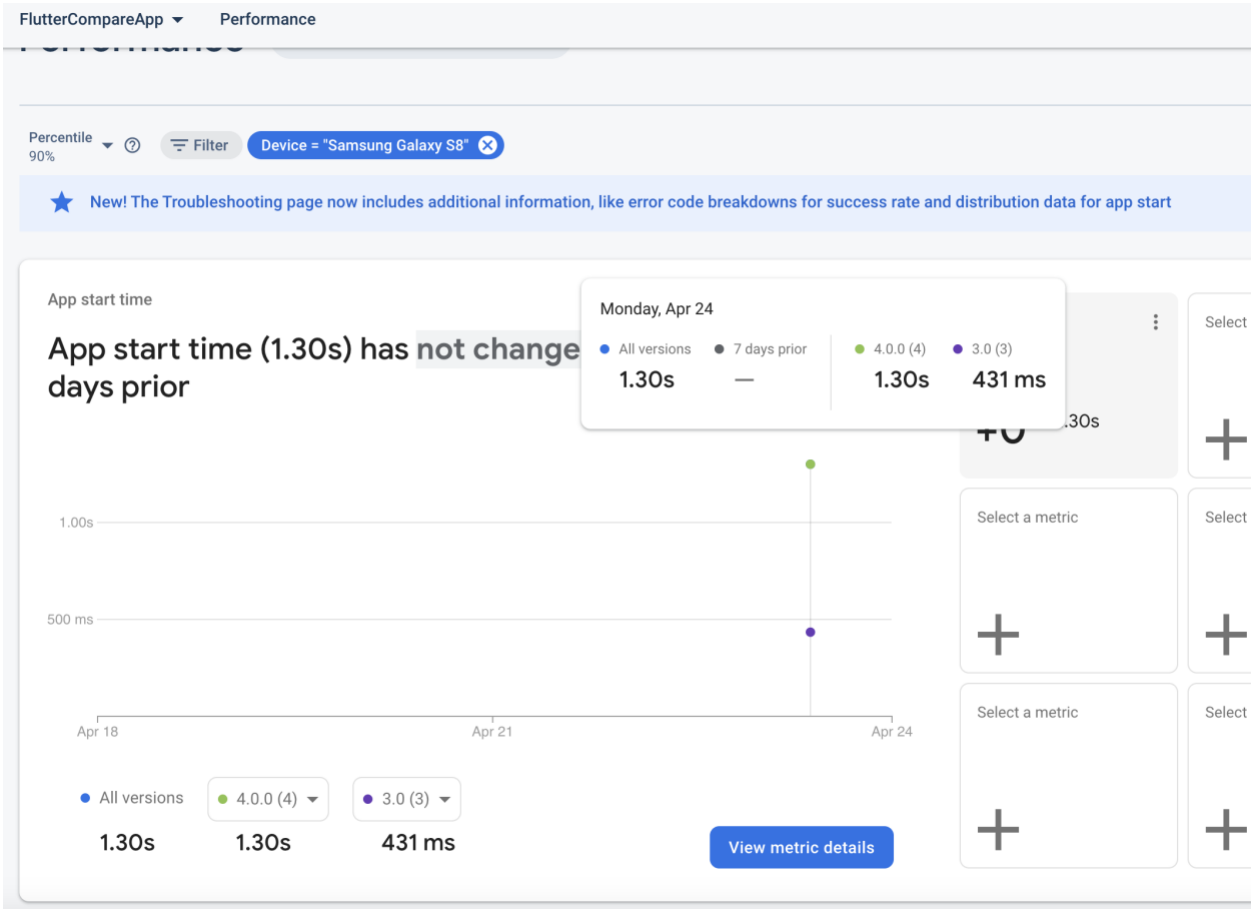
Slika 5.1 Veličina aplikacije pri instalaciji

5.1.2. Vrijeme pokretanja

Vrijeme pokretanja u milisekundama aplikacije može se vidjeti na slici 5.2 Note20 Ultra uređaj te na slici 5.3 na Samsung S8. Prema rezultatima zaključuje se da Flutteru treba skoro 70% više vremena da starta u odnosu na Android native aplikaciju. Za potrebe ovog mjerenja koristio se Firebase Performance Monitoring koji ima mogućnost filtrirati rezultate prema različitim parametrima kao što je razdoblje, uređaj, verzija i ostali mogući filtri. Sve podatke moguće je prikazati na grafikonu u obliku linija ili drugih oblika. Za raspoznavanje aplikacija korištena je verzija 3 za native aplikaciju te verzija 4 za Flutter aplikaciju.



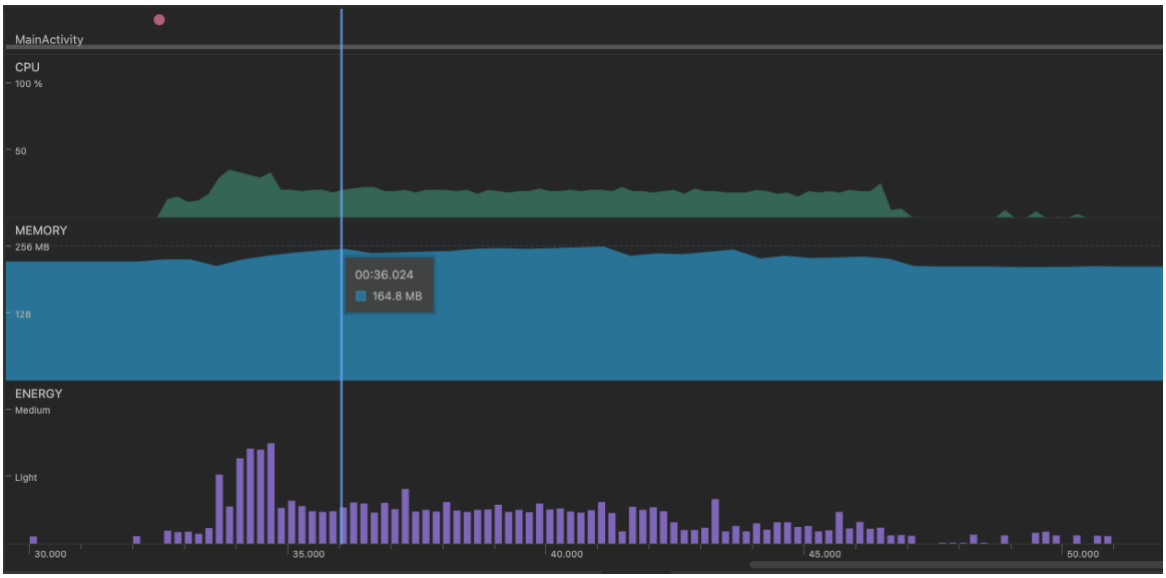
Slika 5.2 Vrijeme pokretanja aplikacije na Samsung Note20 Ultra uređaju



Slika 5.3 Vrijeme pokretanja aplikacije na Samsung Galaxy S8 uređaju

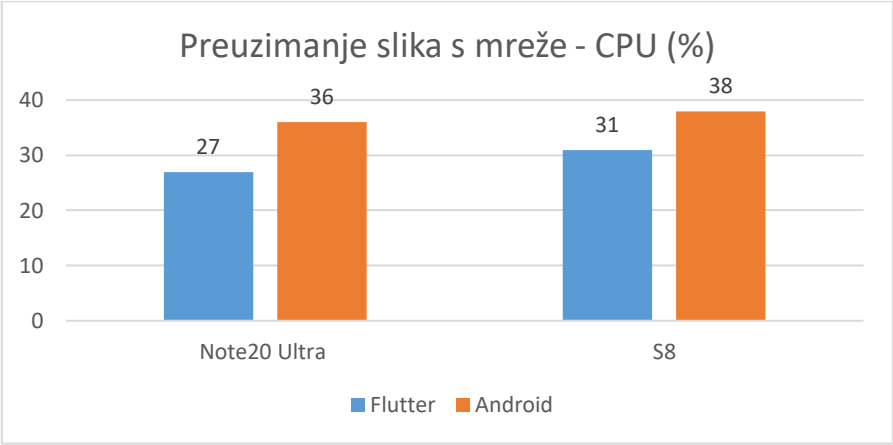
5.1.3. Prikaz liste slika

Za potrebe mjerenja korištenja procesora i memorije koristi Android Studio razvojno okruženje i njegovi ugrađeni alati za profiliranje. CPU *profiler* se koristi za mjerenje performansi vezan za vrijeme izvođenja, dok se *Memory profiler* koristi za praćenje alokacije memorije [14]. CPU *profiler* omogućuje praćene korištenja procesa u stvarnom vremenu i prema određenim nitima i funkcijama dok se koristi aplikacija. *Memory profiler* omogućuje provjeru ako dolazi do curenja memorije koja može uzrokovati usporavanje aplikacija pa čak i pucanje i zatvaranje aplikacije. Svi podaci mogu se vidjeti u brojčanom obliku, ali i grafičkom u obliku grafova koji se mogu analizirati i eksperimentirati s njima. Na slici 5.4 može se vidjeti korištenje *Profiler*-a u razvojnom okruženju uz prikaz CPU i memorijskog korištenja.

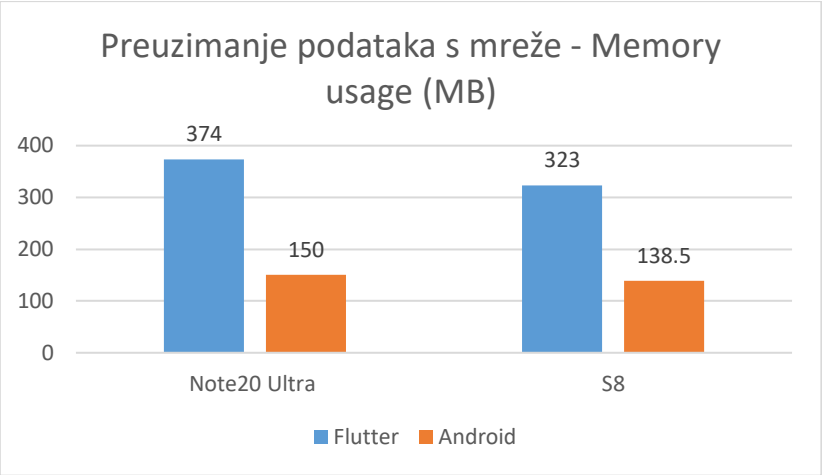


Slika 5.4 Korištenje alata *Android Studio*-a za provjeru performansi

Rezultati testiranja prosječnog korištenja procesora u postotcima za uređaje Note20 Ultra i S8 pri dohvaćanju podataka su prikazani na slici 5.5, te korištenje memorije na slici 5.6. Sa slika je vidljivo da aplikacija napravljena u *Androidu* koristila veći udio procesora pri dohvaćanju podataka s mreže u odnosu na aplikaciju u *Flutter-u*. Međutim, potrošnja memorije od 323 i 374 MB je duplo veća u *Flutteru* za oba uređaja. Ovaj rezultat nije iznenađujući ako se u obzir uzme da *Flutter* ne koristi nativne *plugine* za HTTP zahtjeve, stoga je moguće da njegova implementacija ima manju potrošnju CPU.

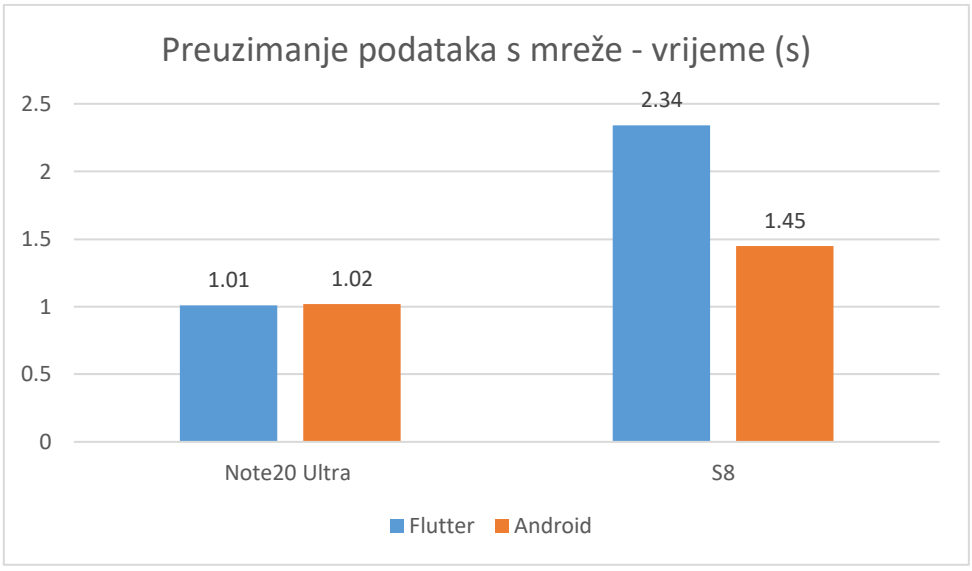


Slika 5.5 Prosječno opterećenje procesora pri preuzimanju slika s mreže



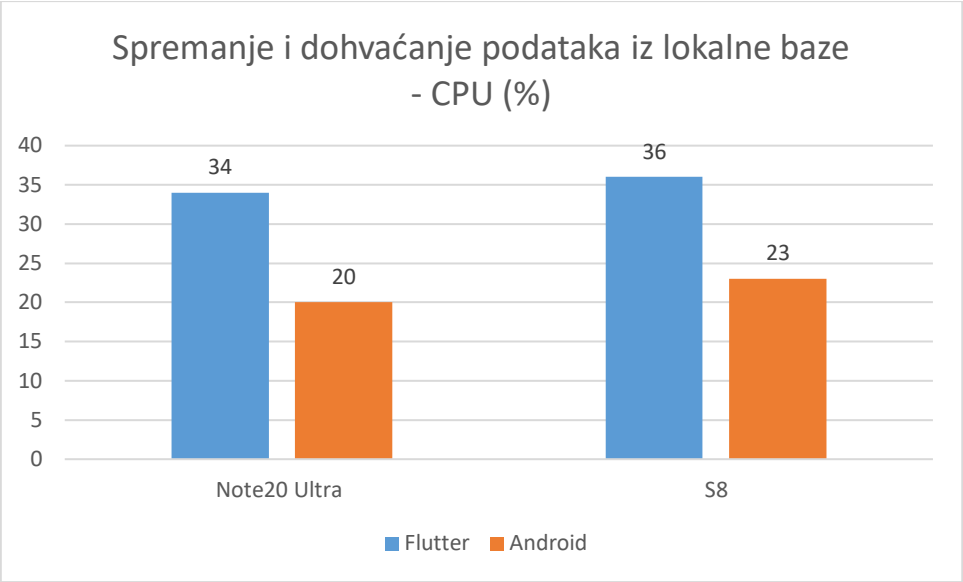
Slika 5.6 Zauzeće memorije pri preuzimanju slika s mreže

Isto tako može se primijetiti da vrijeme potrebno za dohvaćanje slika s API-ja je gotovo jednako u obje platforme, sa malom razlikom od 1 sekunde na starije Galaxy S8 uređaju.

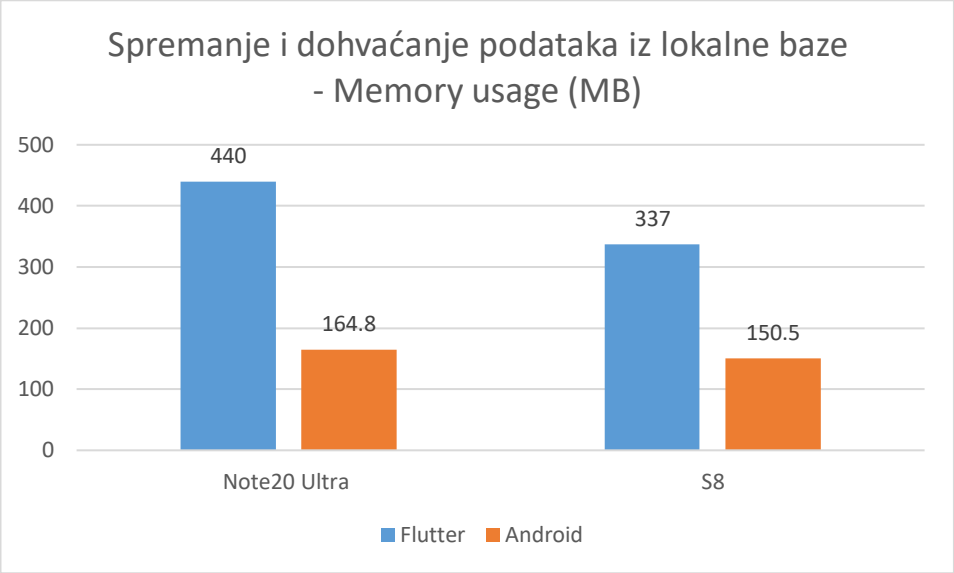


Slika 5.7 Vrijeme odziva pri preuzimanju slika s mreže

Rezultati spremanja slika u lokalnu bazu podataka podosta se razlikuju po platformama. Vidljivo je na slici 5.8, da Flutter troši barem duplo više resursa u obliku memorijskog zauzeća i korištenja procesora u odnosno na nativnu aplikaciju. Isto se može primjetiti da ne postoji velika razlika u ovim rezultatima ako se uspoređuju uređaji.

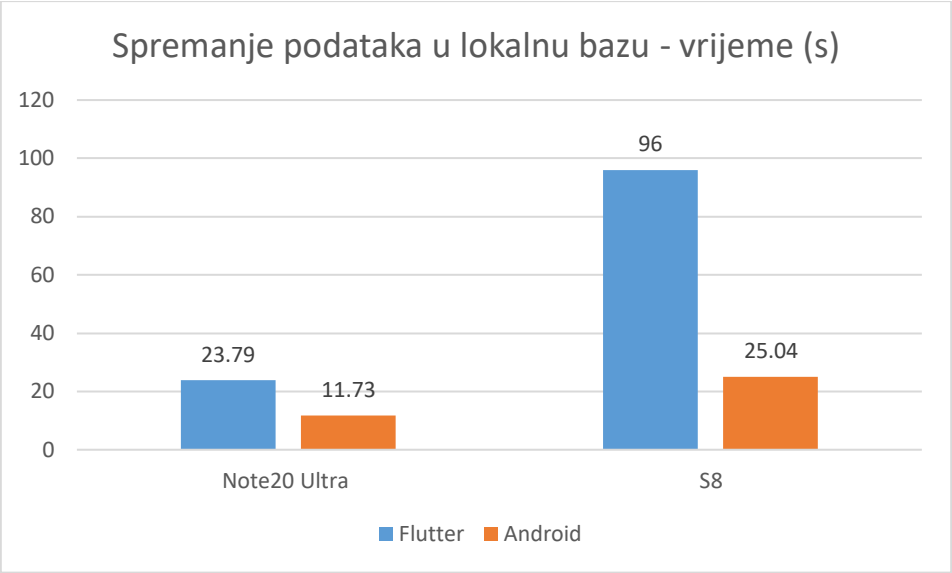


Slika 5.8 Prosječno opterećenje procesora pri spremanju i dohvaćanju podataka iz lokalne baze



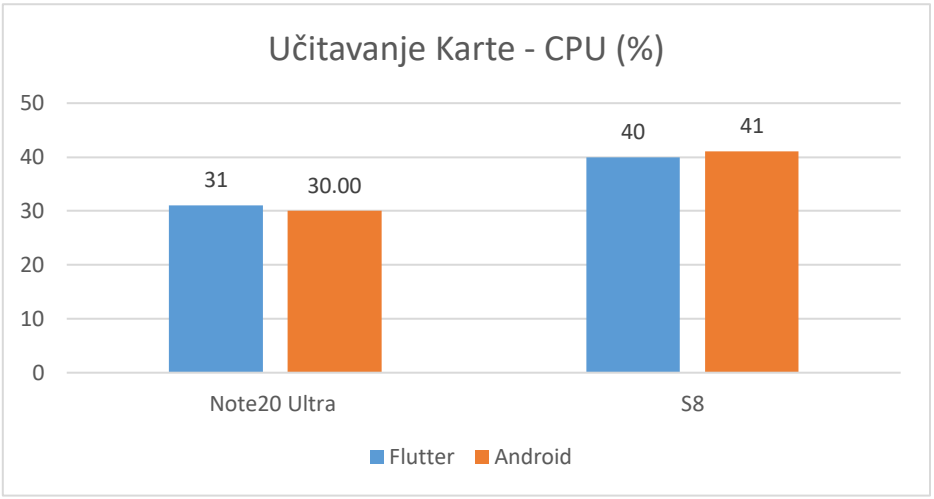
Slika 5.9 Zauzeće memorije pri spremanju i dohvaćanju podataka iz lokalne baze

Flutter-u treba duplo duže vremena da spremi podatke u lokalnu bazu podatku koristeći *drift* paket koji pruža apstrakciju nad *SQLite* motorom. Još je veća razlika na starijem uređaju za koji paket koji se koristi nije dovoljno optimiziran u kojem je razlika gotovo 4 puta veća. 25 sekundi na Android-u u usporedbi sa 96 sekundi predstavlja veliku razliku za korisničko iskustvo.



Slika 5.10 Vrijeme odziva spremanja podataka u lokalnu bazu

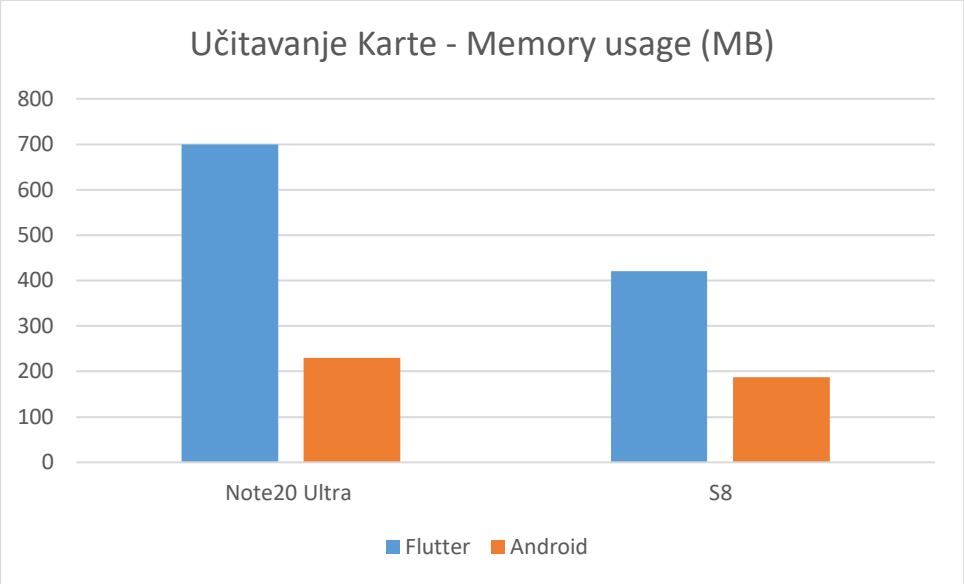
Pri učitavanju *Google Maps* karte prosječan postotak korištenja procesora je sličan na obje platforme i iznosi oko 30% na Note 20 Ultra i 40% na Galaxy S8. Rezultati se mogu vidjeti na slici 5.11.



Slika 5.11 Prosječno korištenje procesora pri učitavanju karte

Zauzeće memorije pri učitavanju je duplo veće na starijem Galaxy S8 uređaju dok je na Note 20 Ultra skoro 4 puta veće. Rezultati su vidljivi na slici 5.12. Google karta je memorijski zahtjevan *widget* te alocira puno memorije za svoje učitavanje. Iako biblioteka korištena u *Flutteru* koristi kod specifičan za Android platformu koji se onda preko klijenta može uključiti u *Flutter* kod,

Flutter generalno u prosjeku zauzima nešto više memorije pri kreiranju zaslona i elemenata pa je to tako i u ovom slučaju.



Slika 5.12 Zauzeće memorije pri učitavanju karte

5.2. Analiza rezultata

Rezultati dobiveni na temelju mjerenja revolucionarnim alatom razvojnog okruženja kazuju da u svim slučajevima nativna aplikacija alocira manje memorije pri izvršavanju zadataka kao što je upisivanje u bazu, dohvaćanje s poslužitelja i slično. Razlog tome je što *Flutter* ima dosta varijabli i elemenata koji su nužni za prikaz korisničkog sučelja i povećavaju memorijsku popunjenost.

Vrijeme odziva pri dohvaćanju podataka s mreže dosta ovisi o uređaju pa tako na starijim Android uređajima, vrijeme za dohvaćanje podataka je znatno veće u *Flutter* aplikaciji, nego u Android. Međutim, na novijim uređajima razlika u vremenu dohvaćanja gotovo i da nema. Uzrok tome može biti drukčija implementacija biblioteke za dohvaćanje u *Flutter* ili native koja je u slučaju ovog rada *Retrofit* ovisno o verziji Androida. Vrijeme odziva za spremanje i dohvaćanje podataka iz lokalne baze može se jasno odrediti i ukazuje na to da je u *Flutter* aplikaciji taj proces puno sporiji neovisno o uređaju.

U izvršenim testovima postotak zauzeća procesora ovisi o radnji koja se obavlja. Tako za preuzimanje slika s poslužitelja korištenje procesora je za 25% manje na novijem uređaju i oko 20% manje na starijem kada se pokreće *Flutter* aplikacija. Međutim, za radnju spremanja i dohvaćanja podataka iz lokalne baze situacija se mijenja jer potrošnja procesora je otprilike u

sličnim postotcima veća u *Flutter* aplikaciji. U ovom slučaju može se zaključiti da korištenje procesora ovisi o vanjskim bibliotekama čija implementacija u ova dva različita *frameworka* može biti drukčija po kompleksnosti koda, način na koji procesira podatke, efikasnost koda i ostalo.

Veličina aplikacije pri instaliranju na uređaj je neusporedivo veća za *Flutter* instaliranu datoteku i to za velikih 87% u odnosu na nativnu aplikaciju. *Flutter* je *framework* koji ima ugrađene neke funkcionalnosti, primjerice, animacije, multimedijske datoteke i slično, ali cijena toga je više dodatnog koda što povećava veličinu aplikacije. Isto tako *Dart* je jezik koji ima svoje biblioteke sa raznim funkcionalnostima koje povećavaju veličinu aplikacije.

Vrijeme potrebno za pokretanje aplikacije vidljivo je na slikama ___ i __ te u prosjeku je tri puta duže potrebno da se pokrene *Flutter* aplikacija u odnosu na nativnu. Razlog tome je da je *Flutteru* potrebno neko vrijeme da učita i konfigurira *DartVM* koji je odgovoran za izvršavanje koda *Dart* programskog jezika, ali i vrijeme koje potrebno da se *widgeti* inicijaliziraju pri otvaranu aplikacije.

5.3. Analiza korisničkog iskustva i subjektivni doživljaj

Vrijeme pokretanja aplikacije do 1 sekunde neće se primijetiti pri korištenju od strane krajnjeg korisnika, ali ako je to vrijeme veće utjecat će negativno na korisničko iskustvo, što je češći slučaj na starijim uređajima. Takav primjer vidi se na Samsung Galaxy S8 uređaju gdje je to vrijeme skoro jednu i pol sekundu i otvaranje prvog zaslona aplikacije je dosta usporeno. Tu možemo vidjeti razliku u performansama native i *Flutter* aplikacije.

Vremena odziva do 100 milisekundi se dožive trenutačnim sa strane korisnika, dok su vremena odziva do par sekundi dozvoljena ako se događaju rijetko [11]. Uzimajući to u obzir, možemo vrednovati u kojim slučajevima će razlike u performansama *Flutter* i nativne aplikacije biti opaženi.

Vremena odziva pri dohvaćanju podataka slika sa poslužitelja je slično na obje platforme i ne utječe na iskustvo korisnika jer iako je u pitanju veliki skup podataka dohvaćanje se odvija veoma brzo. Međutim, kod spremanja podataka u kojem je to vrijeme od više sekundi na nativnom uređaju neće znatno utjecati na ponašanje korisnika jer vidi da je spremanje u procesu zbog prikazanog elementa učitavanja, ali u slučaju *Flutter*-a kad je to vrijeme iznad jedne minute korisnik može pomisliti da se dogodila neka greška i da se taj proces nikad neće dovršiti što je nepoželjan doživljaj za korisnika.

Može se primijetiti da FPS (eng. *Frames per second*) je sličan na oba dva *frameworka*, barem što se može primijetiti kada se prolazi kroz listu slika velikom brzinom na oba dva uređaja. Ne može

se primijetiti nikakvo zastajkivanje ili u još gore pucanje aplikacije stoga to ukazuje da su obje platforme dosta brze pri prikazivanju elemenata i ne uzrokuju nikakve probleme za korisnika.

Subjektivni doživljaj kreiranja ovih aplikacija pomoću Android native *frameworka* i *Flutter*-a je podjednako izazovan, ali jednostavan. Obzirom da je korišten deklarativni način prikazivanja elemenata na zaslonu korištenjem na obje platforme, što je na Androidu kombinacija novog *Jetpack Compose* modernog skupa alata i *ViewModel*-a, a u *Flutter* standardni način kreiranja stabla elemenata i *Riverpod*-a koji drži stanje, implementiranje se odvijalo na jako sličan način. Obzirom da se prije nikad nije radilo s *Jetpack Compose* prvo je kreirana *Flutter* aplikacija, a onda u vrlo kratkom roku je naučen i *Jetpack Compose* te primijenjen za implementaciju u Android-u. Vrijeme potrebno za implementaciju je bilo manje za Android, ali treba uzeti u obzir da su se funkcionalnosti i logika prvo pisali za *Flutter* tako da se može reći da su vremenski implementacije dosta slične te iskustvo programera pri pisanju jer su biblioteke korištene slične sintakse i način definiranja elemenata i arhitektura projekta je isti. Ako se uzme u obzir da *Flutter* aplikacija se može pokretati i na iOS, Web-u i desktop aplikacijama, za ovaj ogledni primjer aplikacije resursno je isplativije od nativnog Android-a, ali situacija se svakako mijenja kad su potrebni kompleksniji elementi i funkcionalnosti ovisni o operacijskom sustavu, gdje se vrijeme potrebno za istraživanje i traženje rješenja može uvelike povećati i u tom slučaju bi ovaj zaključak bio nevažeći.

6. ZAKLJUČAK

Cilj ovog rada bio je istražiti mogućnosti, izazove i moguće probleme pri implementaciji jednostavnog primjera aplikacije sa nekoliko funkcionalnosti za Android operacijski sustav koristeći nativni Android razvoj koristeći *Kotlin* programski jezik te *Flutter* za višeplatformski razvoj. Nakon implementacije provedena je analiza performansi razvijenih projekata te su provedeni testovi od strane programera i korisnika kako bi se dobila i subjektivna mišljena o pogodnostima i pristupu ova dva načina implementacije. Implementacija se odnosila na funkcionalne i nefunkcionalne zahtjeve s naglasak na nefunkcionalne i testiranju performansi.

Analiziranjem dobivenih rezultata dolazi se do zaključka da aplikacije izrađene u *Flutteru* pružaju mnoge pogodnosti kao što je pisanje jednog programskog koda koji se može pokretati na više platformi, a u slučaju ovog rada na Android-u.Taj benefit nosi i izazove kao što su slabije performanse koje utječu na korisničko iskustvo, ali i često u nekim kompleksnijim slučajevi i nemogućnost implementacije ili jako teško implementiranje nekih funkcionalnosti ako se pristupa elementima specifičnih za platformu. Prema ovim testovima oba dva načina razvoja ispunjavaju potrebe i očekivanja korisnika i programera, ali ne može se potvrditi da je bolje koristiti jedno rješenje u odnosu na drugo. Sugestija je da se izabere nativna Android aplikacija ako ona ima velike zahtjeve za performansama kao što su brzina pokretanja, spremanja podataka, manja alokacija memorije i slično. Višeplatformska *Flutter* aplikacija je puno bolje rješenje ako je potreban brzi razvoj, performanse nisu prioritet, iskorištavanje koda za više platformi i smanjenje potrebnih resursa i drugih troškova te nije potrebno pristupati funkcionalnostima specifičnih za platformu. Ono što je velika prednost *Flutter* nasuprot drugim višeplatformski okvirima je to što postoji mogućnost uključivanja nativne logike u *Flutter* ekosustav kroz API-je. Nativni razvoj će uvijek imati podršku tvrtka koje rade na platformi, a svi ostali alati i okviri kao što je *Flutter* će kasniti sa implementacijom novih mogućnosti. Obje platforme imaju podršku od tvrtke Google koja ih je kreirala i nastavit će se razvijati u budućnosti tako da je moguće da će određeni izazovi biti zaboravljeni te da će ovi testovi davati nešto drukčije rezultate. Prema navedenom pobjednika ove analize nema, nego izbor ovisi o poslovnom modelu, zahtjevima i dostupnim resursima kojima se raspolaže.

LITERATURA

[1] »The State of Mobile App Market + Predictions for 2023 & Beyond,« Udonis, 9 veljača 2023. [Mrežno]. Available: https://www.blog.udonis.co/mobile-marketing/mobile-apps/mobile-app-market-forecast. [Pokušaj pristupa 20 travanj 2023].

[2] P. Que, X. Guo i M. Zhu, »A Comprehensive Comparison between Hybrid and Native App Paradigms,« u *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2016.

[3] S. Watts, »Application Developer Roles and Responsibilities,« BMC Software, 19 listopad 2018. [Mrežno]. Available: https://www.bmc.com/blogs/application-developer-roles-responsibilities/. [Pokušaj pristupa 20 travanj 2023].

[4] Purvi, »How Business Get Successful With Help of Cross-Platform Application Development?,« Prismetric Technologies , 15 studeni 2021. [Mrežno]. Available: https://www.prismetric.com/how-cross-platform-app-development-boost-business-success/. [Pokušaj pristupa 20 travanj 2023].

[5] Google Inc, »Google Pay,« Google Inc, [Mrežno]. Available: https://flutter.dev/showcase/google-pay. [Pokušaj pristupa 20 travanj 2023].

[6] T. Mai, »Top 7 best Native App Example in 2023 that Merchants can learn from,« Magenest JSC, 30 studeni 2021. [Mrežno]. Available: https://magenest.com/en/native-app-example/. [Pokušaj pristupa 20 travanj 2023].

[7] A. Marchuk, »Native vs Cross-Platform Development: How to Choose,« UPTECH, [Mrežno]. Available: https://www.uptech.team/blog/native-vs-cross-platform-app-development. [Pokušaj pristupa 20 travanj 2023].

[8] S. J. Bigelow, »platform,« TechTarget, [Mrežno]. Available: https://www.techtarget.com/searchitoperations/definition/platform. [Pokušaj pristupa 20 travanj 2023].

[9] »Software Ecosystem,« Ingram Micro, [Mrežno]. Available: https://www.cloudblue.com/glossary/software-ecosystem/. [Pokušaj pristupa 20 travanj 2023].

[10] »What is JVM (Java Virtual Machine): Architecture Explained!,« Guru99, 4 ožujak 2023. [Mrežno]. Available: https://www.guru99.com/java-virtual-machine-jvm.html. [Pokušaj pristupa 20 travanj 2023].

[11] »What is cross-platform mobile development?,« Kotlin Foundation, [Mrežno]. Available: https://kotlinlang.org/docs/cross-platform-mobile-development.html. [Pokušaj pristupa 20 travanj 2023].

[12] M. Dabbagh i S. P. Lee, »An Approach for Integrating the Prioritization of Functional and Nonfunctional Requirements,« *The Scientific World Journal*, p. 13, 2014.

[13] T. P. Bowen, G. B. Wigle i J. T. Tsai, »Specification of Software Quality Attributes. Volume 1. Final Report,« BOEING AEROSPACE, Seattle,WA,USA, 1985.

[14] Google Inc, »Profile your app performance,« Google Inc, [Mrežno]. Available: https://developer.android.com/studio/profile. [Pokušaj pristupa 20 travanj 2023].

[15] »DevTools,« Google Inc, [Mrežno]. Available: https://docs.flutter.dev/tools/devtools/overview. [Pokušaj pristupa 20 travanj 2023].

[16] Google Inc, »Firebase Performance Monitoring,« Google Inc, [Mrežno]. Available: https://firebase.google.com/docs/perf-mon. [Pokušaj pristupa 20 travanj 2023].

[17] Storemaven, »App Size,« Storemaven, [Mrežno]. Available: https://www.storemaven.com/glossary/app-size-definition/. [Pokušaj pristupa 20 travanj 2023].

[18] A. Nitze i A. Schmietendorf, »Universal Performance Assessment of Cross-Platform Mobile Applications,« u *31st UK Performance Engineering Workshop*, Leeds, UK, 2015.

[19] »Mobile Operating System Market Share Worldwide,« StatCounter , [Mrežno]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Pokušaj pristupa 20 travanj 2023].

[20] A. Turner, »Google Play App Statistics,« BankMyCell, travanj 2023. [Mrežno]. Available: https://www.bankmycell.com/blog/number-of-google-play-store-apps/. [Pokušaj pristupa 20 travanj 2023].

[21] Kotlin Foundation, »Kotlin for Android,« Kotlin Foundation, [Mrežno]. Available: https://kotlinlang.org/docs/android-overview.html. [Pokušaj pristupa 20 travanj 2023].

[22] »Android Architecture – Detailed Explanation,« InterviewBit, 3 lipanj 2022. [Mrežno]. Available: https://www.interviewbit.com/blog/android-architecture. [Pokušaj pristupa 20 travanj 2023].

[23] Google Inc, »Meet Android Studio,« Google Inc, [Mrežno]. Available: https://developer.android.com/studio/intro. [Pokušaj pristupa 20 travanj 2023].

[24] Google Inc, »Thinking in Compose,« Google Inc, [Mrežno]. Available: https://developer.android.com/jetpack/compose/mental-model. [Pokušaj pristupa 20 travanj 2023].

[25] Google Inc, »Flutter - build apps for any screen,« Google Inc, [Mrežno]. Available: https://flutter.dev/. [Pokušaj pristupa 20 travanj 2023].

[26] Google Inc, »Dart overview,« Google Inc, [Mrežno]. Available: https://dart.dev/overview. [Pokušaj pristupa 20 travanj 2023].

[27] Google Inc, »Flutter architectural overview,« Google Inc, [Mrežno]. Available: https://docs.flutter.dev/resources/architectural-overview. [Pokušaj pristupa 20 travanj 2023].

[28] »Flutter Project Structure: Feature-first or Layer-first?,« Coding With Flutter , 22 ožujak 2022. [Mrežno]. Available: https://codewithandrea.com/articles/flutter-project-structure/. [Pokušaj pristupa 20 travanj 2023].

[29] Google Inc, »Guide to app architecture,« Google Inc, [Mrežno]. Available: https://developer.android.com/topic/architecture. [Pokušaj pristupa 20 travanj 2023].

[30] Google Inc, »Google Maps Platform FAQ,« Google Inc, [Mrežno]. Available: https://developers.google.com/maps/faq. [Pokušaj pristupa 20 travanj 2023].

[31] Google Inc, »Request location permissions,« Google Inc, [Mrežno]. Available: https://developer.android.com/training/location/permissions. [Pokušaj pristupa 20 travanj 2023].

[32] K. Kluz, »Flutter vs native iOS & Android comparison – an overview of Kotlin, Swift vs Dart,« CrustLab, 13 rujan 2022. [Mrežno]. Available: https://crustlab.com/blog/flutter-ios-android-comparison-overview-kotlin-swift-dart/. [Pokušaj pristupa 20 travanj 2023].

[33] Xilinx, »Embedded System Tools Reference Manual - Embedded Development Kit,« Xilinx, 2008.

[34] Digilent, »Nexys3 Board Reference Manual,« Digilent, Pullman, WA, 2013.

[35] Xilinx, »MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 10.1i,« Xilinx, 2008.

[36] P. Marwedel, Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, Springer Netherlands, 2011.

[37] J. O. Hamblen, T. S. Hall i M. D. Furman, Rapid Prototyping of Digital Systems - SOPC Edition, Springer US, 2008.

SAŽETAK

Trenutno jedna od najrelevantnijih tema koja se provlači kroz polje mobilnih tehnologija u svijetu IT-a je koja tehnologija je najnaprednija i može donijeti najbolje poslovne rezultate.

U ovom radu istraživane su mogućnosti i problemi višeplatformskog i nativnog razvoja mobilnih aplikacija, alati, te programska arhitektura i načina testiranja performansi aplikacija. Naglasak je na razvoju Android mobilne aplikacije koristeći Android native i njegov Jetpack Compose skupinu alata za korisničko sučelje te Flutter za višeplatformski razvoj. Nakon definiranja funkcionalnih i nefunkcionalnih zahtjeva aplikacije i implementacije, glavni cilj je testiranje performansi oba dva pristupa i njihov utjecaj na krajnjeg korisnika te subjektivni doživljaj programera tijekom razvoja. Nakon obavljenog testiranja performansi, ali i funkcionalnosti koristeći alate razvojnog okruženja, provedena je analiza dobivenih rezultata te zaključak o tome koji pristup je prikladniji za određeni projekt i tvrtku koja mora odlučiti koju tehnologiju će koristiti. Rezultati su pokazali da je najpogodnije koristiti nativni razvoj ako su potrebne visoke performanse i kompleksni korisnički zahtjevi, a Flutter ako nema puno raspoloživih resursa i potrebno je dijeljenje koda.

Ključne riječi: Android, performanse, mobilna aplikacija, testiranje, nativni razvoj, višeplatformski razvoj.

ABSTRACT

Comparison of Flutter and Android native applications

Currently, one of the most relevant topics running through the field of mobile technologies in the world of IT is which technology is the most advanced and can bring the best business results. In this paper, the possibilities and problems of multi-platform and native mobile application development, tools, and program architecture and methods of application performance testing were investigated. Emphasis is on Android mobile application development using Android native and its Jetpack Compose suite of user interface tools and Flutter for cross-platform development. After defining the functional and non-functional requirements of the application and implementation, the main goal is to test the performance of both approaches and their impact on the end user and the subjective experience of developers during development. After performing performance and functionality testing using the tools of the development environment, an analysis of the obtained results was carried out and a conclusion was made as to which approach is more suitable for a specific project and company, which must decide which technology to use. The results showed that it is most suitable to use native development if high performance and complex user requirements are required, and Flutter if there are not many available resources and code sharing is required.

Key words: Android, performance, mobile app, testing, native development, cross-platform development

ŽIVOTOPIS

Matej Kovačević rođen je 11. ožujka 1999. godine u Zagrebu. Osnovnu školu je upisao i završio u Osijeku. Nakon završetka osnovne škole upisuje III. Gimnaziju u Osijeku, koju završava 2017. godine. Tokom iste godine upisuje preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.

_____

Potpis autora