Analiza mogućnosti višeplatformskog nativnog razvoja mobilnih aplikacija

Štajcer, Ivan

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

Sveučilišni diplomski studij

ANALIZA MOGUĆNOSTI VIŠEPLATFORMSKOG NATIVNOG RAZVOJA MOBILNIH APLIKACIJA

Diplomski rad

Ivan Štajcer

Osijek, 2022.

SADRŽAJ

1. UVOD

Ovaj rad bavi se usporedbom izrade mobilne iOS aplikacije koristeći noviju višeplatformsku tehnologiju Flutter te izradom identične mobilne aplikacije koristeći najnoviji nativni pristup izrade iOS aplikacija, odnosno *SwiftUI*. Cilj ovog rada je što bolje usporediti performanse višeplatformske i nativne aplikacije te njihov krajnji utjecaj na korisnika, u svrhu boljeg vrednovanja odabira pristupa pri izradi mobilnih aplikacija. Na temelju prikladno odabranih radova testnih mobilnih aplikacija, za ostvareno nativno i višeplatformsko programsko rješenje provest će se testiranje performansi i osvrnuti se na rezultate dobivene u drugim radovima. Usporedit će se dobiveni rezultati performansi višeplatformske i nativne aplikacije i analizirati njihov utjecaj na korisnika i na odabir prikladnog pristupa od strane programera.

U drugom poglavlju govori se o problemima pri izradi mobilnih aplikacija sa stajališta programera, klijenta i korisnika, dan je pregled trenutnog stanja mobilnih aplikacija te primjer napravljenih višeplatformskih i nativnih mobilnih aplikacija. Treće poglavlje opisuje funkcijske i nefunkcijske zahtjeve aplikacije, daje prikaz građe aplikacije te opis testnog okruženja i parametara testiranja. U četvrtom poglavlju će se opisati korištene tehnologije pri izradi aplikacija i implementacije nativne i višeplatformske aplikacije. U petom poglavlju dat će se pregled rezultata testiranja, analiza rezultata i korisničkog iskustva.

2. PRIKAZ STANJA U PODRUČJU I IZAZOVI U RAZVOJU MOBILNIH APLIKACIJA

2.1. Izazovi u razvoju mobilnih aplikacije

Tijekom stvaranja mobilnih aplikacija postoje brojni izazovi koji se razlikuju od aplikacije do aplikacije radi zahtijeva same poslovne logike pojedine aplikacije. Zahtjevi pri izradi mobilnih aplikacija ne proizlaze samo iz potrebne funkcionalnosti koje aplikacija treba izvodit, potrebno je implementirati ono što je korisniku najvažnije.

Osim problema koji aplikacija rješava, korisnici žele imati ugodan izgled aplikacije s lijepim animacijama, dobrim korisničkim iskustvom te intuitivnim korištenjem aplikacije. Također, korisnik očekuje dobre performanse aplikacije u pogledu da ona vizualno ne zastajkuje te da ako obavlja neke proračune, obavlja ih što brže, po mogućnosti odmah. Korisniku je više bitno korisničko iskustvo te problem koji rješava aplikacija nego kako je ona napravljena [1].

Kako se funkcionalnosti povećavaju, sve je teže održati ove kriterije i zadovoljiti korisnika u svim pogledima. No, treba držati na umu da to korisnika ne bi trebalo ni zanimati te da je na kraju posao tima koji radi na proizvodu implementirati zadane funkcionalnosti i ispuniti očekivanja. Na koncu, ako se neki proizvod predstavlja kao dobro rješenje, onda se toga treba i držati u svakom aspektu.

Programer kao član tog tima, ima bitnu ulogu. Dizajner može napraviti izvrsno korisničko sučelje s dobrim animacijama i vrlo intuitivnim korisničkim iskustvom. Klijent može napraviti funkcionalnosti koje pomažu svim korisnicima i mogu biti potpuno uvjereni da će korisnici biti oduševljeni s rješenjem problema kojeg aplikacija rješava. No, ako programer to ne može spojiti u zamišljenu cjelinu, ne može implementirati neke animacije za koje se zahtijeva da se obave programski ili njegov algoritam ne može ispuniti očekivanja brzine izračuna, korisnici neće biti zadovoljni s proizvodom.

Osim vještina i koordiniranja tima pri izradi proizvoda, potrebno je prije svega izabrati alat za izradu mobilne aplikacije. Ako se radi o aplikaciji za samo jednu platformu, ima smisla raditi nativnu aplikaciju u programskom jeziku Swift za iOS ili Kotlin za Android. Ako aplikacija treba biti pokretana na Android i na iOS uređajima, moguća su dva izbora, a to su nativni razvoj ili višeplatformski razvoj. Nažalost, pri nativnom razvoju potrebne su dvije različite implementacije za istu aplikaciju. Ovaj problem rješavaju višeplatformske razvojne okoline kao što je Flutter, koji omogućuje korištenje jedne implementacije aplikacije na više platformi.

Pri višeplatformskom razvoju smanjuju se troškovi i vrijeme razvoja mobilne aplikacije te je potrebno manje ljudi angažirati za projekt. Međutim, višeplatformski razvoj ima slabije performanse, povećanu veličinu aplikacije te ovisi o više vanjskih biblioteka. Pri donošenju odluke za odabir nativnog ili višeplatformskog razvoja, potrebno je imati uvid u situacije u kojima je moguće koristiti višeplatformski pristup izrade mobilnih aplikacija, čime se smanjuju troškovi i vrijeme te situacije gdje je nativni pristup s većim performansama bolji odabir.

2.2. Trenutno stanje razvoja mobilnih aplikacija

Mobilne aplikacije konstantno se razvijaju i potrebna je stalna inovativnost, novi pristupi pri izradi te bolja kvaliteta krajnjih proizvoda. Vrijeme provedeno na aplikacijama na mobilnim uređajima raste iz godine u godinu. Primijećen je stalni porast objavljivanja aplikacija na trgovinama, tako u 2021. godini je objavljeno preko dva milijuna aplikacija [2].

Broj skidanja aplikacija s trgovina također raste, tako je do sada u prvoj četvrtini 2022. godine, ukupan broj skidanja dosegao iznos od 36.9 bilijuna. To predstavlja povećanje za 1.4% na 2021. godinu [3].

2.1. Primjeri aplikacija napravljenih u Flutteru

Svake godine izlazi sve veći broj izvrsno napravljenih aplikacija. Dobar primjer aplikacije napravljene koristeći *Flutter* je Aplikacija *Google Ads* od tvrtke Google [4]. Google Ads pomaže pratiti korisničke kampanje oglašavanja preko Google korisničkog računa.

Aplikacija *Reflectly* je još jedan primjer aplikacije napravljene u *Flutteru* [5]. S visokom ocjenom od 4.6 na Apple Store trgovini, ova aplikacija napravljena koristeći *Flutter* služi za pisanje nota, kratkih poruka te pruža korisniku priliku da podijeli kako se trenutno osjeća.

2.2. Primjeri aplikacija napravljenih u *SwiftUI*

Aplikacija *Weather* od tvrtke Apple u svojem novijem izdanju koristi *SwiftUI* pri izgradi korisničkog sučelja [6]. Aplikacija *Weather* ima ocjenu 3.2 na App Store trgovini te preko četiri tisuće recenzija.

Aplikacija *Corona Virus Stats & Advices* je aplikacija napravljena koristeći SwiftUI, aplikacija praćenje najnovijih podataka o statusu trenutne pandemije [7]. Osim statistika, pruža korisniku korisne savjete o tome kako zaštiti svoje zdravlje.

3. MODEL I GRAĐA MOBILNE APLIKACIJE

U ovom poglavlju dani su funkcijski i ne funkcijski zahtjevu aplikacije, pregled izgleda aplikacije te opis alata korištenih za testiranje aplikacija.

3.1. Funkcijski i nefunkcijski zahtjevi na aplikaciji

Funkcijski zahtjevi mobilne aplikacije opisuju potrebne funkcionalnosti koje aplikacija treba implementirati te kakve radnje korisnik treba obaviti kako bi koristio aplikaciju. Nefunkcijski zahtjevu na aplikaciju opisuju kakva svojstva rješenja za implementiranje aplikacije imaju, kako će aplikacija raditi i zašto [8].

3.1.1. Funkcijski zahtjevi

Funkcijski zahtjevi uključuju zahtjeve poslovne logike unutar aplikacije, korisničke zahtjeve, administrativne zahtjeve, zahtjeve pri autorizaciji korisnika, vanjske zahtjeve aplikacije i slično. Funkcijski zahtjevu se dijele na slučajeve upotrebe, pri čemu se svaki slučaj dijeli na sudionike, funkcionalnost i cilj slučaja. [9]. Primjeri čestih funkcijski zahtijeva mobilnih aplikacija su prijava i odjava korisnika iz aplikacije, uvid u statistiku korištenja aplikacije i grešaka, listanje kroz brojne elemente na prikazu, plaćanje putem interneta i virtualne košarice, navigacija, prikaz slika, lokalno spremanje podataka, rad s formama i slično [10].

Uzimajući u obzir najčešće funkcijske zahtjeve, implementirana aplikacija treba omogućiti sljedeće:

- Korisnik ima pregled kvalitetnih slika pomoću liste, te može listati kroz listu slika kako bi mogao odabrati sliku koja mu se sviđa.
- Korisnik može, klikom na sliku, navigirati na prikaz detalja slike te vidjeti nešto više na slici.
- Korisnik može primijeniti filter na sliku, kako bi promijenio sliku prema svojoj želji.
- Korisnik može spremiti filtriranu sliku u galeriju na mobilnom uređaju kako bi je sačuvao.

3.1.2. Nefunkcijski zahtjevi

Primjeri nefunkcijskih zahtjeva mobilnih aplikacija uključuju performanse u pogledu vremena trajanja izvršenja nekog procesa, pokretanja aplikacije, korištenje memorije, veličina aplikacije i

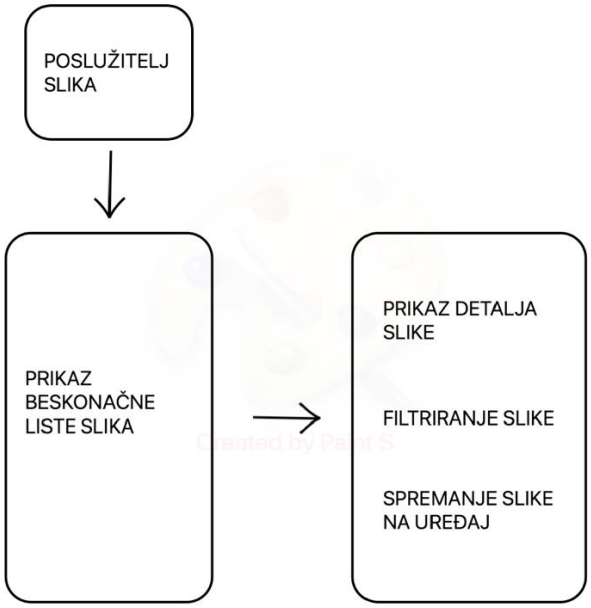
slično. Nefunkcijski zahtjevi uključuju skalabilnost, kao mogućnost primanja više korisnika ili rada sa više podataka. Uključuju pouzdanost, kao zahtijeva da se sa svakim korištenjem aplikacije dobije isti rezultat pri određenim radnjama. Nefunkcijski zahtjevi također uključuju brzi odziv povratnih informacija prema korisniku [10].

S obzirom na česte nefunkcijske zahtjeve mobilnih aplikacija, implementirana aplikacija bi trebala zadovoljiti:

- Kada korisnik otvori aplikaciju treba vidjeti prikaz slika na zaslonu unutar pet sekundi
- Aplikacija mora prikazati svaku promjenu na zaslonu bez vidnog zastajkivanja
- Aplikacija mora izgledati funkcionalno na uređajima različitih veličina zaslona
- Kada korisnik spremi filtriranu sliku na uređaj, mora dobiti povratu informaciju o uspješnom spremanju slike
- Aplikacija mora obaviti spremanje filtrirane slike na uređaj unutar tri sekunde
- Kada korisnik klikne na sliku u listi slika, aplikacija uvijek mora navigirati na prikaz detalja slike koju je korisnik pritisnuo

3.2. Građa mobilne aplikacije

Implementirana mobilna aplikacija sastoji se od početnog zaslona koji prikazuje beskonačnu listu slika dohvaćenih s poslužitelja (https://unsplash.com/) i zaslon prikaza detalja slike. Prikaz okoline rada i građe izrađene aplikacije je vidljiv na slici 3.1.



Sl. 3.1. Građa implementirane mobilne aplikacije

Pojedinačni element unutar liste prikazuje sliku dobivenu s poslužitelja, broj pozitivnih reakcija korisnika na sliku te korisničko ime autora slike. Klikom na sliku unutar liste, aplikacije prikazuje stranicu s detaljima slike koju je korisnik pritisnuo.

Na zaslonu detalja slike prikazana je prethodno pritisnuta slika s informacija vezanim za sliku, kao što su broj pozitivnih reakcija korisnika na sliku, profilna slika autora slike, korisničko ime autora slike te tekstualni opis slike od strane autora slike. Prikazan je gumb pomoću kojeg korisnik primjenjuje filter na odabranu sliku te se filtrirana slika prikazuje na mjestu originalne slike. Na zaslonu je također prikazan gumb pomoću kojeg korisnik može spremiti filtriranu sliku u galeriju slika na mobilnom uređaju.

3.3. Testiranje u višeplatformskom razvoju mobilnih aplikacija

3.3.1. Testno okruženje

Testiranje je obavljeno koristeći iste alate za testiranje višeplatformske i nativne aplikacije, kako bi se dobili vjerodostojni rezultati. Za testiranje korišteni su testni alati *XCode* razvojnog okruženja, kao i u [11] te alat Apptim. Apptim je alat za mjerenje performansi mobilnih aplikacija koji omogućava vrednovanje performansi i korisničkog iskustva mobilnih aplikacija [12]. Apptim interno koristi teste alate *XCode* razvojnog okruženja, s tim da pruža dobro korisničko sučelje.

Testiranje je poželjno obaviti na što više mobilnih uređaja, kao što je napravljeno u [11] i [13]. Testiranje aplikacije se obavilo na dva mobilna uređaja iPhone 12 i iPhone 13 Pro Max, čije su specifikacije vidljive na tablici 3.1.

Tablica 3.1. Obilježja testnih uređaja

	iPhone 12	iPhone 13 Pro Max
iOS verzija	15.5	15.4
model	iPhone 13,12	iPhone 14,3
Procesor	Apple A14 Bionic arm64e	Apple A15 Bionic arm64e
Broj jezgri	6	6
RAM	4GB DDR4	6GB DDR4
Rezolucija zaslona	1170x2532	1284x2778
Gustoća prikaza	460	458

3.3.2. Testni slučajevi

Testni slučajevi su provedeni ciljajući bitne stavke s obzirom na ograničene resurse na mobilnim uređajima, zahtjeve mobilnih aplikacije te već provedene testove performansi mobilnih aplikacija.

3.3.2.1. Vrijeme odziva

Vremena odziva radnji unutar aplikacije ispod 100 milisekundi čine se trenutačna s pogleda korisnika. Vremena u trajanju do jedne sekunde ili par sekundi smatraju se dozvoljenima ako se ne događaju često. Sva odzivna vremena iznad toga narušavaju korisničko iskustvo [11]. Uzimajući to u obzir, provedena su mjerenja vremena potrebnog pri pokretanju aplikacije i pri navigaciji na iduću stranicu, kao što je to napravljeno i pod [11]. Također, provedena su mjerenja vremena odziva za primjenjivanje filtera na sliku i spremanja slike na uređaj.

3.3.2.2. Postotak korištenja procesora

Ako mobilna aplikacija koristi preveliki postotak korištenja procesora, postoji šansa da negativno utječe na ostale procese na uređaju [11]. Provedeno je mjerenje postotka korištenja procesora kroz čitavu aplikaciju, kao i u [11] te je provedeno mjerenje postotka korištenja procesora u određenim radnjama aplikacije, kao i u [1], [11], [13] te [14].

3.3.2.3. Količina korištene memorije

Količina korištene memorije odnosi se na količinu RAM memorije koju zauzima mobilna aplikacija. Memorija je posebno bitna pri pokretanju aplikacije na uređajima nižih specifikacija, gdje može doći do loših performansi aplikacije te se performanse same platforme na kojoj se pokreće aplikacije mogu narušiti [11]. Uzimajući u obzir memoriju kao bitnu stavku pri izradi mobilnih aplikacija te provedena mjerenja u [1], [11] i [13], provedeno je mjerenje količine zauzeća memorije pri početnom stanju aplikacije te pri kraju testiranja.

3.3.2.4. Veličina aplikacije

Velike aplikacije manje se instaliraju, više je vjerojatno da će biti izbrisane s uređaja te trebaju više vremena pri skidanju i instalaciji na uređaj. Uzimajući u obzir mjerenja veličine aplikacije u [1], provedena su mjerenja veličine aplikacije pri skidanju i instaliranju na uređaj.

3.3.2.5. Mrežni promet

Bit će obavljeno mjerenje mrežnog prometa aplikacije, kao što je napravljeno u [1]. Aplikacija ima intenzivni mrežni promet s obzirom na to da prikazuje više slika velike kvalitete koje preuzima s poslužitelja. Mrežni promet unutar aplikacije nije uspoređen kao što je memorije ili postotak korištenja procesora, jer ovisi o utjecajima same mreže i uvjetima u trenutku testiranja. Pregled

mrežnog prometa je dan kao faktor kojeg treba uzeti u obzir pri analizi rezultata testiranja, gdje obrađivanje podataka s mreže može biti razlog povećanog udio korištenja procesora.

4. PROGRAMSKO RJEŠENJE NATIVNE I VIŠEPLATFORMSKE APLIKACIJE

U ovom poglavlju opisano je rješenje korišteno kod implementacija zadane aplikacije u višeplatformskom i nativnom okruženju. Značajno je navesti točnu arhitekturu, upravljanje stanjima unutar aplikacije, navigaciju te logiku radi boljeg razumijevanja dobivenih rezultata testiranja.

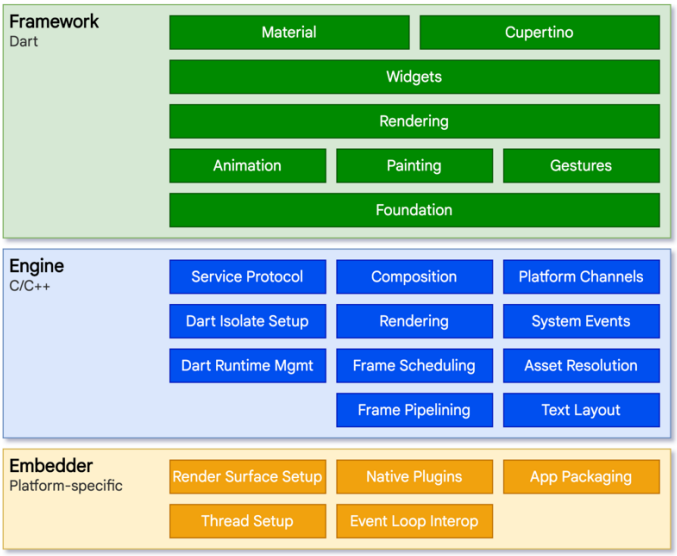
4.1. Korištene tehnologije, alati i jezici

4.1.1. Flutter

Flutter je višeplatformski alat za razvoj programa koji omogućava korištenje istog koda na različitim operacijskim sustavima kao što su iOS i Android, uz dopuštanje aplikacijama da direktno komuniciraju sa uslugaima pojedine platforme [15]. Implementacija alata je otvorena i slobodna za izmjene od strane bilo koje osobe. Cilj alata je pružiti nativni osjećaj pri korištenju pojedinih platformi uz održavanje visokih performansi. Kao programski jezik koristi *Dart*. *Dart* je klijentski optimiran jezik za razvoj aplikacija na bilo kojoj platformi uz pružanje značajki kao što su *type saftey* i *sound null saftey* [16]. Pri višeplatformskom razvoju *Dart* pruža prevođenje koda u strojni kod. Kod nativnih platformi to se postiže pomoću virtualnog stroja koja pruža *just in time* (JIT) prevodilac pri postupku izrade aplikacija, dok u produkcijskom okruženju se koristi „*ahead of time* (AOT) prevodilac kako bi preveo kod u strojni jezik. Za internetske aplikacije koristi se prevoditelj za razvojno (*Dartdevc*) i produkcijsko (*Dart2js*) okruženje kako bi se kod napisan u Dart programskom jeziku preveo u JavaScript.

4.1.1.1. Arhitektura Flutter aplikacije

Flutter je dizajniran kao višeslojni sustav sastavljen od više međusobno neovisnih biblioteka, pri čemu svaka biblioteka ovisi o sloju na kojem se nalazi. Svaki sloj je osmišljen kao nezavisni te ga je moguće totalno zamijeniti s nekim drugim [17].

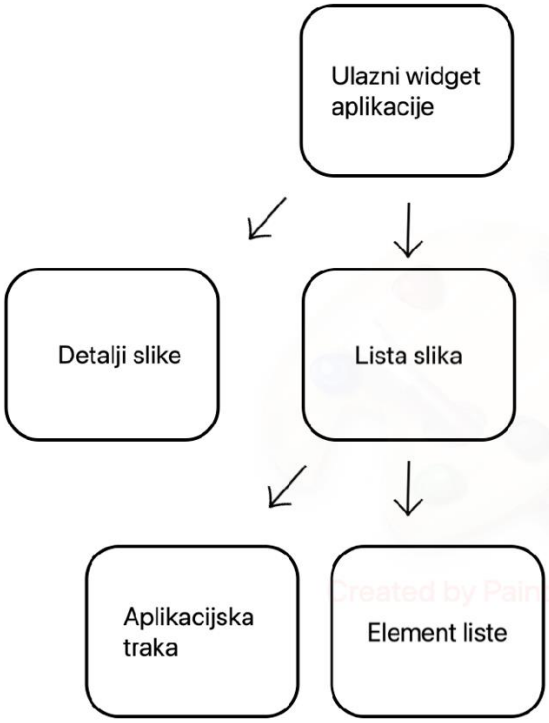


Sl. 4.1. Slojevi arhitekture *Fluttera* [17]

Svaki operacijski sustav komunicira s *Embedder* slojem te se on razlikuje ovisno o kojoj se platformi radi. Njegova svrha je komunikacija s operacijskim sustavom i pakiranje aplikacije tako da je operacijski sustav ne razlikuje od nativne aplikacije. Pri tome, moguće je uvesti *Flutter* aplikaciju unutar već postojeće aplikacije kao modul. Također, on komunicira s *Flutter* strojem napisanom u C/C++ programskom jeziku, koja predstavlja centralni sloj. U centralnom sloju se nalaze implementacije središnjeg programskog sučelja *Fluttera*. Na vrhu sloja se nalazi samo programsko okruženje *Fluttera* koje se koristi pri razvoju aplikacija. Komunikacija sa strojem omogućena je pomoću *Dart:ui* biblioteke sastavljene od *Dart* klasa, koje služe kao adapteri klasama napisanim u C/C++ programskom jeziku iz samog stroja.

4.1.1.2. Widgeti i trenutno stanje aplikacije

Flutter čitavo korisničko sučelje gradi pomoću *widgeta*. *Widget* predstavlja osnovnu jedinicu za izgradnju korisničkog sučelja te je implementiran kao nepromjenjiva klasa. *Widget* pomoću *build* metode opisuje kako trenutno izgleda korisničko sučelje s obzirom na trenutno stanje unutar aplikacije [18]. Više različitih *widgeta* formiraju sučelje koristeći kompoziciju, koja time reflektira trenutnu trenutno stanje aplikacije. Primjer hijerarhije widgeta prikazan je na slici 4.2.



Sl. 4.2. Hijerarhija *widgeta*

Svaki *widget* od roditelja dobiva *context*, preko kojeg prima informacije o tome gdje se nalazi unutar formiranog stabla. Prilikom promjene stanja, kao na primjer pri reakciji na interakciju korisnika s aplikacijom, pojedini *widget* unutar hijerarhije se može zamijeniti s drugim te time reflektirati promijenjeno stanje unutar aplikacije.

Postoje dvije vrste widgeta:

- *Stateless widget*
- *Stateful widget*

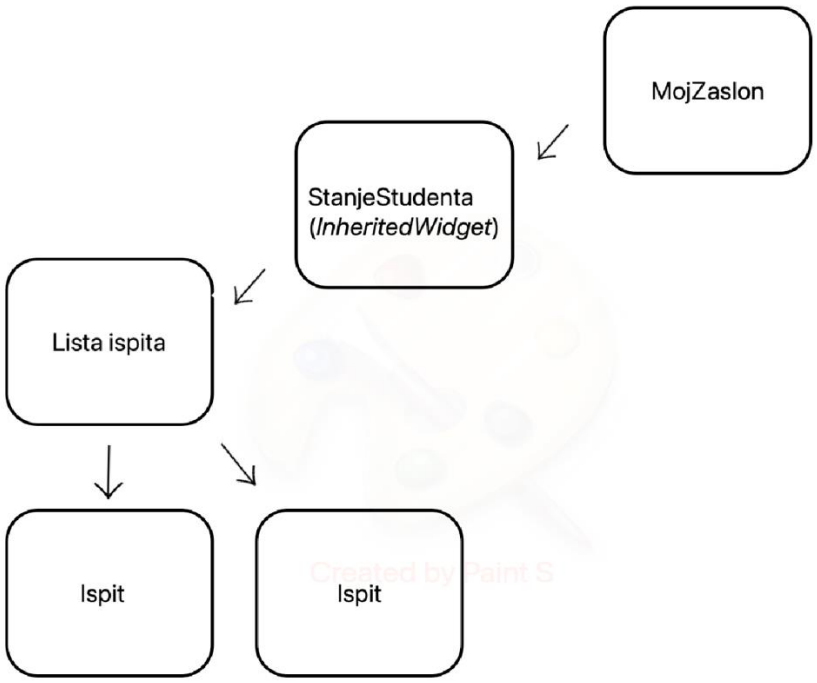
Stateless widget ne sadrži nikakvo stanje koje je podložno promjenama. Kao na primjer *Text widget* koji služi za prikazivanje teksta na zaslon. *Stateful widget* sadrži jedan ili više atributa koji su podložni promjeni. Promjena se može pojaviti usred, na primjer, interakcije korisnika s aplikacijom ili promjene stanja mobilnog uređaja. Trivijalni primjer bi bila početna aplikacija dana pri postavljanju svakog novog *Flutter* projekta, koja sadrži gumb i tekst koji prikazuje koliko puta je gumb pritisnut.

Stateful widget mijenja stanje, ali s obzirom na to da *widgeti* sami po sebi nisu promjenjivi, spremaju takva stanja unutar posebne *State* klase. *State* klasa sadrži *build* metodu, koja treba biti pozvana nakon svake promjene stanja kako bi se novo stanje reflektiralo na zaslonu. Pozivajući

metodu *setState*, Flutter dobiva informaciju o razlici trenutnog stanja *widgeta* i onog prikazanog na zaslonu te poziva *build* methodu *State* klase.

4.1.1.3. Propagacija stanja

Kako je *widget* klasa, moguće mu je predati podatke kao parametre konstruktora i koristiti ih u *build* metodi. Ovaj pristup je primjenjiv dok se aplikacija sastoji od svega par *widgeta*. Kako se dodaju nove funkcionalnosti i implementiraju zasloni, postupak prosljeđivanja podataka kroz konstruktore postaje vrlo težak. Stoga, kao rješenje *Flutter* pruža treći tip widgeta, *InheritedWidget,* čiji primjer korištenja je vidljiv na slici 4.3. *InheritedWidget* omogućuje jednostavan pristup podacima svim *widgetima* koji se nalaze na nižoj razini u hijerarhiji stabla.



Sl.4.3. Primjer *InheritedWidge*-a u stablu *widgeta*

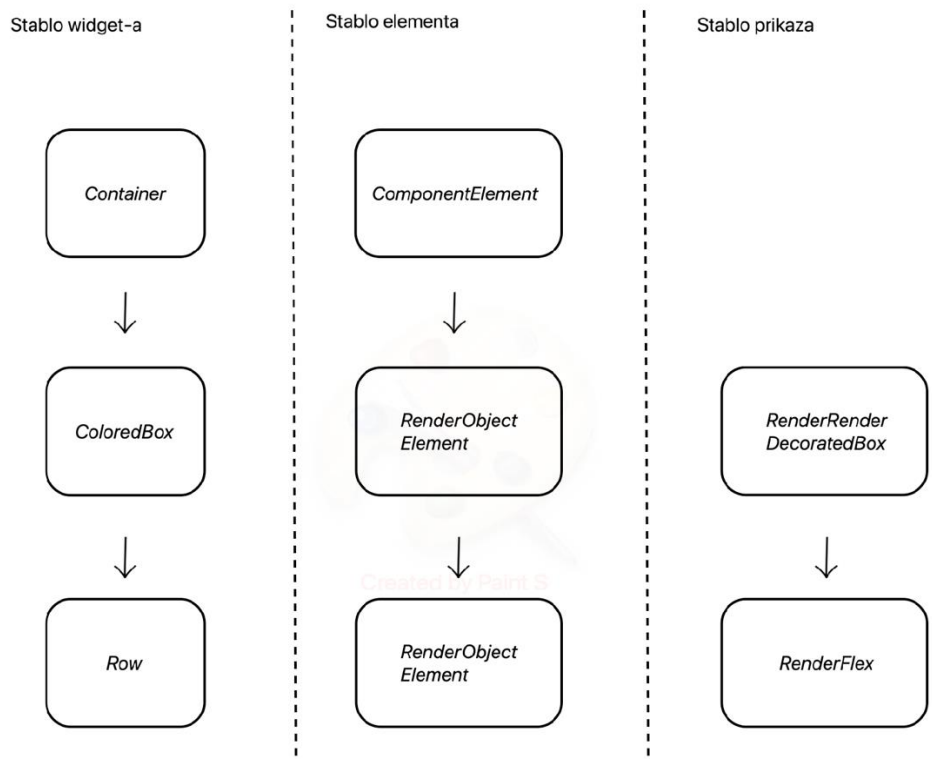
Ako Ispit treba podatke od StanjaStudenta, moguće ih je dobiti preko *context* parametra, kojeg svaki *widget* pruža kao argument svoje *build* metode. Prilikom toga, pruža se stanje od najbližeg *widgeta* koji se nalazi iznad unutar hijerarhije stabla tipa StanjeStudenta. Ova metodologija propagiranja stanja unutar aplikacije se koristi od samog *Flutter* okruženja te je osnova mnogih biblioteka napravljenih od strane programera u svrhu upravljanja stanjem unutar aplikacije.

4.1.1.4. Prikazivanje korisničkog sučelja

Nativne aplikacije, zajedno s bibliotekama pružaju komponente koje predstavljaju što se prikazuje na zaslonu. Ti elementi se koriste od strane programa *Skia* koji upravlja grafičkom karticom (GPU) te ovisno o primljenim komponentama govori grafičkoj kartici što treba prikazati na zaslonu. Obično višeplatformske tehnologije iskorištavaju ovo postojeću implementaciju nativnih tehnologija te jednostavno pružaju fasadu preko njih koja komunicira s obje platforme. *Flutter* je drugačiji po tome što totalno zamjenjuje tu postojeću funkcionalnost nativnih tehnologija sa svojom. Time se postiže bolja i brža komunikacija sa *Skia* programom te ima bolju kontrolu o tome što se prikazuje na korisničkom sučelju. Prikaz komuniciranja *Fluttera* sa grafičkom karticom vidljiv je na slivi 4.4.

Sl. 4.4. Komuniciranje *Flutter* aplikacije sa GPU

Flutter kao rješenje za određivanje što se prikazuje na korisničkom sučelju, koristi kompozicijsku strukturu stabla, i to tri različite vrsta stabla [19]. Tri stabla *Fluttera* prikazana su na slici 4.5.



Sl. 4.5. Stabla *Fluttera*

Stablo *widgeta* predstavlja upravu onu kompoziciju *widgeta* koju programer napiše unutar koda. Ono predstavlja deklarativan pristup izricanja kako treba izgledati korisničko sučelje u ovisnosti o trenutnom stanju aplikacije.

Postupak prikazivanja korisničkog sučelja, započinje *build* fazom unutar koje se izvršavaju *build* metode pojedinih *widgeta*. Iako je widget sam po sebi nepromjenjiv (immutable), tijekom izvođenja *build* metode *Flutter* može ubaciti nove *widgete* u stablo. Tijekom izvođenja *build* faze, *Flutter* preslikava stablo *widgeta* u stablo elemenata u odnosu jedan na prema jedan. Stablo elemenata predstavlja stvarni opis komponente korisničkog sučelja za pojedini *widget,* zajedno s njegovim stanjem ako je tipa *StatefulWidget*. Također, stablo elemenata povezuje stablo *widgeta* i stablo prikaza te upravlja stablima.

Svaki element zna koji mu je pripadajući *widget* iz stabla *widgeta* prema njegovom tipu i ključu, ako postoji. Ključ je jedinstveni identifikator koji se može predati svakom *widgetu*, kako bi element unutar stabla elemenata mogao bolje odrediti s kojim je *widgetom* povezan.

Svaki element unutar stabla elementata je jedan od dva osnovna tipa elemenata:

- Element komponente (*ComponentElement*)
- Element prikaza (*RenderObjectElement*)

Element komponente služi kao spremnik drugim elementima, dok element prikaza služe kao poveznica između pojedinog *widgeta* unutar stabla *widgeta* i objekta prikaza unutar stabla prikaza. Stablo prikaza se sastoji od različitih objekata prikaza kao na primjer *RenderImage* ili *RenderParagraph*, koji nasljeđuju od osnovnog tipa *RenderObject*. Svaki objekt sadrži informacije o tome kako pozicionirati i nacrtati pojedini *widget* na korisničkom sučelju.

RenderObject zna samo tko su mu djeca unutar stabla te koja su njihova ograničenja s obzirom na veličinu. Koristeći taj princip, *Flutter* prolazi kroz stablo koristeći *depth-first* iteraciju stabla. Prilikom posjete, svaki objekt preda djeci njihova dimenzijska ograničenja. Dijete pojedinog objekta, s obzirom na predana ograničenja, odgovori roditelju koje će mu biti dimenzije. Koristeći ovu tehniku, moguće je proći kroz stablo u linearnom vremenu. Nakon završetka prolaska kroz stablo, određene su dimenzije svakog objekta te su spremni za prikaz na korisničkom sučelju.

4.1.1.5. Prikazivanje promjene stanja

Iako *widget* nije promjenjiv, element unutar stabla elemenata je. *Flutter* koristi ovo u svoju prednost prilikom prikaza novog stanja na korisničkom sučelju. Kada se stablo *widgeta* promjeni, kako bi se indiciralo novo stanje aplikacije, *Flutter* ne gradi posve novo stablo elemenata. Ako je moguće, *Flutter* iskorištava već postojeće elemente unutar stabla. Svaki element zna s kojim je *widgetom* povezan s obzirom na njegov tip i ključ. Ako se *widget* zamjeni s *widgetom* istog tipa, i istog ključa, element se ponovo iskorištava i povezuje s novim *widgetom*. Jedina stavka koja se mijenja, su nove informacije iz novonastalog *widgeta*. To mogu biti informacije o veličini teksta, boji itd. Time, element prosljeđuje potrebne informacije objektu prikaza unutar stabla prikaza, koji se također ponovo koristi. Nakon prosljeđivanja novih informacija objektu prikaza, postupak pozicioniranja *widgeta* na zaslonu se ponavlja te se oni oslikavaju na korisničkom sučelju koristeći nove upute o prikazu na zaslon.

4.1.2. SwiftUI

SwiftUI je novi Apple-ov alat za razvoj korisničkih sučelja za iOS, macOS, tvOS i watchOS. Omogućen je za korištenje prvi puta od 2019 godine, od kada ga Apple unaprjeđuje iz godine u godinu [20]. Kao programski jezik koristi Swift. Swift je programski jezik namijenjen za opću namjenu pri uzimanju modernog pristupa kod sigurnosti, performansi i metodologijama dobrog dizajna [21]. Swift je namijenjen kao zamjena *Objective-C* programskom jeziku te se koristi za razvoj svih aplikacija na Apple platformama.

4.1.2.1. Razlika s *UIKit* alatom

Iako se *SwiftUI* razlikuje od *UIKit* alata, u pozadini koristi elemente iz *UIKit* alata kako bi prikazao elemente korisničkog sučelja. Za razliku od UIKit alata, koji se koristi za izradu aplikacija još od 2008. godine, *SwiftUI* koristi deklarativan pristup pri izradi aplikacija te podržava više platformi od strane Apple firme [22].

Unutar *UIKit* alata, programer stvara pojedine prikaze te ih povezuje kako bi stvorio hijerarhiju prikaza koje skupa čine korisničko sučelje. Prilikom promjena na zaslonu, potrebno je ponovno kalkuliranje veličina i ograničenja pojedinih prikaza te moguće dodavanje i brisanje prikaza iz hijerarhije. Potrebno je aktivno praćenje i detektiranje promjena te aktivno ažuriranje promijenjenih prikaza. Kontrola prikaza, njihovo stvaranje i brisanje se izvodi na eksplicitni način te se razina kompleksnosti može dosta povećati u kratkom vremenu.

SwiftUI pruža programsko sučelje pomoću kojeg programer deklarira kako korisničko sučelje treba izgledati. *SwiftUI* koristi te informacije kako bi prikazao ispravno korisničko sučelje. Time je korisničko sučelje prikazano kao funkcija trenutnog stanja aplikacije. Promjenom stanja aplikacije, promjena se reflektira na zaslonu. Ovaj deklarativni pristup pri stvaranju prikaza na zaslonu ne zahtijeva ručno stvaranje i brisanje prikaza na zaslon od strane programera, što smanjuje kompleksnost i mogućnost grešaka u kodu.

SwiftUI pruža lakše rukovanje s animacijama za razliku od *UIKit alata* te dosta animacija i prijelaza dolaze direktno od samog alata. *SwiftUI* zahtijeva korištenje novije inačice operacijskog sustava na uređaju za kojeg se aplikacija radi. *SwiftUI* je dostupan od iOS 13 inačice, ali nije spreman za pravu komercijalnu upotrebu u toj početnoj verziji alata. Stoga, mnogi programeri koriste *UIKit* kako bi pokrili sve ciljane verzije operacijskih sustava korisnika. No, s vremenom minimalna ciljana verzija se diže te *SwiftUI* postaje bolja opcija pri odabiru alata za izradu korisničkih sučelja. Razlike između *UIKit* i *SwiftUI* su prikazane u tablici 4.1.

Tablica 4.1. Usporedba *SwiftUI* i *UIKit* alata

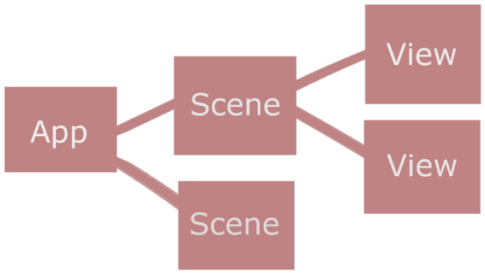
	UIKit	*SwiftUI*
Pristup rada	Imperativan	Deklarativan
Minimalna inačica iOS	iOS 9	iOS 13
Rad s alatom	14 godina	3 godine

Pokriva sve scenarije	da	ne
Brz proces izrade aplikacije	ne	da
Interface Builder podržan	da	ne
Podržava više platformi	ne	da
Rad s animacijama	težak	lagan
Podržava widgete	ne	da
Podržava Live Preview	ne	da

4.1.2.2. Struktura SwiftUI aplikacije

Struktura *SwiftUI* aplikacije se iskazuje na deklarativan način, kao što se iskazuju i prikazi na zaslonu. Početna točka svake aplikacije je kreirati strukturu koja nasljeđuje *App* protokol te ju je potrebno naznačiti sa *@main* ključnom riječi koja pruža osnovnu implementaciju *main* funkcije. Ovaj protokol predstavlja strukturu i ponašanje aplikacije [23].

App protokol zahtijeva implementiranje *body* atributa koji definira sadržaj aplikacije. Tijelo aplikacije je sastavljeno od scena te je svaka scena sastavljena od kompozicije različitih prikaza, kao što je prikazano na slici 4.6.



Sl. 4.6. Struktura *SwiftUI* aplikacije

Kako je svaki element struktura, a ne klasa, poboljšavaju se performanse pri izvođenju aplikacije. Strukture unutar Swift programskom jezika su *value type* tipovi podataka, što znači da su tretirane kao bilo koja druga vrijednost te ih je moguće predavati s jednog mjesta na drugi bez razmišljanja o stvaranju preljeva memorije ili održavanju pokazivača na njih.

Svaki prikaz mora naslijediti *View* protokol, koji također nalaže implementaciju *body* atributa koji mora vratiti točno jedan tip prikaza [24]. Korisničko sučelje tvori se kompozicijom više različitih prikaza, kako bi prikazali trenutno stanje aplikacije.

Moguće je razdvojiti posebne dijelove korisničkog sučelje u manje prikaze. Time se postiže veća jasnoća koda, moguće je ponovno korištenje istog prikaza na više različitih mjesta unutar koda te se smanjuje razina kompleksnosti i time smanjenje broja grešaka. Kako bi se pojedini prikaz konfigurirao na točno određeni način, *SwiftUI* pruža razne modifikatore za modificiranje izgleda prikaza. Oni predstavljaju metoda nad prikazima koje kao povratnu vrijednost vraćaju novi prikaz te se time mogu vezati pri pozivanju jedan iza drugoga.

4.1.2.3. Prikazivanje korisničkog sučelja

SwiftUI, kao *UIKi*, je apstrakcija iznad petlje događaja koja je odgovorna za slanja poruka kodu. Time, kod odgovoran za korisničko sučelje može odrediti koji se dijelovi zaslona trebaju ažurirati. Unutar *SwiftUI* alata petlja događaja je skrivena te nije potrebno direktno komunicirati s njom niti znati da ona postoji [25]. Na svim Apple platformama, nalazi se instanca petlje događaja *CFRunLoop* koja dolazi iz *Core Foundation* biblioteke te je prisutna na svim alatima za izradu korisničkog sučelja još od inačice Mac OS X 10.0.

Petlja događaja dio je infrastrukture povezane s nitima. Njen zadataka unutar te infrastrukture je rezervacija poslova koji se trebaju napraviti i organizacija nadolazećih događaja. Ova petlja daje nitima posao kada posla ima te stavlja niti u stanje spavanja kada nemaju posla za obradu [26]. Svi događaji koji dolaze unutar petlje događaja su obrađeni različitim redom, ovisno o tipu događaja. Postoje četiri različita tipa nadolazećih događaja:

- Tip 0. Događaji koji direktno komuniciraju sa *CFRunLoop* instancom, kao na primjer dodiri na zaslonu.
- Tip 1. Događaj sinkronizacije, ažuriranja zaslona i ponovnog prikazivanja na zaslonu te neki asinkroni događaji za dohvaćanje podataka s interneta.
- Mjerači vremena.
- Glavni red za obrađivanje.

Osim navedenih ulaznih događaja, moguće je dodati slušatelje na instancu *CFRunLoop* koji budu obaviješteni kada petlja dosegne određeni dio [27].

Animacije se vrte na glavnoj niti, ali ako glavna nit zablokira, moguće je da će animacije i dalje raditi. Razlog tome je što animacije, iako se vrte na glavnoj niti, ne vrte se na istom procesu. *SwiftUI* aplikacija pomoću biblioteke *Core Animation* prikazuje svoje animacije na zaslon, ali ih prikazuje na više različitih slojeva koji se nazivaju *CALayer*. Pomoću kompozitora aplikacija spaja više različitih slojeva te ih prikazuje na zaslon, dok *Core Animation* komunicira s poslužiteljem za crtanje prikaza na zaslon kako bi mu dao informaciju o tome što treba nacrtati i animirati.

4.1.2.4. Obnavljanje stanja na zaslonu

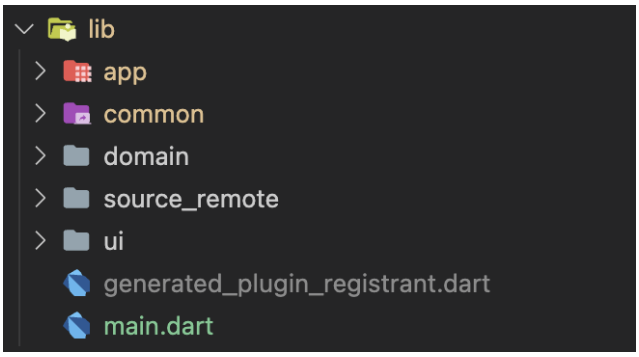
Pri promjeni stanja unutar aplikacije, potrebno je reflektirati tu promjenu na zaslonu promjenom pojedini dijelovi prikaza kao što su boja ili tekst. Nakon napravljenih promjena na pojedinim slojevima, potrebno je komunicirati tu promjenu poslužitelju za crtanje te prikazati ih. Aplikacija ne može znati kada su sve promjene gotove i spremne za crtanje na zaslon. Za signaliziranje završetka promjena koristi se transakcija *CATransaction*. Transakcija se mora pokrenuti kada se promjene dogode i završiti kada su promjene gotove. Pomoću ovog mehanizma aplikacija zna da su sve promjene gotove za trenutni sloj te se pomoću poslužitelja za crtanje prikazuju na zaslon [28].

Transakcije se mogu manualno stvoriti i završiti te su osnova unutar svake Apple, pa tako i *SwiftUI* aplikacije. Tijekom izrade aplikacija, programer ne komunicira direktno s transakcijama, iako to može učiniti uz pomoć *Core Animation* biblioteke, već se transakcija automatski pokrene dolaskom promjena i završi na kraju jednog obilaska petlje događaja.

4.2. Implementacija u Flutteru

4.2.1. Arhitektura Flutter aplikacije

Arhitektura unutar višeplatformske aplikacije u *Flutteru* je podijeljena na dijelove prikazane na slici 4.7.



Sl. 4.7. Pregled arhitekture *Flutter* aplikacije na najvećoj razini apstrakcije

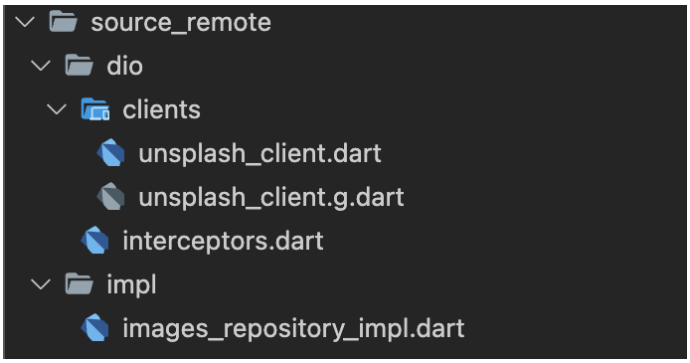
Unutar app foldera se nalaze stvari specifične za aplikaciju, kao što su rute za navigaciju kroz aplikaciju, sistem za ubacivanje klasnih ovisnosti (*dependency injection*), ulazni *widget* unutar aplikacije i slično. Sama *main* funkcija, koja predstavlja ulaznu točku unutar aplikacije, se nalazi unutar main.dart datoteke. Ona se nalazi na vrhu stabla datoteka i direktorija radi bolje preglednosti.

Ostatak aplikacije je podijeljen na tri glavna sloja:

- Podatkovni sloj
- Domenski sloj
- UI sloj

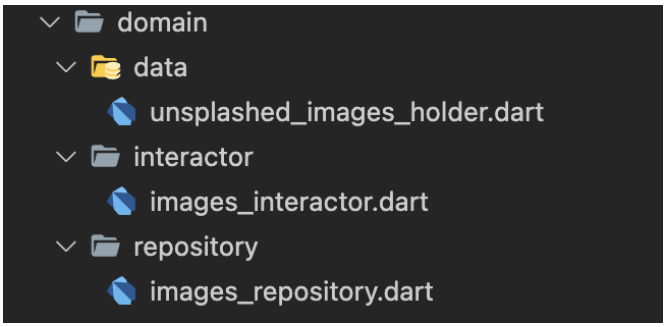
Podatkovni sloj je vanjski sloj aplikacije. On sadržava implementacije usluga/repozitorija koji komuniciraju s nekim vanjskim izvorima podataka (poslužiteljima), lokalnim izvorima (lokalna baza podataka), usluge za upravljanje sklopovljem te bilo kakve usluge ili menadžere koji nemaju povezanosti s poslovnom logikom same aplikacije te nisu vezani za aplikaciju. U tom smislu, mogli bi se izdvojiti posebno unutar posebne biblioteke te koristiti i unutar drugih aplikacija.

Ovom sloju pripada *source_remote* direktorij. U implementiranoj aplikaciji nema komunikacije s lokalnom bazom podataka ili sklopovljem uređaja. Prikaz arhitekture podatkovnog sloja u *Flutter* aplikaciji je prikazan na slici 4.8.



Sl. 4.8. Arhitektura podatkovnog sloja *Flutter* aplikacije

Domenski sloj je srednji sloj aplikacije koji sadrži apstrakciju usluga/repozitorija i menadžera koje aplikacije treba koristiti za potrebe izvođenja poslovne logike unutar aplikacije. Ove apstrakcije se odnose na implementacije iz podatkovnog sloja. Time se usluge iz vanjskog sloja aplikacije mogu lako zamijeniti drugom uslugom, bez mijenjanja poslovne logike. Arhitektura domenskog sloja *Flutter* aplikacije prikazana je na slici 4.9.



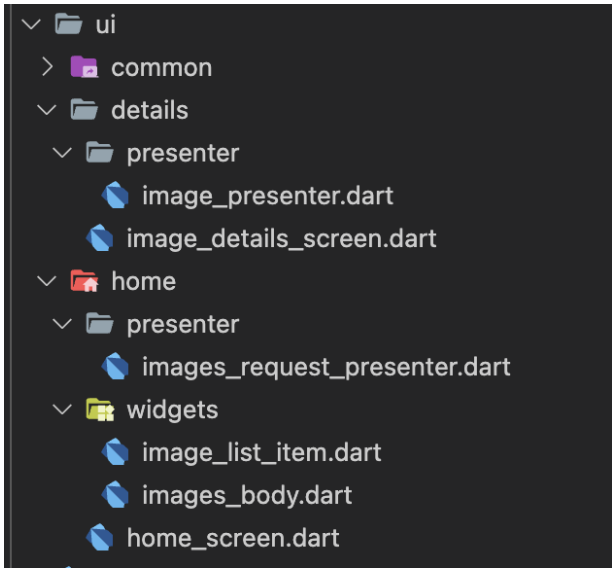
Sl. 4.9. Domenski sloj *Flutter* aplikacije

Domenski sloj također sadrži apstrakciju i konkretnu implementaciju upravljača. Svrha upravljača je koordiniranje obavljanja nekog zadatka kojeg zahtijeva poslovna logika aplikacije. Upravljači koriste repozitorije potrebne za obavljanje određenog zadatka. Time oni ne obavljaju nikakav konkretan rad, već pozivaju određene metode nad repozitorijima u određenom redoslijedu kako bi se postigao određeni rezultat ili cilj.

Također, upravljač može sadržavati privremene spremnike podataka, što su samo klase koje sadrže kao atribut određeni podatak te metode za spremanje i dohvaćanje tog podatka. Ovi privremeni spremnici olakšavajupristup podacima koji su raspoloživi svim repozitorijima.

UI sloj sadržava sve stranice koje aplikacija treba prikazati, pri čemu je svaka stranica idealno razbijena na više manjih *widgeta*. UI sloj sadržava sve stvari vezene za korisničko sučelje aplikacije, pa tako se u ovom sloju nalaze specifične stvari za temu aplikacije kao što su boje,

tipografija, podaci o vremenskim intervalima korišteni u animacijama ili mjerenjima vremena, konfiguracija teme aplikacije na osnovu danih podatka i slično. Arhitektura prikaza UI sloja *Flutter* aplikacije je prikazana na slici 4.10.

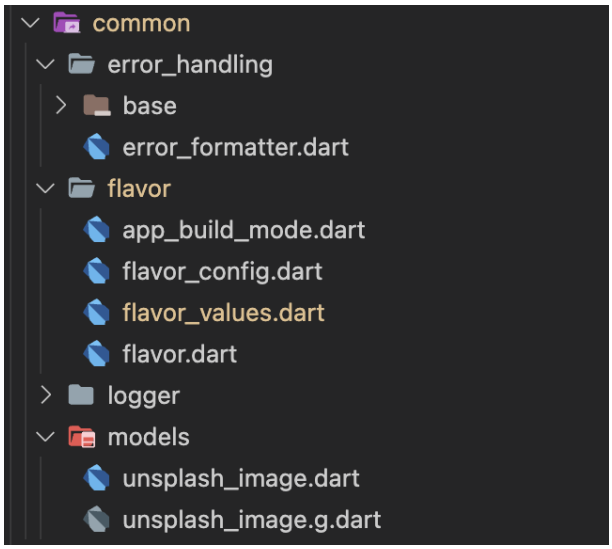


Sl. 4.10. UI sloj aplikacije

Osim izgleda aplikacije, ovaj sloj sadrži *state management* aplikacije. *State management* se odnosi na upravljanje stanjima aplikacije te predstavlja dio koji povezuje domenski sloj s UI slojem. Trenutni izgled aplikacije funkcija je trenutnog stanja aplikacije, time se promjenom stanja aplikacije mijenja korisničko.

Klase odgovorne za upravljanje stanjem aplikacije se nazivaju prezenteri. Svrha prezentera je da upravlja određenim dijelom korisničkog sučelja aplikacije tako da upravlja stanjem aplikacije koje se reflektira na taj dio korisničkog sučelja. Prezenter također može reagirati na razne događaje s korisničkog sučelja, kao što su pritisak na gumb, listanje kroz listu na zaslonu ili bilo kakvu akciju od strane korisnika. Prezenteri su jedina komponenta koja komunicira s komponentama korisničkog sučelja. Kao izlaz daju nekakvo stanje koje opisuje to sučelje, a kao ulaz reagiraju na razne događaje.

Direktorij *common*, na samom vrhu stabla, sadrži stvari koje se koriste u više slojeva aplikacije. Unutar *common* direktorija se nalaze stvari kao što su greške unutar aplikacije, različita okruženja aplikacije, modeli podataka koji se koriste na više slojeva aplikacije, pomoćne funkcije i slično. Direktorij *common* je vidljiv na slici 4.11.

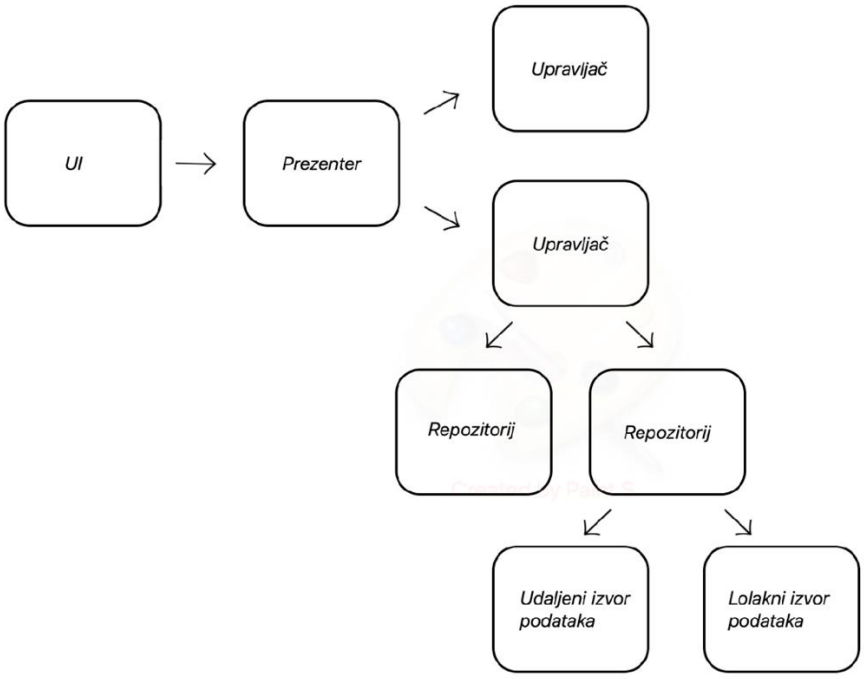


Sl. 4.11. Common direktoriji

Također, svaki sloj može sadržavati svoj *common* direktoriji, unutar kojem se nalaze stvari specifične za taj sloj. Na primjer unutar UI sloja to bi se odnosilo na boje, konfiguraciju teme, generičke klase za prezentere i slično. Modeli podataka unutar aplikacije također se mogu nalazi unutar posebnih slojeva.

4.2.2. Upravljanje stanjima unutar Flutter aplikacije

Komunikacija između slojeva unutar *Flutter* aplikacije prikazana je na slici 4.12.



Sl. 4.12. Upravljanje stanjima unutar *Flutter* aplikacije

Prezenter reagira na neki događaj, na primjer zahtjev za dohvaćanje dodatnih slika u listi slika. Prezenter potom poziva određeni upravljač koji je zadužen za koordiniranje obavljanja tog zadatka. Upravljač poziva repozitorije potrebne za izvršavanje tog zadatka. Zatim, repozitorij poziva uslugu za komuniciranje s poslužiteljem slika. Čeka se na odgovor od poslužitelja. Nakon što odgovor stigne, on se pretvara u željeni model podatka kako bi se mogao koristiti unutar aplikacije. Repozitorij vraća rezultat operacije upravljaču koji je pozvao metodu nad repozitorijem te upravljač prosljeđuje rezultat prezenteru. Prezenter obavještava korisničko sučelje o novom stanju te se korisničko sučelje ažurira s obzirom na novonastalo stanje.

Za implementaciju prezentera koristila se biblioteka *Riverpod*, koja predstavlja jedno od mogućih rješenja upravljanja stanjima unutar aplikacije.

4.2.3. Beskonačna lista slika u Flutteru

Implementacija beskonačne liste slike se izvela pomoću *infinite_scroll_pagination* biblioteke, čije korištenje je prikazano na slici 4.13.

```
@override
Widget build(BuildContext context) {
  ref.listen<ImagesRequestPresenter>(imagesRequestPresenter,
      (_, presenter) => onPreseterStateChanged(presenter));

  return PagedListView<int, UnsplashImage>(
    pagingController: _pagingController,
    itemExtent: 400,
    builderDelegate: PagedChildBuilderDelegate<UnsplashImage>(
      itemBuilder: (ctx, image, _) => ImageListItem(image: image),
      firstPageErrorIndicatorBuilder: (_) => ListError(
          onTryAgain: () => _pagingController.retryLastFailedRequest()),
      newPageErrorIndicatorBuilder: (_) => ListError(
          onTryAgain: () => _pagingController.retryLastFailedRequest()),
      firstPageProgressIndicatorBuilder: (_) => const Center(
        child: CircularProgressIndicator.adaptive(
          strokeWidth: 1.4,
        ), // CircularProgressIndicator.adaptive
      ), // Center
      newPageProgressIndicatorBuilder: (_) => const Center(
        child: CircularProgressIndicator.adaptive(
          strokeWidth: 1.4,
        ), // CircularProgressIndicator.adaptive
      ), // Center
      noItemsFoundIndicatorBuilder: (_) => const Center(
        child: CircularProgressIndicator.adaptive(
          strokeWidth: 1.4,
        ), // CircularProgressIndicator.adaptive
      ), // Center
      noMoreItemsIndicatorBuilder: (_) => const Center(
        child: CircularProgressIndicator.adaptive(
          strokeWidth: 1.4,
        ), // CircularProgressIndicator.adaptive
      ), // Center
    ), // PagedChildBuilderDelegate
  ); // PagedListView
}
```

Sl. 4.13. Implementacija beskonačne liste slika

Biblioteka *infinite_scroll_pagination* omogućuje definiranje izgleda pojedinačnog element unutar liste, izgleda pri čekanju na dodatne elemente koji se trebaju nadodati na listu te izgled greške pri dohvaćanju novih elemenata liste. Za prikaz elementa liste koristi se *widget ImageListItem,* za učitavanja se prikazuje indikator učitavanja podataka, dok za stanje greške prikazuje se *ListeError widget* koji prikazuje korisniku grešku na zaslonu.

Osim vizualnih reprezentacija, biblioteka pruža način slušanja na obavijesti o tome kada je potrebno dohvatiti nove podatke za prikaz u listi, kao što je prikazano na slici 4.14.

```
class _ImagesBodyState extends ConsumerState<ImagesBody> {
  late final PagingController<int, UnsplashImage> _pagingController;

  @override
  void initState() {
    _pagingController = PagingController(firstPageKey: 0);
    _pagingController.addPageRequestListener((pageKey) {
      ref.read(imagesRequestPresenter.notifier).fetchImages(pageKey);
    });
    super.initState();
  }

  @override
  void dispose() {
    _pagingController.dispose();
    super.dispose();
  }
```

Sl. 4.14. Zahtjev za dohvaćanjem dodatnih slika

Za slušanje na događaje i kontroliranje stanja liste, koristi se klasa iz biblioteke *PagingController*. Potrebno je napraviti novu instancu ove klase, pridružiti je *widgetu* liste koju kontrolira te ju osloboditi kada se *widget* trajno makne iz stabla. Pri obavještavanju o potrebi za novim podacima, koristi se metoda *addPageRequestListener* koja pruža se podatak *pageKey* koji predstavlja broj zadnjeg elementa unutar liste. Unutar nje poziva se *fetchImages* metoda nad prezenterom sa pruženim *pageKey* parametrom.

Prezenter zatim poziva upravljača da obavi radnju dohvaćanja potrebnih slika. Potrebno je poslati broj stranice s koje se želi dohvatiti slike. Prezenter nasljeđuje klasu *RequestProvider* koja služi kao fasada oko izvođenja asinkronih zahtjeva te vraća određeno stanje. To stanje može biti uspješno i vratiti neke podatke (u ovom slučaju listu modela *UnsplashImage*), u početnom stanju, neuspješno ili u procesu učitavanja.

Zahtjev koji se želi izvesti izvršava se pomoću metode *executeRequest,* koja kao parametar prima asinkronu funkciju koju želimo izvršiti. Implementacija prezentera vidljiva je na slici 4.15.

```dart
final imagesRequestPresenter =
    ChangeNotifierProvider.autoDispose<ImagesRequestPresenter>(
  ((ref) => ImagesRequestPresenter(getIt<ImagesInteractor>())),
);

class ImagesRequestPresenter extends RequestProvider<List<UnsplashImage>> {
  ImagesRequestPresenter(this._imagesInteractor);

  final ImagesInteractor _imagesInteractor;
  int pageNumber = 0;

  void fetchImages(int pageKey) {
    if (pageKey == 0) {
      pageNumber = 0;
    }
    executeRequest(requestBuilder: () async {
      final result = await _imagesInteractor.fetchImages(pageNumber);
      pageNumber++;
      return result;
    });
  }
}
```

Sl. 4.15. Prezenter liste slika

Kako bi lista se nadopunila s novim podacima, potrebno je slušati na stanja koja prezenter daje kao izlaz. To se postiže tako da se sluša promjena na samom prezenteru pomoću metode *listen*. Pri promjeni stanja poziva se metoda onPresenterStateChanged, koja pomoću kontrolera za listu ubacuje nove podatke na kraj liste ili obavještava kontroler o neuspješnom dohvaćanju novih podataka. Slušanje promjena stanja prezentera prikazano je na slici 4.16.

```dart
void onPreseterStateChanged(ImagesRequestPresenter presenter) {
  final nextPageKey = _pagingController.nextPageKey ?? 0;
  presenter.state.maybeWhen(
    success: (result) {
      _pagingController.appendPage(result, nextPageKey + result.length);
    },
    orElse: () => _pagingController.error = ImagesFetchingException(),
  );
}
```

Sl. 4.16. Slušanje na nova stanja prezentera u *Flutteru*

4.2.4. Filtriranje i spremanje slike u Flutteru

Filter slike primjenjuje se pomoću metode *_transformPixels*, vidljivoj na slici 4.17.

```
Future<void> _transformPixels(String imageUrl) async {
  final image = NetworkImage(imageUrl);
  Bitmap bitmap = await Bitmap.fromProvider(image);
  late final Uint8List newImage;
  if (filteredImage != null) {
    newImage =
        bitmap.apply(BitmapAdjustColor(exposure: intensitiy)).buildHeaded();
  } else {
    newImage = bitmap.buildHeaded();
  }
  setState(() {
    filteredImage = newImage;
  });
}
```

Sl. 4.17. Filtriranje slike u *Flutteru*

Unit8List predstavlja pojedinačne bajtove slike pri čemu se jedan bajt klasificira kao osam bitova. Preko tih bajtova predstavlja se intenzitet pojedine boje piksela. Pojedini piksel slike se predstavlja kao četiri bajta, čijom promjenom mijenjamo izgled slike na zaslonu. Transformacija se obavlja pomoću *Bitmap* klase koja omogućuje operacije boje nad bajtovima koji predstavljaju pojedinačni piksel te dodaje zaglavlje u listu bajtova potrebno za prikaz slike na zaslon pomoću *Image widgeta*.

Spremanje slike u galeriju slika na uređaju korisnika se obavlja pomoću *_capturePng* metode, koja se pozivi kada korisnik pritisne gumb za spremanje trenutno filtrirane slike te je vidljiva na slici 4.18.

```
Future<void> _capturePng() async {
  try {
    if (filteredImage == null) {
      return;
    }
    await ImageGallerySaver.saveImage(filteredImage!,
        quality: 100, name: 'file_name${DateTime.now()}');
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text('Saved to gallery!')));
  } catch (e) {
    print(e);
  }
}
```

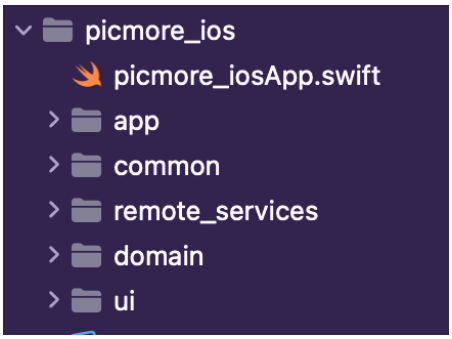
Sl. 4.18. Spremanje filtrirane slike

Spremanje slike an uređaj se odvija pomoću *image_gallery_saver* biblioteke. Biblioteka pruža *ImageGallerySaver* klasu koja sadrži *saveImage* metodu. Pomoću *saveImage* metode sprema se filtrirana slika na korisnikov uređaj.

4.3. Implementacija u SwiftUI

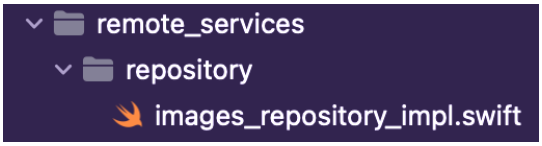
4.3.1. Arhitektura SwiftUI aplikacije

Arhitektura korištena u *SwiftUI* aplikacije je slična aplikaciji napravljenoj u *Flutteru*. *SwiftUI* aplikacija sastoji se od tri već spomenuta (podatkovni sloj, domenski sloj i UI sloj). Arhitektura *SwiftUI* aplikacije prikazana je na slici 4.19.



Sl. 4.19. Arhitektura *SwiftUI* aplikacije

App direktoriji sadržava stvari vezane za aplikaciju, kao greške unutar aplikacije, *dependency injection* i slično. Ulazna točna unutar aplikacije je App struktura, koja se nalazi unutar picmore_iosApp direktorija. Aplikacije se dijeli na prethodno spomenuta tri sloja (podatkovni, domenski i UI). Podatkovnom sloju, prikazanom na slici 4.20., pripada direktoriji remote_services, koji sadrži implementaciju repozitorija zaslužnog za upravljanje slikama.

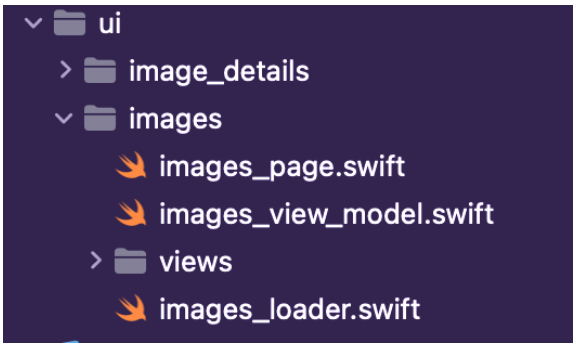


Sl. 4.20. Podatkovni sloj iOS aplikacije

Domenski sloju pripada *domain* direktorij, unutar kojeg se nalazi apstrakcija repozitorija, apstrakcija i implementacija upravljača. Arhitektura domenskog sloja *SwiftUI* aplikacije prikazan je na slici 4.21.

Sl. 4.21. Domenski sloj iOS aplikacije

UI sloju aplikacija pripada UI direktorij, prikazan na slici 4.22. Unutar njega nalaze se implementacija zaslona liste slike i detalja slike.

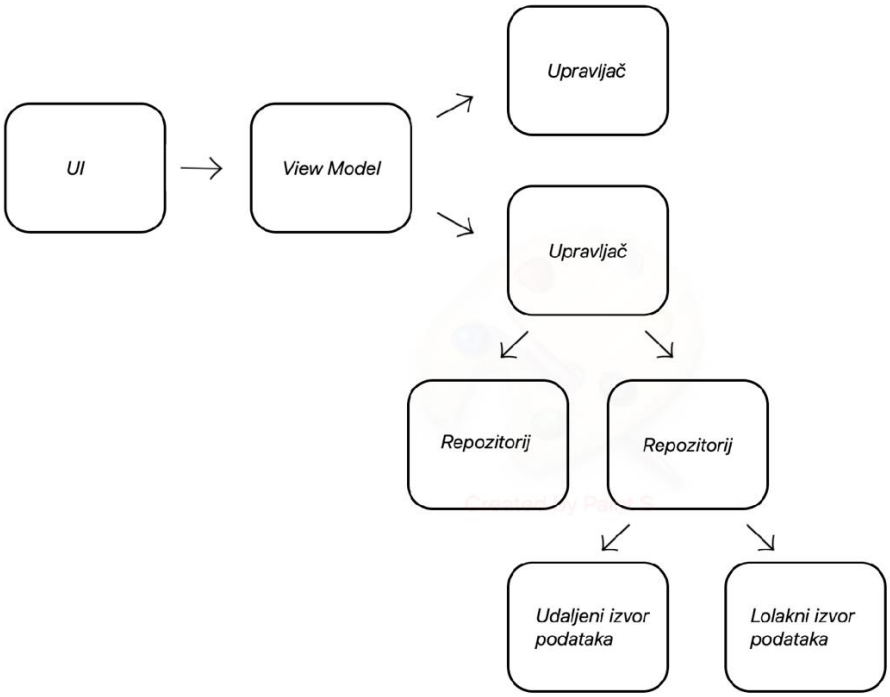


Sl. 4.22. UI sloj iOS aplikacije

Osim korisničkog sučelja, unutar UI sloja se nalaze klase koje upravljaju stanjima korisničkog sučelja, nazvanih *View Models*. Oni imaju istu ulogu kao što prezenter ima unutar *Flutter* aplikacije te je jedina razlika između njih je njihov naziv i sam način upravljanja stanja. *View Model* zaslužan je za reagiranje na događaje s korisničkog sučelja te izmjenu stanja na korisničkom sučelju.

4.3.2. Upravljanje stanjima unutar SwiftUI aplikacije

Kao i arhitektura, upravljanje stanjima je u teoriji identično kao i u *Flutter* aplikaciji te je prikazana na slici 4.23. Stvarna razlika je u konkretnoj implementaciji i mehanizmima koji se koriste u oba pristupa.



Sl. 4.23. Upravljanje stanjima unutar iOS aplikacije

Korisničko sučelje šalje poruku *View Model*-u o nekom događaju, kao na primjer klik na gumb ili dolazak na zaslon. U ovom slučaju, zahtijevaju se novi podaci o slikama. *View Model* potom poziva upravljač za taj slučaj. Upravljač koordinira repozitorije kako bi dobio određene podatke ili izvršio određeni cilj. Repozitorij zatim poziva neke udaljene ili lokalne izvore podataka kako bi dobio željene podatke, neke usluge koje odrađuju posao ili menadžere koji su upravljaju za nekim dijelom sklopovlja.

4.3.1. Beskonačna lista slika u SwiftUI

InfiniteList je prikaz koji je zaslužan za prikazivanje beskonačne liste slike, čija je implementacija vidljiva na slici 4.24.

```
struct InfiniteList<Data, Content, LoadingView>: View
where Data: RandomAccessCollection, Data.Element: Hashable, Content: View, LoadingView: View  {
    // MARK: - Properties
    @Binding var data: Data
    @Binding var isLoading: Bool
    @Binding var isError: Bool
    let loadingView: LoadingView
    let loadMore: () -> Void
    let content: (Data.Element) -> Content
    let myColor = Color(🟨)
    let blueColor = Color(🟦)

    // MARK: - Init
    init(data: Binding<Data>,
         isLoading: Binding<Bool>,
                isError: Binding<Bool>,
         loadingView: LoadingView,
         loadMore: @escaping () -> Void,
         @ViewBuilder content: @escaping (Data.Element) -> Content) {
        _data = data
        _isError = isError
        _isLoading = isLoading
        self.loadingView = loadingView
        self.loadMore = loadMore
        self.content = content
    }
```

Sl. 4.24. Atributi prikaza beskonačne liste

Atribut koji mijenja stanje prikaza koristi se atribut *data*, koji predstavlja podatke koji se prikazuju unutar liste. U ovom slučaju, to je lista modela *UnsplashImage*. Atribut *isLoading* predstavlja trenutno stanje učitavanja novih slika u listi. Atribut *isError* prikazuje grešku na korisničkom sučelju, čime se informira korisnika o neuspješnom dohvaćanju slika. Kao atribute kojima se postavlja izgled ovog prikaza, prikaz sadrži loadingView atribut, koji se prikazuje dok lista čeka na nove slike. Atribut *content* predstavlja funkciju koja gradi pojedini element unutar liste. Kako

bi ovaj prikaz zahtijevao dohvaćanje novih slika, ima kao atribut *loadMore,* koji predstavlja funkciju koju ovaj prikaz poziva kada zatraži nove elemente u listi.

Kao i svaki prikaz koji se koristi unutar *SwiftUI*, potrebno je da prikaz nasljeđuje protokol *View* te u *body* atributu definira izgled svoga prikaza. *InfiniteList* prikaz implementira *body* atribut tako da prvo prikaže sve elemente liste. Nakon liste slijedi prikaz pogreške, ako se je *isError* postavljen na vrijednost *true* te ako se više ne učitavaju vrijednosti novih slika. Dok se učitavaju nove slike, prikazuje se indikator učitavanja na dnu liste. Implementacija prikaza beskonačne liste u *SwiftUI* prikazana je na slici 4.25.

```
var body: some View {
        List {
            listItems
        }
        .onAppear(perform: loadMore)
        .listStyle(GroupedListStyle())
}

var listItems: some View {
    Group {
        ForEach(data, id: \.self) { item in
            content(item)
                .listRowBackground(myColor)
                .onAppear {
                    if item == data.last {
                        loadMore()
                    }
                }
        }
        if isError && !isLoading {
            Text("Error fetchign, try again?")
                .onTapGesture {
                    loadMore()
                }
        }
        if isLoading {
            loadingView
                .frame(idealWidth: .infinity, maxWidth: .infinity, alignment: .center)
        }
    }
}
```

Sl. 4.25. Implementiranje prikaza beskonačne liste iOS aplikacije

Prikaz *ImagesPage* je glavni prikaz na stranici gdje se prikazuje lista slika, u sebi koristi *InfiniteList* prikaz te ga konfigurira s *View Model*-om koji je odgovoran za upravljanje ovim dijelom korisničkog sučelja. Implementacija *ImagesPage* prikaza prikazana je na slici 4.26.

```
struct ImagesPage: View {
    @ObservedObject var viewModel: ListViewModel = ListViewModel()

    init() {•••}

    var body: some View {
        InfiniteList(data: $viewModel.images,
                     isLoading: $viewModel.isLoading,
                     isError: $viewModel.isError,
                     loadingView: ProgressView(),
                     loadMore: viewModel.loadMore
        ) { image in
            NavigationLink {
                ImageDetails(image: image)
            } label: {
                ListItem(image: image)
            }

        }
    }
}
```

Sl. 4.26. Implementiranje prikaza beskonačne liste

Prilikom zahtjeva novih slika za prikaz u listi, *InfiniteList* prikaz će pozvati *loadMore* metodu nad *View Model*-om. Time *View Model* mijenja stanje u stanje učitavanja novih slike te poziva upravljač da obavi dohvaćanje novih slika s poslužitelja.

Nakon što upravljač dohvati nove slike, *View Model* ažurira stanje s dohvaćenim slikama s poslužitelja te se to reflektira na korisničkom sučelju. Kako bi korisničko slučelje slušalo na promjene stanja s *View Model*-a, unutar *View Model*-a dodaje se ključna riječ *@Published* ispred atributa čije stanje se želi promatrati. U ovom slučaju, *View Model* koristi ključnu riječ *@Published* ispred *images* atributa, koji služi za prikaz slika te ispred *isLoading* i *isError* atributa, koji služe za prikaz greške ili indikatora učitavanja. Implementacija *View Modela* prikazana je na slici 4.27.

```swift
class ListViewModel: ObservableObject {
    // MARK: - Properties
    @Published var images = [UnsplashImage]()
    @Published var isLoading = false
    @Published var isError = false
    private var page = 0
    private let imagesInteractor: ImagesInteractor = ImagesInteractorImpl()
}

// MARK: - Public methods
extension ListViewModel {
    func loadMore() {
        guard !isLoading else { return }
        isLoading = true
        Task {
            do {
                let newImages = try await imagesInteractor.fetchImages(for: page)
                showLoaded(with: newImages)
            } catch(let error) {
                showError()
                print("Error fetching images: \(error)")
            }
        }
    }
}

// MARK: - Private methods
private extension ListViewModel {
    func showError() {
        DispatchQueue.main.async { [weak self] in
            self?.isError = false
            self?.isLoading = false
        }
    }

    func showLoaded(with newImages: [UnsplashImage]) {
        DispatchQueue.main.async { [weak self] in
            self?.page += 1
            self?.images.append(contentsOf: newImages)
            self?.isError = true
            self?.isLoading = false
        }
    }
}
```

Sl. 4.27. Implementacija *View Model*-a beskonačne liste

4.3.1. Filtriranje i spremanje slike u SwiftUI

Pritiskom na gumb za primjenu filtera poziva se metoda *applyProcessing* koja primjenjuje filter izoštrenja i sprema rezultata unutar *processedImage* varijable. Varijabla *processedImage* se pomoću *Image* prikaza prikazuje na korisničkom sučelju. Implementacija prikaza filtriranja slike je prikazana na slici 4.28.

```
var body: some View {
        VStack(alignment: .center) {
            Image(uiImage: processedImage)
                .resizable()
                .aspectRatio(contentMode: .fill)
                .frame(width:  UIScreen.main.bounds.width, height: 600, alignment: .center)
                .clipped()
                .cornerRadius(0)

            Button("Apply filter.") {
                applyProcessing()
            }
            .padding(EdgeInsets(top: 10, leading: 20, bottom: 10, trailing: 20))
            Button("Save image to gallery.") {
                save()
            }
        }
        .onReceive(imageLoader.didChange) { data in
            print("Loaded")
            processedImage = UIImage(data: data) ?? UIImage()
            loadImage()
        }

}
```

Sl. 4.28. Implementacija prikaza filtriranja u SwiftUI

Metoda *applyProcessing* primjenjuje filter na slici mijenjanjem vrijednosti matrice sastavljene od četiri vektora, pri čemu svaki vektor predstavlja vrijednost bolje piksela. Matrica se potom množi sa slikom, pri čemu se pojedini pikseli slike mijenjaju i dobiva se željeni filter. Implementacija filtera je vidljiva na slici 4.29.

```
func applyProcessing() {
    DispatchQueue.global().async {
        currentFilter.setValue(CIVector(x: CGFloat(filterIntensity), y: 0, z: 0, w: 0), forKey: "inputRVector")
        currentFilter.setValue(CIVector(x: 0, y: CGFloat(filterIntensity), z: 0, w: 0), forKey: "inputGVector")
        currentFilter.setValue(CIVector(x: 0, y: 0, z: CGFloat(filterIntensity), w: 0), forKey: "inputBVector")
        guard let outputImage = currentFilter.outputImage else { return }
        if let cgimg = context.createCGImage(outputImage, from: outputImage.extent) {
            DispatchQueue.main.async {
                processedImage = UIImage(cgImage: cgimg)
            }
        }
    }
}
```

Sl. 4.29. Implementacija filtriranja slike u SwiftUI

Korisnik pritiskom na gumb s tekstom „Save image to gallery“ sa slike 4.19, sprema filtriranu sliku u galeriju slika na uređaju. Klasa *ImageSaver* zaslužna je za spremanja slike u galeriju te je vidljiva na slici 4.30.

```swift
class ImageSaver: NSObject {
    var successHandler: (() -> Void)?
    var errorHandler: ((Error) -> Void)?

    func writeToPhotoAlbum(image: UIImage) {
        UIImageWriteToSavedPhotosAlbum(image, self, #selector(saveComplete), nil)
    }

    @objc func saveComplete(_ image: UIImage, didFinishSavingWithError error: Error?, contextInfo:
        UnsafeRawPointer) {
        if let error = error {
            errorHandler?(error)
        } else {
            successHandler?()
        }
    }
}
```

Sl. 4.30. Implementacija filtriranja slike u SwiftUI

Koristeći metodu *UIImageWriteTosavedPhotosAlbum* iz *UIKit* biblioteke, filtrirana slika sprema se u galeriju. Prilikom završetka spremanja slike, izvršava se metoda *saveComplete* koja je poziva funkcije za obavijest o uspješnom ili neuspješnom spremanju slike.

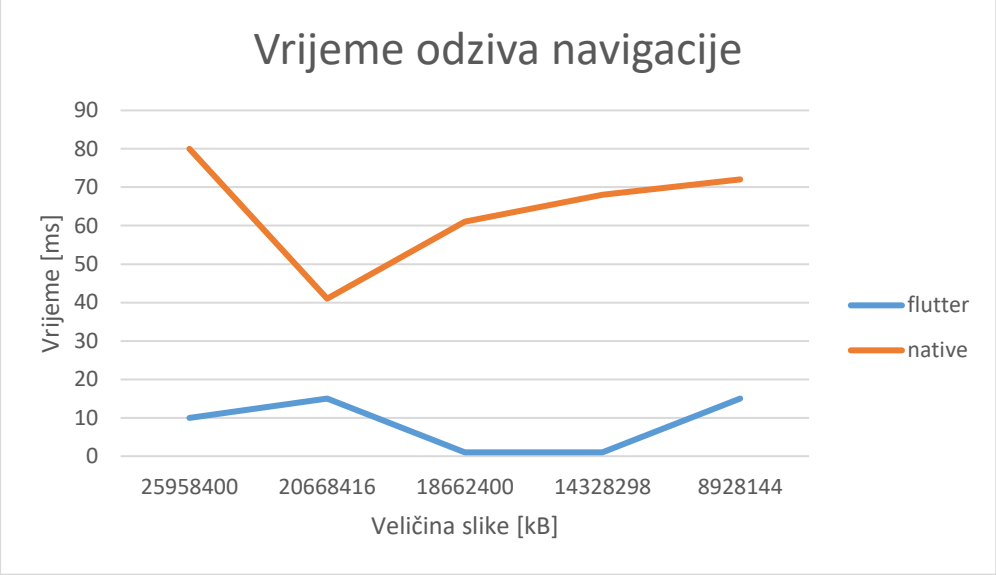
5. TESTIRANJE I ANALIZA PROGRAMSKOG RJEŠENJA

U ovom poglavlju prikazani su rezultati testiranja nativne i višeplatformske aplikacije na dva različita uređaja. Testni slučajevi i specifikacije su navedene u drugom poglavlju rada. Dan je opis rezultata te usporedba s rezultatima testiranja iz drugih radova za slučajeve gdje su ti podaci dostupni.

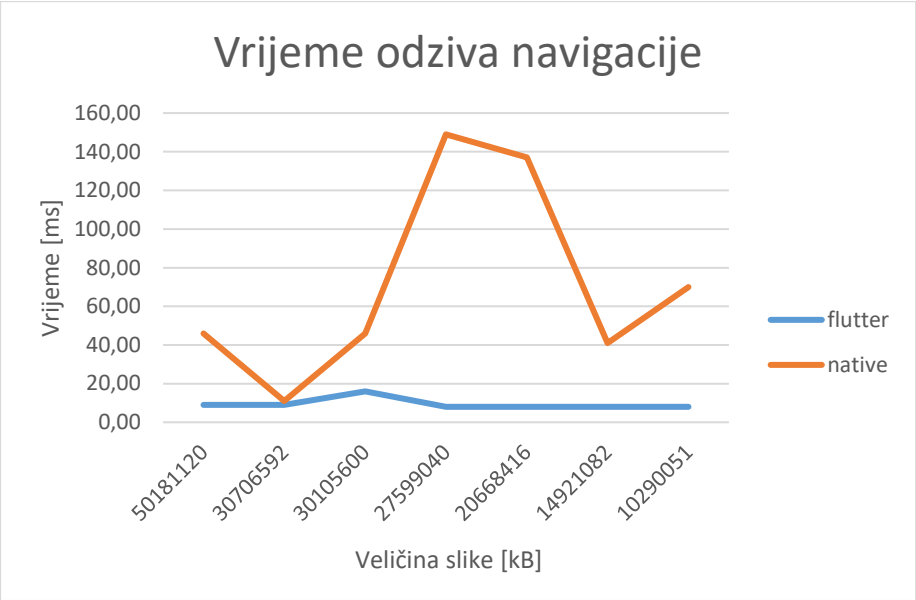
5.1. Testiranje

5.1.1. Vremena odziva

Rezultati testiranje vremena odziva u milisekundama navigacije sa zaslona prikaza liste slika na detalje slike u ovisnosti o veličini slika, u kilobajtima, su prikazani na slici 5.1. za uređaj iPhone 12 te slici 5.2. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* treba manje vremena da obavi prijelaz s jednog zaslona na drugi nego aplikacija napisana u *SwiftUI*. Obje aplikacije zadovoljavaju pravilo od vremena odziva manjeg od 100 ms.

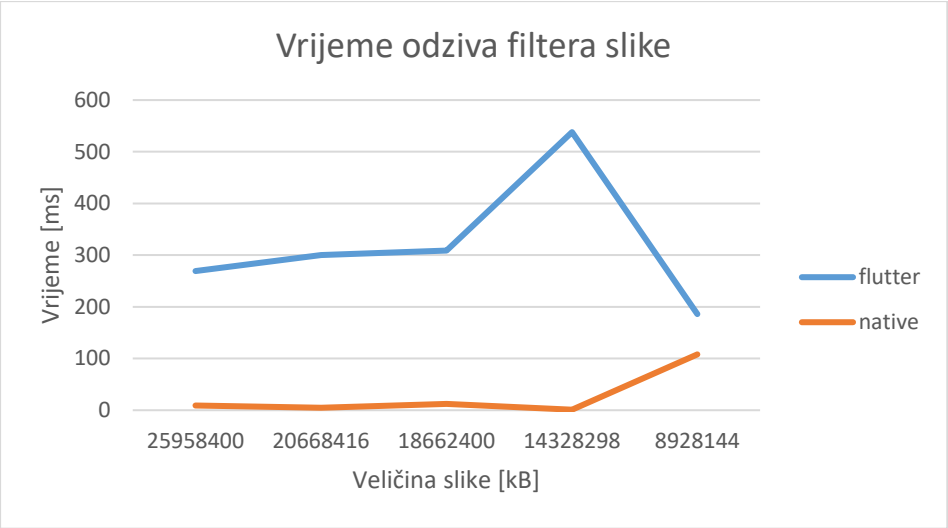


Slika 5.1. Vrijeme odziva navigacije iPhone 12

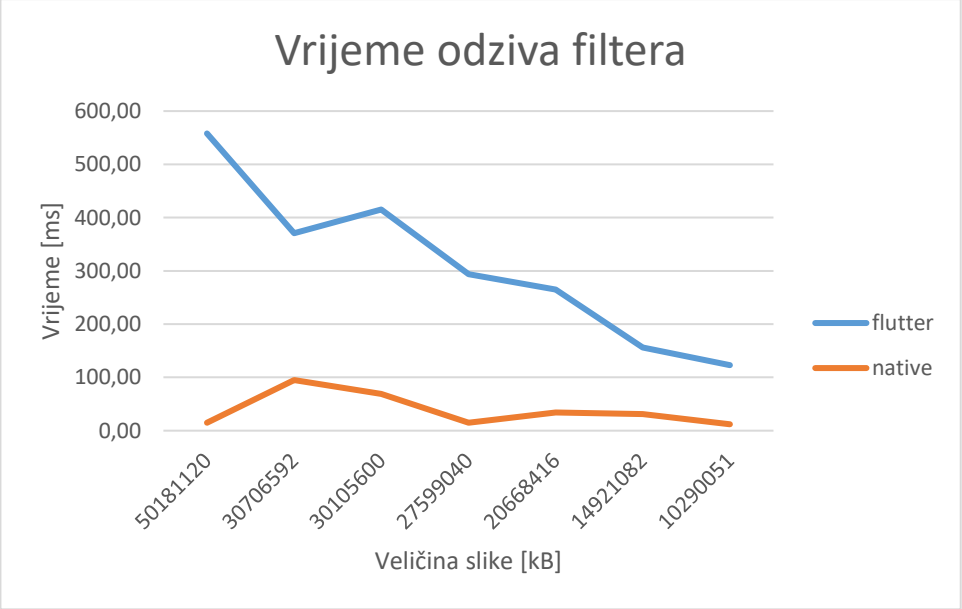


Slika 5.2. Vrijeme odziva navigacije iPhone 13 Pro Max

Rezultati testiranje vremena odziva u milisekundama funkcije filtriranja slike u ovisnosti o veličini slike, u kilobajtima, su prikazani na slici 5.3. za uređaj iPhone 12 te slici 5.4. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* treba znatno više vremena u odnosu na aplikaciju u *SwiftUI* i to u razmjeru do 6 puta sporijeg vremena odziva. Vidljivo je sa slike 5.4. da pri većim veličinama slika, vremena odziva postaju veća, pri čemu je porast vremena odziva aplikacije u *Flutteru* znatno veći nego aplikacije u *SwiftUI*. Gledajući sliku 5.3. i 5.4., najmanje vrijeme odziva *Flutter* aplikacije je očitano pri oko 200 milisekundi, dok je najviše vrijeme odziva aplikacije u *SwiftUI* očitano pri oko 100 milisekundi. S obzirom na da aplikacija u *SwiftUI* ima vremena odziva do sto milisekundi, događaj se izvršava trenutačno s pogleda korisnika, dok prema dobivenim rezultatima aplikacija u *Flutteru* će imati kratko zastajkivanje.



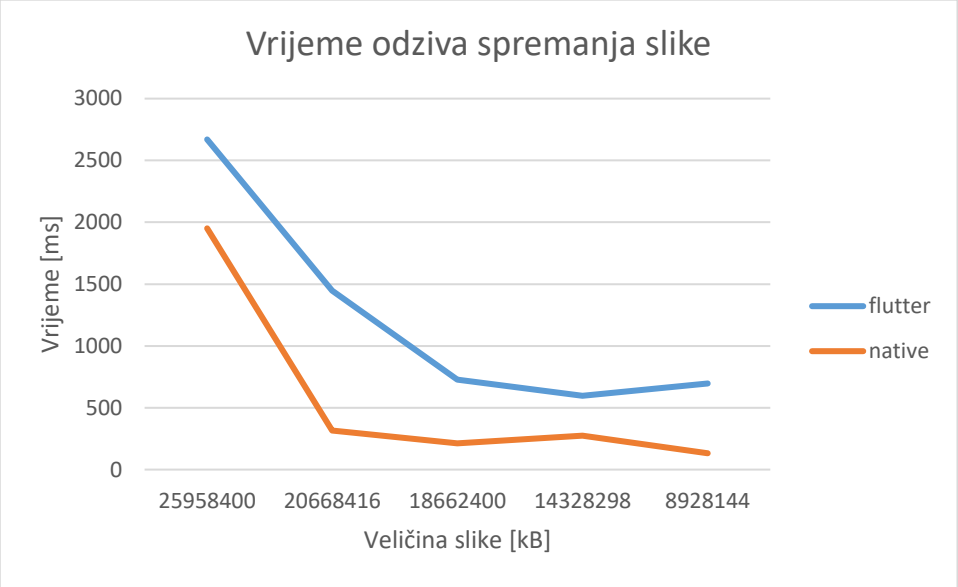
Slika 5.3. Vrijeme odziva funkcije filtriranja slike na uređaju iPhone 12



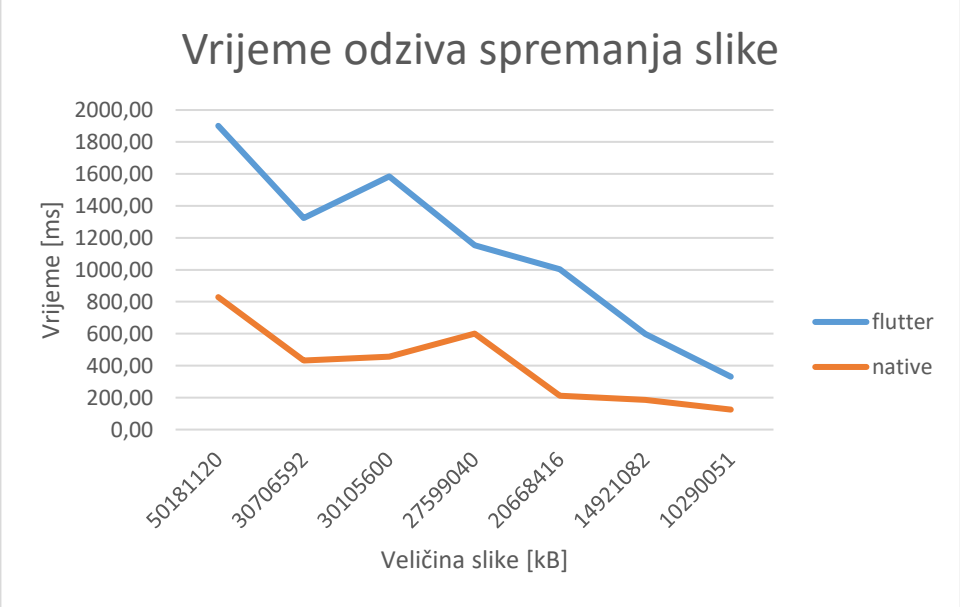
Slika 5.4. Vrijeme odziva funkcije filtriranja slike na uređaju iPhone 13 Pro Max

Rezultati testiranje vremena odziva u milisekundama funkcije spremanja slike na uređaj u ovisnosti o veličini slike, u kilobajtima, su prikazani na slici 5.5. za uređaj iPhone 12 te slici 5.6. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* treba u prosjeku dva puta više vremena da spremi sliku na uređaj od korisnika u odnosu na aplikaciju u *SwiftUI*. Vidljivo je sa slika 5.5. i 5.6 da pri većim veličinama slika, vremena odziva postaju veća, pri čemu nema znatne razlike u veličini porasta pojedinih aplikacija. Gledajući sliku 5.5. i 5.6., niti jedna aplikacija ne može spremiti sliku ispod 100 milisekundi te će se operacija spremanja primijetiti sa strane korisnika. Pri tome je bitno istaknuti da kod aplikacije u *SwiftUI* se to dešava u vremenu do jedne sekunde ili maksimalno par, što se smatra uredu ako se dešava rijetko unutar

aplikacije. U *Flutter* aplikaciji, operacija spremanja slike na uređaj u prosijeku traje više od jedne sekunde te ostaje u granicama od par sekundi, što je lošije nego aplikacija u *SwiftUI*, ali također prihvatljivo ako se ova radnja ne događa često.

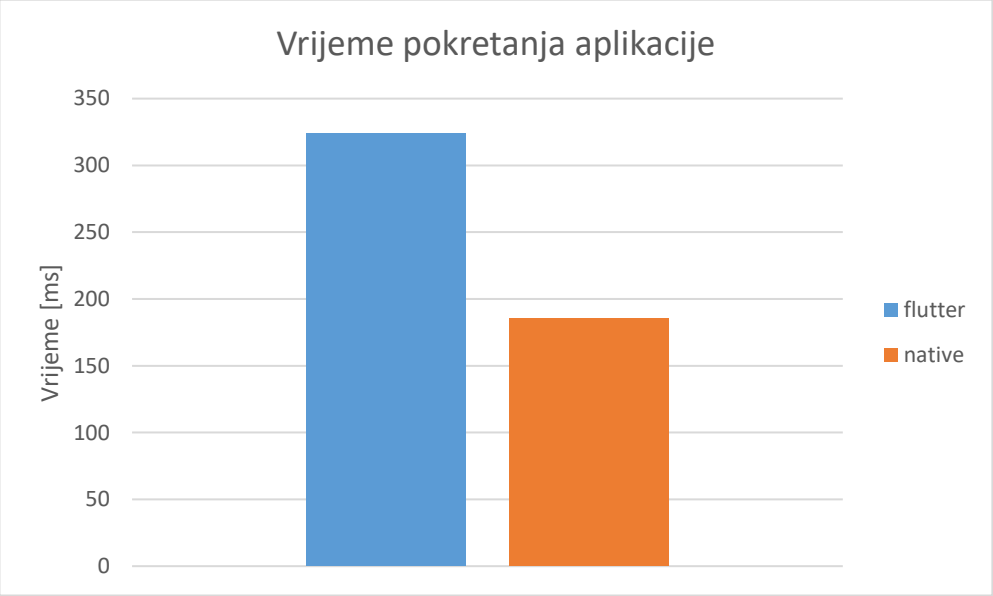


Slika 5.5. Vrijeme odziva funkcije spremanja slike na uređaju iPhone

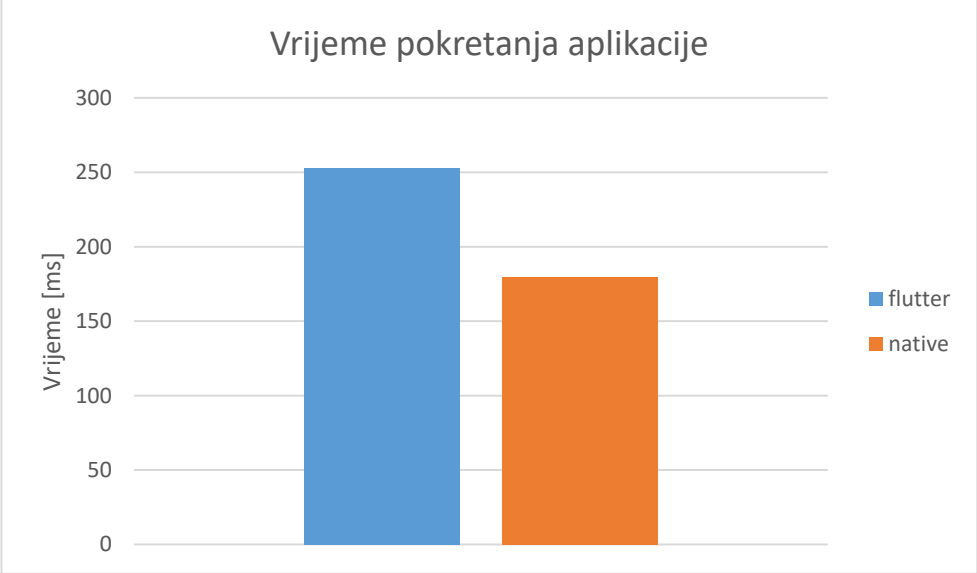


Slika 5.6. Vrijeme odziva funkcije spremanja slike na uređaju iPhone 13 Pro Max

Rezultati testiranje vremena pokretanja aplikacije u milisekundama su prikazani na slici 5.7. za uređaj iPhone 12 te slici 5.8. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* treba 30-40% više vremena da se pokrene u odnosu na aplikaciju u *SwiftUI*.



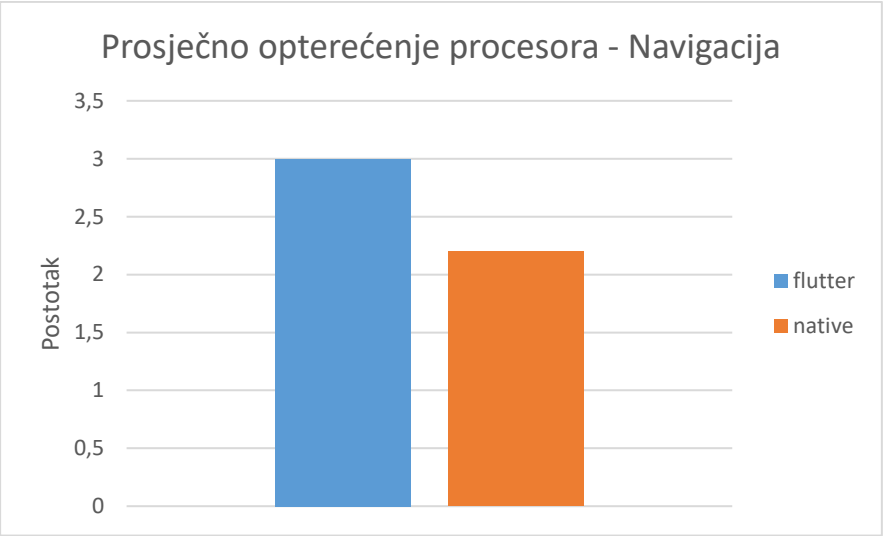
Slika 5.7. Vrijeme odziva pokretanja aplikacija na uređaju iPhone 12



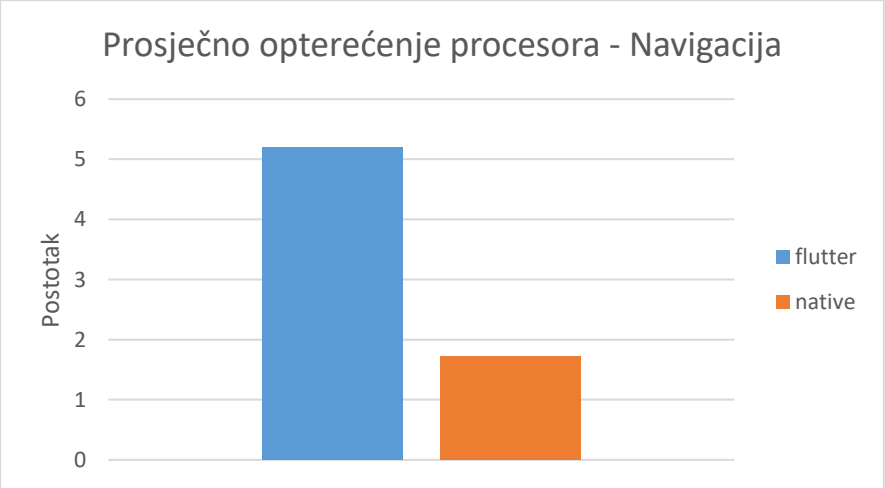
Slika 5.8. Vrijeme odziva pokretanja aplikacija na uređaju iPhone 13 Pro Max

5.1.2. Postotak korištenja procesora i mrežni promet

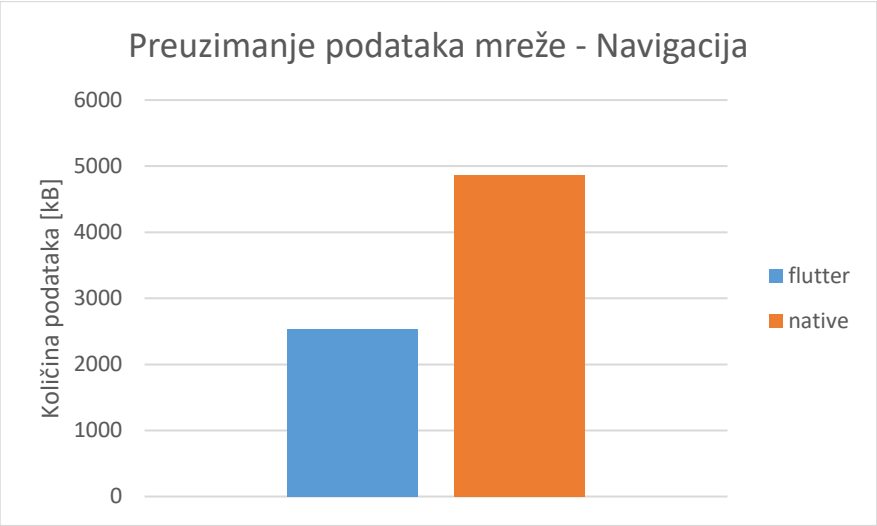
Rezultati testiranje prosječnog postotka korištenja procesora pri navigaciji su prikazani na slici 5.9. za uređaj iPhone 12 te slici 5.10. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* koristila veći udio procesora pri navigaciji u odnosu na aplikaciju u *SwiftUI*. Prosječna količina preuzimanja podataka s mreže pri navigaciji je prikazana na slikama 5.11. i 5.12, iz kojih se vidi da je pri navigaciji na uređaju iPhone 12, mrežni promet *SwiftUI* aplikacije bio duplo veći u odnosu na *Flutter* aplikaciju, dok je pri testiranju na iPhone 13 Pro Max mrežni promet SwiftUI bio za 20% manji. Uzimajući to u obzir, sliku 5.9 prikazuje realnije odnose postotka korištenja procesora između *SwiftUI* i *Flutter* aplikacije te se je vidljivo sa slike 5.9. da je stvarni odnos postotka korištenja procesora u *SwiftUI* aplikaciji od 2 do 3 puta manji u odnosu na *Flutter* aplikaciju.



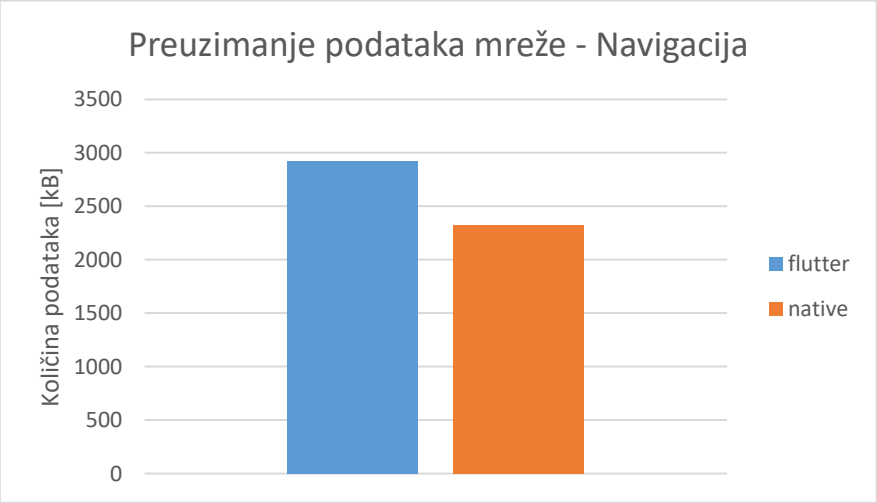
Slika 5.9. Prosječno opterećenje procesora pri navigaciji na uređaju iPhone 12



Slika 5.10. Prosječno opterećenje procesora pri navigaciji na uređaju iPhone 13 Pro max

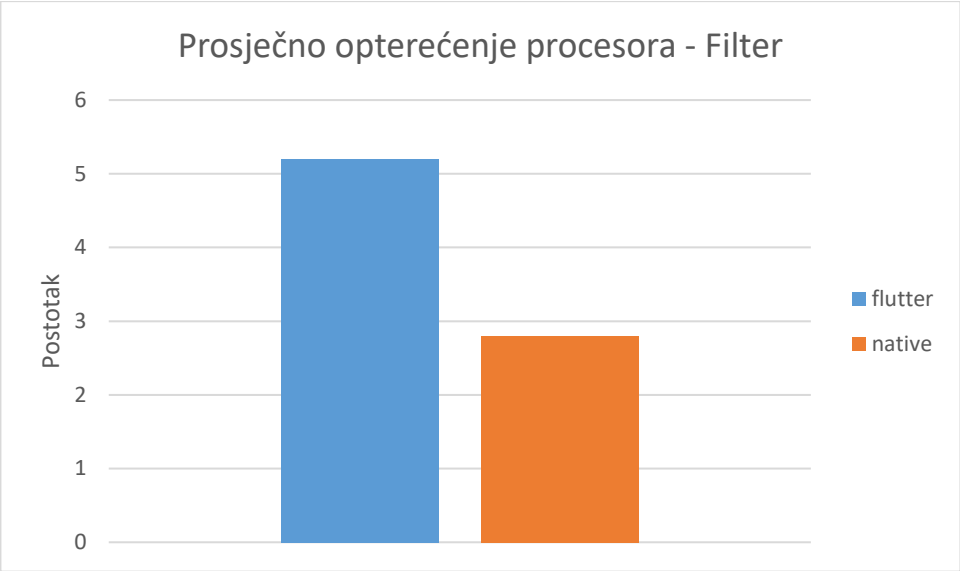


Slika 5.11. Preuzimanje podataka s mreže pri navigaciji na uređaju iPhone 12

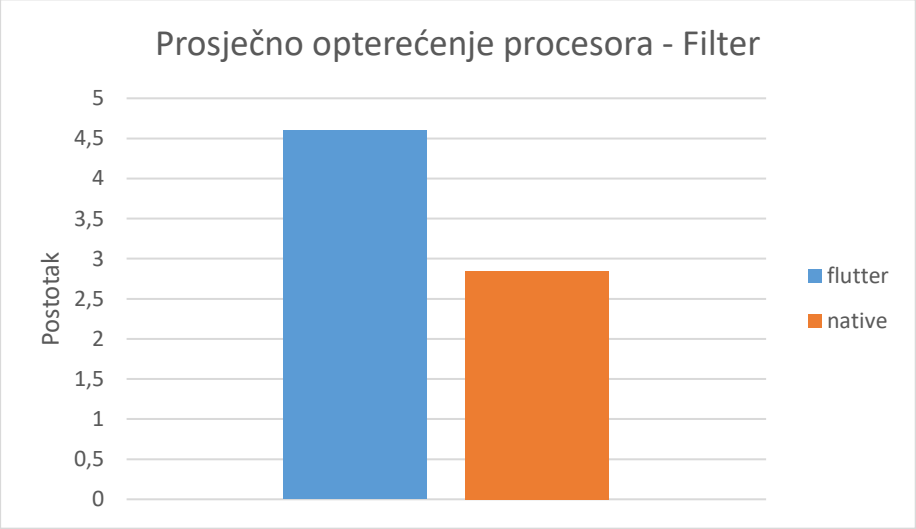


Slika 5.12. Preuzimanje podataka s mreže pri navigaciji na uređaju iPhone 13 Pro Max

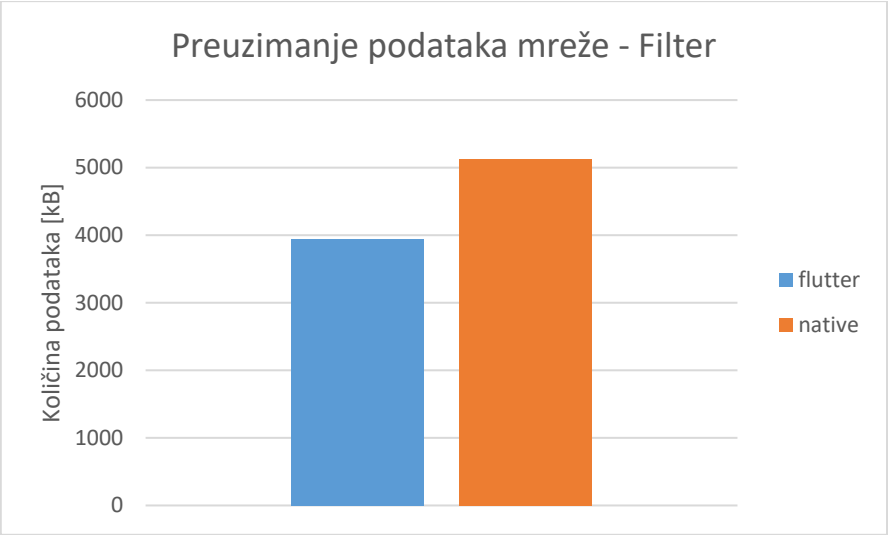
Rezultati testiranje prosječnog postotka korištenja procesora pri filtriranju su prikazani na slici 5.13. za uređaj iPhone 12 te slici 5.14. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* koristila duplo više procesora pri filtriranju u odnosu na aplikaciju u *SwiftUI*. Prosječna količina preuzimanja podataka s mreže pri navigaciji je prikazana na slikama 5.15. i 5.16, iz kojih se vidi da je pri filtriranju na uređaju iPhone 12, mrežni promet *SwiftUI* aplikacije bliži *Flutter* aplikaciji nego pri testiranju na iPhone 13 Pro Max vidljivim na slici 5.16. S obzirom na to može se reći da sliku 5.13. bolje opisuje realne vrijednosti postotka korištenja procesora.



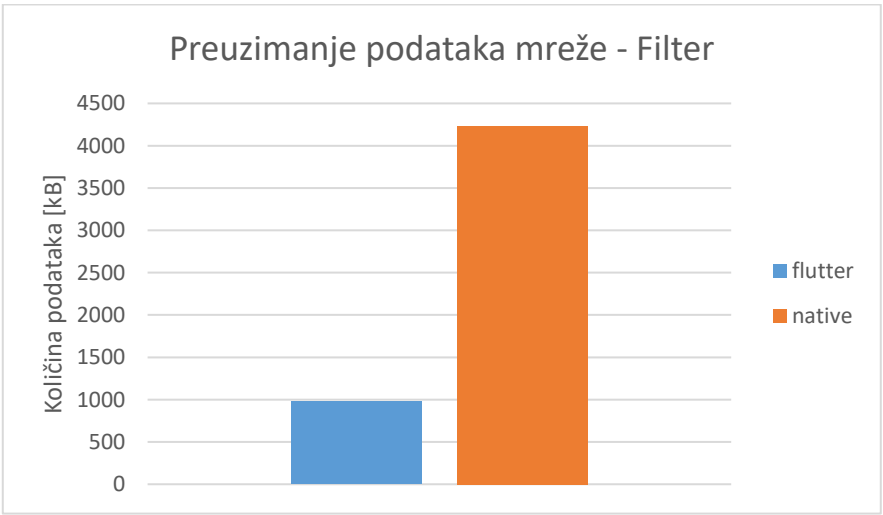
Slika 5.13. Prosječno opterećenje procesora pri filtriranju slike na uređaju iPhone 12



Slika 5.14. Prosječno opterećenje procesora pri filtriranju slike na uređaju iPhone 13 Pro Max

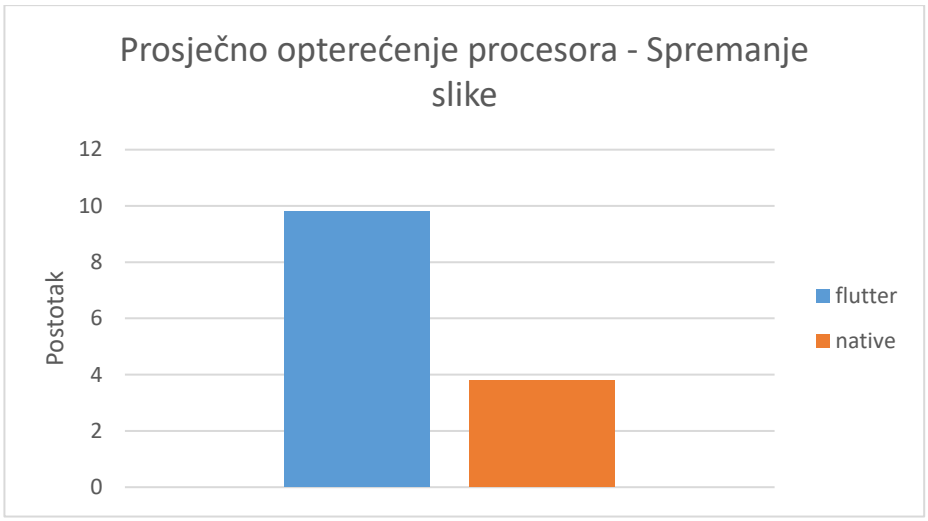


Slika 5.15. Preuzimanje podataka s mreže pri filtriranju slike na uređaju iPhone 12

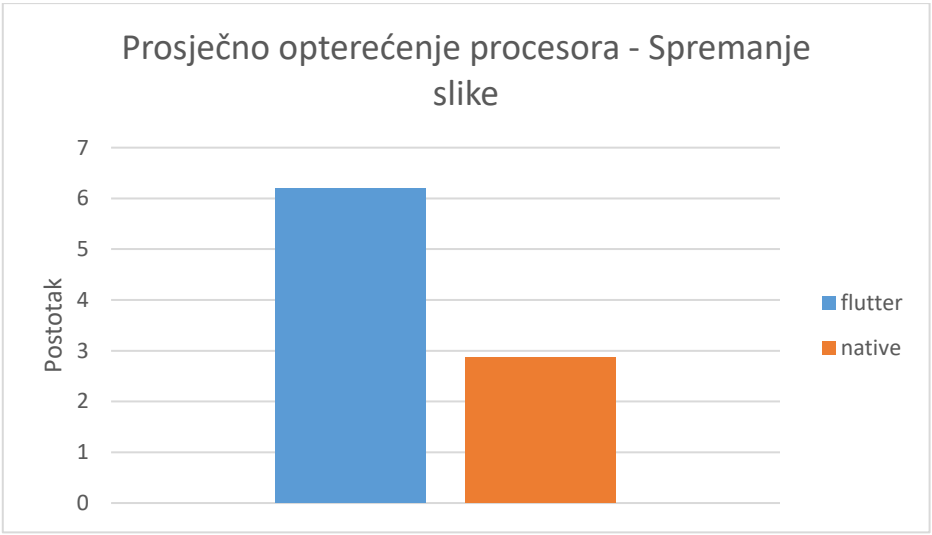


Slika 5.16. Preuzimanje podataka s mreže pri filtriranju slike na uređaju iPhone 13 Pro Max

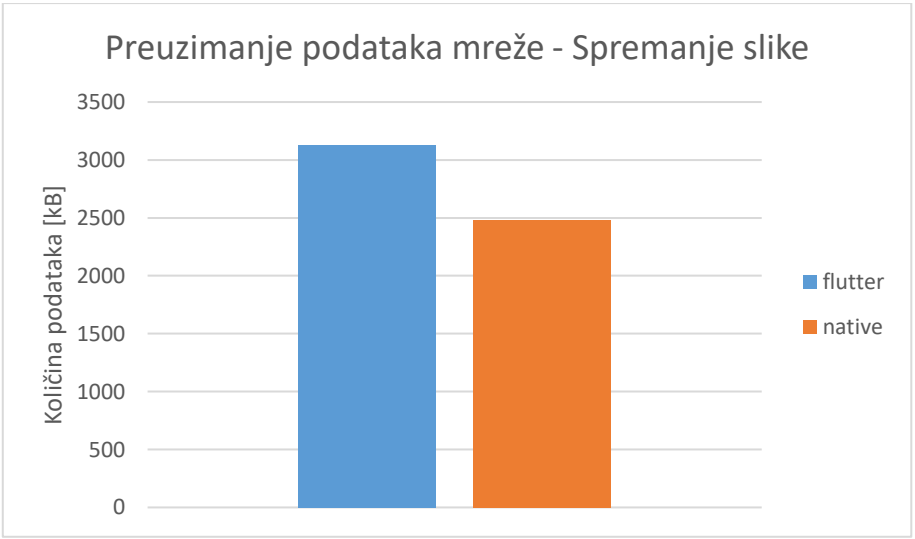
Rezultati testiranje prosječnog postotka korištenja procesora pri spremanju slike su prikazani na slici 5.17. za uređaj iPhone 12 te slici 5.18. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* koristila duplo veći postotak zauzeća procesora odnosu na aplikaciju u *SwiftUI*.



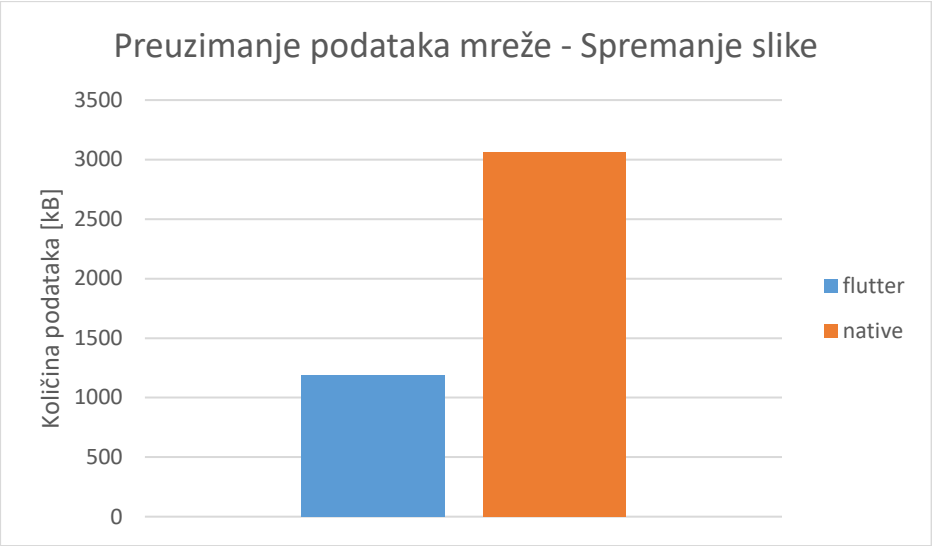
Slika 5.17. Prosječno opterećenje procesora pri spremanje slike na uređaju iPhone 12



Slika 5.18. Prosječno opterećenje procesora pri spremanju slike na uređaju iPhone 13 Pro Max

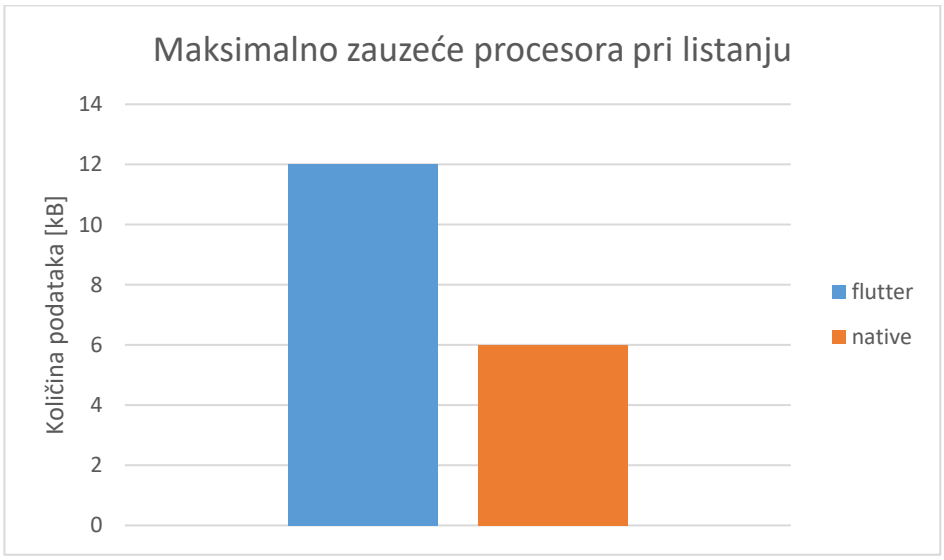


Slika 5.19. Preuzimanje podataka s mreže pri spremanju slike na uređaju iPhone 12

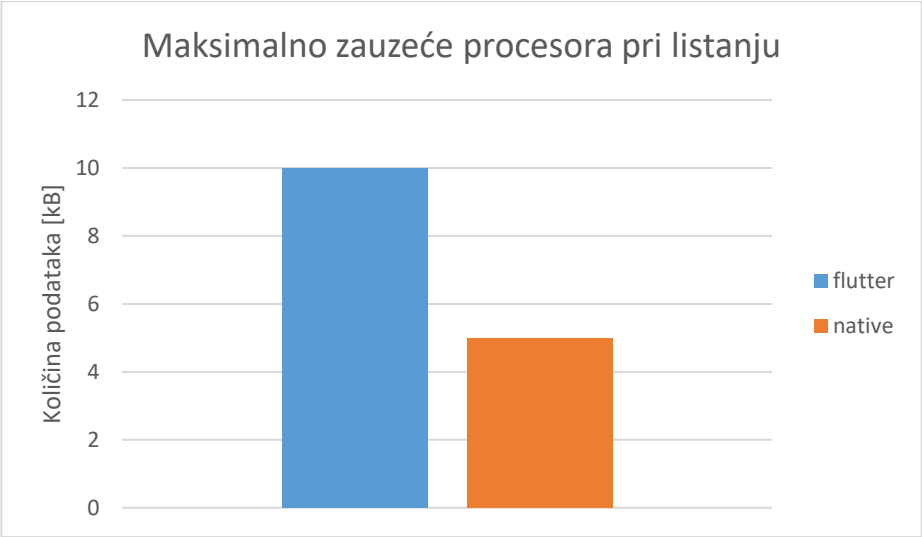


Slika 5.20. Preuzimanje podataka s mreže pri spremanju slike na uređaju iPhone 13 Pro Max

Rezultati testiranje prosječnog postotka korištenja procesora pri listanju kroz listu slika su prikazani na slici 5.21. za uređaj iPhone 12 te slici 5.22. za uređaj iPhone 13 Pro Max. Sa slika je vidljivo da aplikacija napravljena u *Flutteru* koristila duplo više procesora pri listanju kroz beskonačnu listu slika u odnosu na aplikaciju u *SwiftUI*.



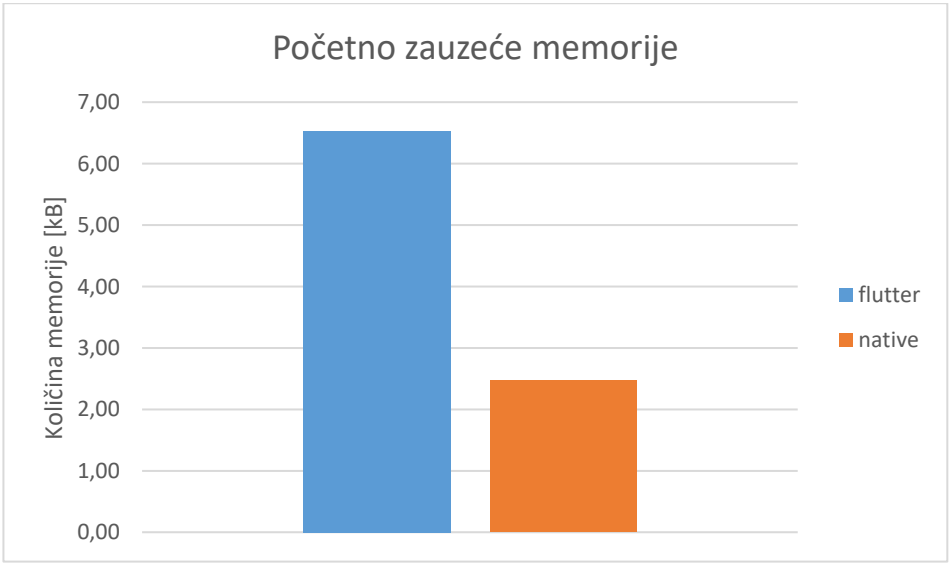
Slika 5.21. Prosječno opterećenje procesora pri listanju beskonačne liste na uređaju iPhone 12



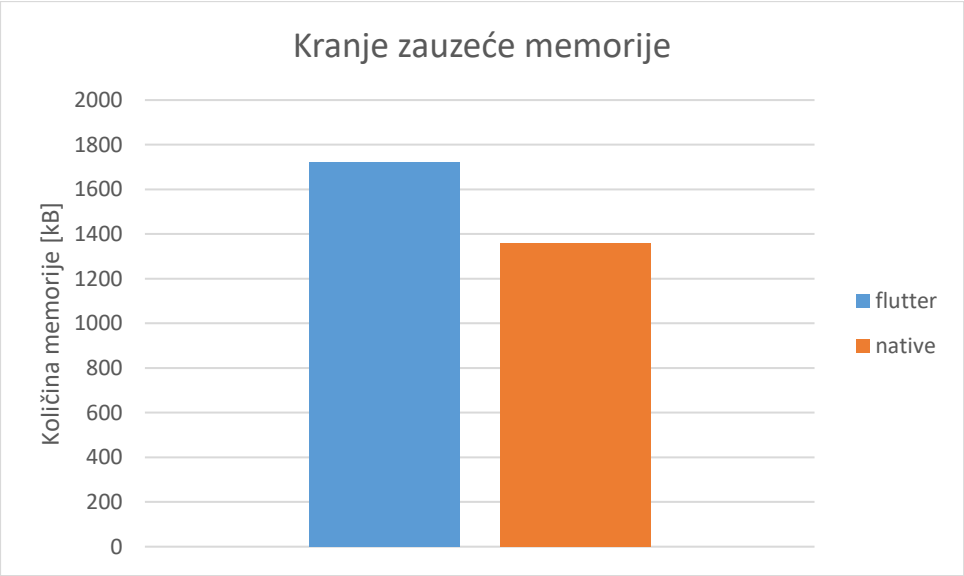
Slika 5.22. Prosječno opterećenje procesora pri listanju beskonačne liste na uređaju iPhone 13 Pro Max

5.1.3. Zauzeće memorije

Rezultati testiranja zauzeća memorije u kilobajtima nakon pokretanja aplikacije su prikazani na slici 5.23., dok je zauzeće memorije na kraju korištenja aplikacije prikazano na slici 5.24. Početno zauzeće memorije je nisko u oba slučaja te se prema slici 5.23 vidi da aplikacija u *Flutter* zauzima više memorije u samom startu nego aplikacija u Swift UI. Brojčana početna razlika u zauzeću memorije je jako mala, svega 4-5 kilobajta. Prema slici 5.24., zauzeće memorije pri završetku korištenja aplikacija je 20% poveća u *Flutter* aplikaciji u odnosu na aplikaciju napravljenu u *SwiftUI*.



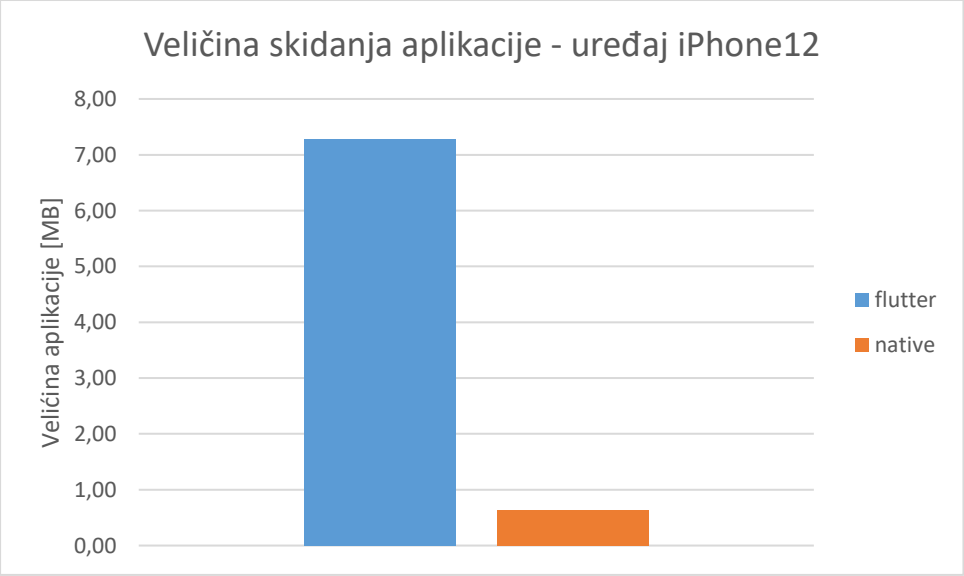
Slika 5.23. Zauzeće memorije pri startu aplikacije



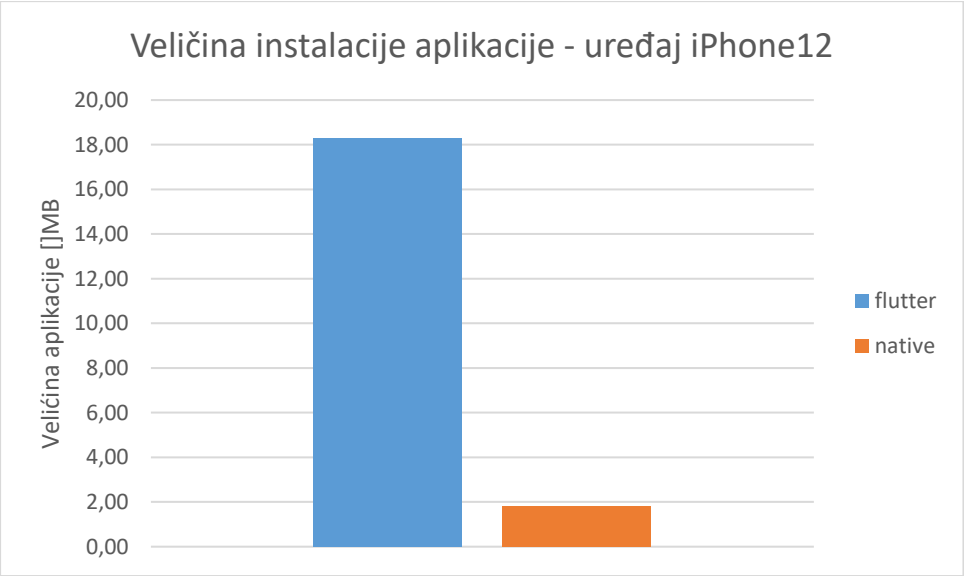
Slika 5.24. Zauzeće memorije pri završetku aplikacije

5.2. Veličina aplikacija

Rezultati testiranja omjera veličina aplikacija pri preuzimanju su prikazani na slici 5.25., dok su veličine aplikacija pri instaliranju prikazane na slici 5.26. Na slici 5.25. je vidljivo da je veličina *Flutter* aplikacije pri preuzimanju čak sedam puta veća nego aplikacija u *SwiftUI*. Sa slike 5.26. se vidi da je aplikacija u Flutteru šest puta veća pri instalaciji nego aplikaciji u *SwiftUI*.



Slika 5.25. Veličina aplikacije [MB] pri preuzimanju



Slika 5.26. Veličina aplikacije [MB] pri instaliranju

5.3. Analiza rezultata testiranja performansi

Iz dobivenih rezultata na slikama 5.1 do 5.26 vidi se da su performanse nativne aplikacije bolje nego performanse *Flutter* aplikacije u svim primjerima, osim navigacije na novi zaslon.

Rezultati testiranja u [13], pri usporedbi nativne iOS aplikacije i višeplatformskih rješenja kao *Famo.us* i *Ionic*, pokazuju da vremena odziva pri prelasku na novu stranicu mogu biti bolja ili jednaka u usporedbi s nativnim iOS. Ovaj ishod nalikuje rezultatima dobivenim testiranjem implementirane *Flutter* i *SwiftUI* aplikacije gdje se promatra brže vrijeme odziva višeplatformske aplikacije u odnosu na nativnu.

Razlika vremena odziva pri obavljanju radnji kao što je listanje kroz beskonačnu listu slika, primjenu filtera na slici te spremanje slike na uređaj su znatno kraća pri aplikaciji napisanoj u *SwiftUI*.

Pri spremanju slika na uređaj *Flutter* aplikacija je sporija u odnosu na *SwiftUI* aplikaciju. Za primjenu filtera slika aplikacija u SwiftUI ne primjećuje znatni porast pri većim veličinama slika, dok se u *Flutter* aplikaciji jasno vidi porast pri porastu veličine slike, kao što je najbolje vidljivo na slici 5.4.

Postotak zauzeća procesora pri obavljanju operacija u *Flutter* aplikaciji je znatno veći te iznosi u prosjeku duplo više nego postotak zauzeća u odnosu na nativnu aplikaciju u *SwiftUI*. Slična promatranja su opažena u [1], gdje je postotak zauzeća procesora hibridne aplikacije u *Cordova*

alatu za razvoj programa bio 106% veći u odnosu na nativnu Android aplikaciju te u [11] gdje je zauzeće procesora u *Ionic* alatu za razvoj programa bio skoro tri puta veći nego u nativnoj aplikaciji. Prema slikama 5.23. i 4.54. vidi se da *Flutter* aplikacija koristi više memorije nego nativna aplikacija u *SwiftUI*, pri čemu je zauzeće memorije u *Flutter* aplikaciji 20% veće u odnosu na nativnu aplikaciju. Veće zauzeće memorije u višeplatformskim aplikacijama je također primijećeno u [1] gdje je zabilježeno čak do 73% veće zauzeće memorije u *Cordova* aplikaciji.

Vrijeme odziva pri pokretanju aplikacije vidljivo na slikama 5.7 i 5.8, traje 43% duže na uređaju iPhone 12 te 29% na urešaju iPhone 13 Max u *Flutter* aplikaciji nego u nativnoj. Velika razlika u omjeru vremena pokretanja aplikacija je dobivena radi samih razlika specifikacija uređaja na kojima su testovi provedeni. Slično ponašanje je dobiveno pri testiranju vremena pokretanja nativnih i višeplatformskih aplikacija u [11], gdje je primijećeno znatno veće vrijeme pokretanja pri višeplatformskim aplikacijama. U [11] gdje je očitano čak 382% veće vrijeme odziva pokretanja aplikacije u *Ionic* alatu za razvoj programa u odnosu na nativnu iOS aplikaciju.

Veličine aplikacija pri preuzimanju na uređaj i instalaciji na uređaj su znatno veće u *Flutter* aplikaciji nego u nativnoj *SwiftUI*. Prema slici 5.25. očitana je čak 1155% veća veličina *Flutter* aplikacije pri skidanju te 1011% veća veličina aplikacije pri instalaciji. *Flutter* aplikacija intenzivno koristi biblioteke, što zasigurno doprinosi krajnjoj veličini aplikacije.

5.4. Analiza implementacije i korisničkog iskustva

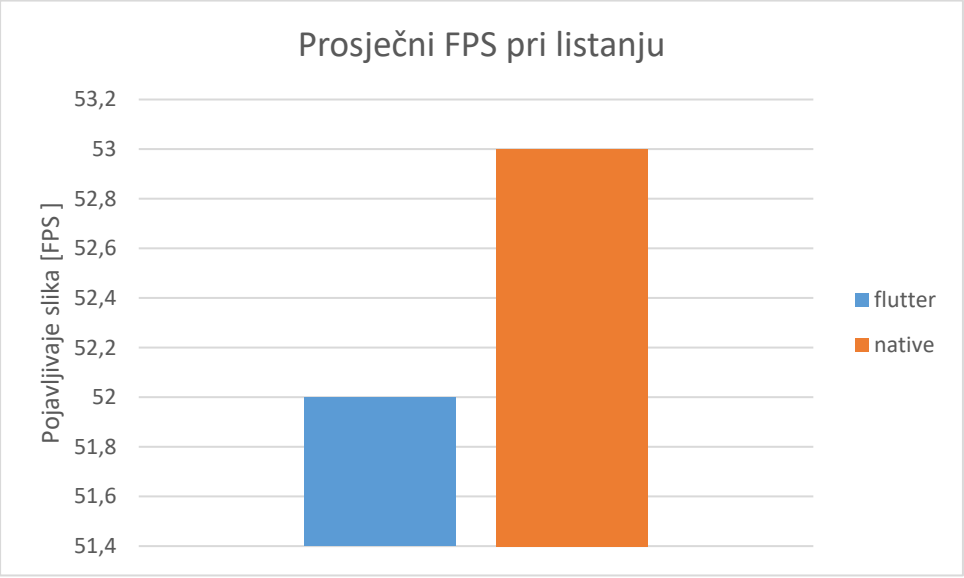
Vremena odziva do 100 milisekundi se dožive trenutačnim sa strane korisnika, dok su vremena odziva do par sekundi dozvoljena ako se dešavaju rijetko [11]. Uzimajući to u obzir, možemo vrednovati u kojim slučajevima će razlike u performansama *Flutter* i nativne aplikacije biti opaženi.

Prema slikama 5.1 i 5.2, vidi se da su vremena odziva pri navigaciji u *Flutter* i *SwiftUI* aplikaciji u rangu do 100 milisekundi te se prema tim podacima i iz iskustva korištenja aplikacije vidno ne primijeti razlika pri navigaciji na stranicu detalja slike.

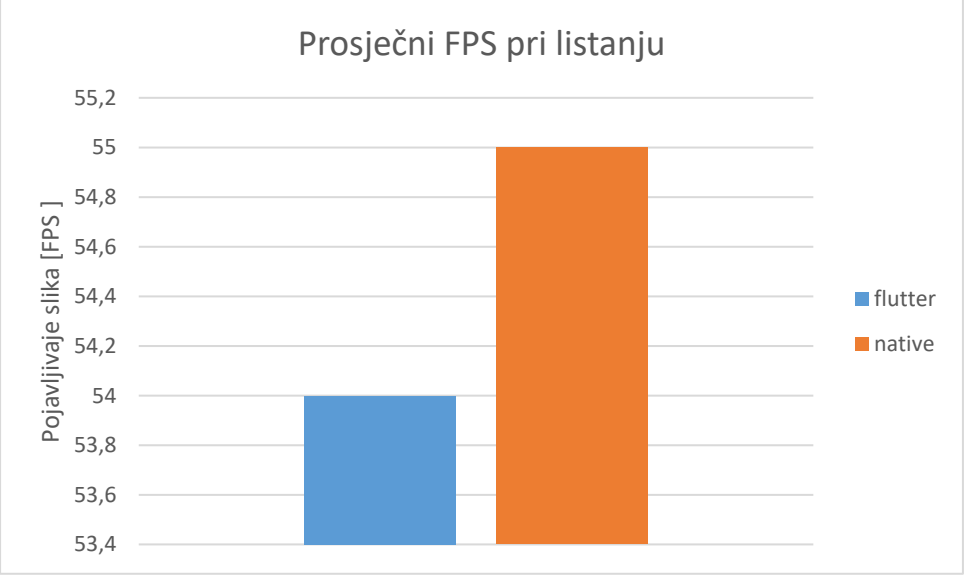
Prema slikama 5.3 i 5.4, vidi se da su odzivna vremena funkcije filtera slike u *SwiftUI* aplikaciji u rangu do 100 milisekundi, dok je vrijeme odzivau *Flutter* aplikaciji u rangu od 200 do 600 milisekundi. Prema tim podacima te iz korištenja aplikacija, primjenjivanje filtera u *SwiftUI* aplikaciji se ne vizualno ne primijeti, dok u *Flutter* aplikaciji se primijeti malo kašnjenje pri primjenjivanju filtera na slici.

Odzivna vremena pri spremanju slika, vidljiva na slikama 5.5 i 5.6, su za *Flutter* i za *SwiftUI* aplikaciju u rangu od par sekundi. Prema tome, pri korištenju aplikacije primijeti se znatno kašnjenje dok se spremanje slike ne obavi te s obzirom na to da su obje aplikacije u istom rangu vremena odziva, pri korištenju aplikacije se ne primijeti znatna razlika između *Flutter* i *SwiftUI* aplikacije.

Korisničko iskustvo listanja kroz beskonačnu listu slika se ne može mjeriti vremenom odziva, stoga su se prikupili podaci o brzini prikazivanja prikaza na zaslonu (FPS) pri listanju, čiji su rezultati vidljivi na slici 5.27 za uređaj iPhone 12 i slici 5.28 za uređaj iPhone 13 Pro Max.



Slika 5.27. Prosječni FPS pri listanju na uređaju iPhone 12



Slika 5.27. Prosječni FPS pri listanju na uređaju iPhone 13 Pro Max

Iz rezultata na slikama 5.27 i 5.28, vidi se da je FPS u *Flutter* i *SwiftUI* aplikaciji pri listanju kroz beskonačnu listu slika ostao visok, pri čemu se može primijetiti da je FPS u *SwiftUI* aplikaciji veći u odnosu na *Flutter* aplikaciju. Prema podacima na slikama 5.27 i 5.28 te pri korištenju aplikacije, vidi se da sa strane korisničkog iskustva nema znatne razlike u listanju pri korištenju *Flutter* i *SwiftUI* aplikacije.

6. ZAKLJUČAK

Ovaj rad analizirao je mogućnosti i izazove nativnog i višeplatformskog razvoja mobilnih aplikacija sa stajališta programera, korisnika i performansi. Naglasak je stavljen na razvoj mobilne aplikacije za iOS platformu koristeći *Flutter* za višeplatformski razvoj te alat *SwiftUI* za nativni razvoj. Uspoređene su performanse višeplatformske i nativne aplikacije te je provedena analiza njihovog utjecaja na korisnika i na odabir prikladnog pristupa od strane programera. Za ostvareno programsko rješenje u skladu s definiranim funkcionalnim i nefunkcionalnim zahtjevima, obavljeno je testiranje nativnog i višeplatfromskog programskog rješenja s naglaskom na testiranje performansi.

Prema analizi dobivenih rezultata, aplikacija izrađena u *Flutteru* pruža mogućnost izrade jedne aplikacije za dvije platforme, pri čemu se smanjuju troškovi izrade, ali pri lošijim performansama u odnosu na nativnu aplikaciju u *SwiftUI.* Nativna aplikacija izrađena u *SwiftUI* nema vizualne indikacije zastajkivanja na zaslonu. Kod *Flutter* aplikacije zastajkivanje se može primijetiti kod zahtjevnih operacija, pri čemu vrijeme odziva operacija ostaje u dopuštenim granicama. Nativna *SwiftUI* aplikacija i višeplatformska *Flutter* aplikacija ispunjavaju potreba i očekivanja korisnika i programera, ali niti jedan pristup se ne može smatrati savršenim rješenjem. Preporuka na osnovu ovog rada je, ako mobilna aplikacija ima velike zahtjeve za performansama, nativna *SwiftUI* aplikacija bila bi dobro rješenje, ali ako performanse nisu zahtjevne te je iskorištavanje istog koda ili smanjenje troškova prioritet, višeplatformska *Flutter* aplikacija se čini kao bolje rješenje.

LITERATURA

[1] Peixin Que, Xiao Guo, Maokun Zhu, A Comprehensive Comparison Between Hybrid and Native App Paradigms, Laboratory of Network Applications and New Technology Communication University of China, International Conference on Computational Intelligence and Communication Networks, 18-20. srpanj 2016, Beijing, China, str. 611-614.

[2] Carmen Ang, The Top Downloaded Apps in 2022 https://www.visualcapitalist.com/top-downloaded-apps-2022/#:~:text=According%20to%20the%20report%2C%20total,increase%20compared%20to%20Q1%202021, 4. svibanj 2022.

[3] Andrea Knezovic, The State of Mobile App Market + Predictions for 2022 & Beyond, Udonis, https://www.blog.udonis.co/mobile-marketing/mobile-apps/mobile-app-market-forecast, 7. siječanj, 2022.

[4] Google Play Store, Google Ads, https://play.google.com/store/apps/details?id=com.google.android.apps.adwords&hl=en&gl=US pristupio srpanj 2022.

[5] Google Play Store, Reflectly, https://play.google.com/store/search?q=reflectly&c=apps&hl=en&gl=US, pristupio srpanj 2022.

[6] App Store, Weather, https://apps.apple.com/us/app/weather/id1069513131, pristupio srpanj 2022.

[7] Alfian Losari, Corona Virus Stats & Advices App with SwiftUI, https://github.com/alfianlosari/CoronaVirusTrackerSwiftUI, pristupio srpanj 2022.

[8] Lvivity, Functional and Non-functional Requirements for Mobile App: What's the Difference?, https://lvivity.com/functional-and-non-functional-requirements, pristupio srpanj 2022.

[9] Igor Tkachenko, Functional vs Non-Fucntional Requirements: Main Diferences & Examples, https://theappsolutions.com/blog/development/functional-vs-non-functional-requirements/, pristupio srpanj 2022.

[10] Clearbridge, How to Write an Effective Mobile App Product Requirements Document, https://clearbridgemobile.com/how-to-build-a-mobile-app-requirements-document/, pristupio srpanj 2022.

[11] Michiel Willocx, Jan Vossaert, Vincent Naessens, Comparing performance parameters of mobile app development strategies, KU Leuven, Technology Campus Ghent, IEEE/ACM International Conference on Mobile Software Engineering and Systems, 2016, Ghent, Belgium, str. 38-47.

[12] Apptim, What is Apptim?, https://help.apptim.com/en/articles/2965623-what-is-apptim, pristupio srpanj 2022.

[13] Xiaoping Jia, Aline Ebone, Yongshan Tan, A Performance Evaluation of Cross-Platform Mobile Application Development Approaches, ACM/IEEE 5th International Conference on Mobile Software Engineering and Systems, 2018, Chicago, IL, USA, str. 92-93.

[14] Lisandro Delía, Nicolás Galdamez, Leonardo Corbalan, Patricia Pesado, Pablo Thomas, Approaches to Mobile Application Development: Comparative Performance Analysis, Computing Conference 2017, 18- 20. srpanj 2017, London, UK, str. 652-659.

[15] Google, Flutter, https://flutter.dev/, pristupio srpanj 2022.

[16] Google, Dart, https://Dart.dev/overview#platform, pristupio srpanj 2022.

[17] Google, Flutter docs, arhitectular overview, https://docs.flutter.dev/resources/architectural-overview, pristupio srpanj 2022.

[18] Geeks for geeks, https://www.geeksforgeeks.org/what-is-widgets-in-flutter/, 4. srpanj 2022.

[19] Flutter docs, inside Flutter, https://docs.flutter.dev/resources/inside-flutter, pristupio srpanj 2022.

[20] Apple, SwiftUI tutorials, Introducing SwiftUI, https://developer.apple.com/tutorials/swiftui, pristupio srpanj 2022.

[21] Apple, About Swift, https://www.swift.org/about/, pristupio srpanj 2022.

[22] Cocoacast, What is SwiftUI, https://cocoacasts.com/swiftui-fundamentals-what-is-swiftui#:~:text=SwiftUI%20is%20Apples%20brand%20new,is%20a%20cross%2Dplatform%20framework, pristupio srpanj 2022.

[23] Apple, App Structure and Behavior, https://developer.apple.com/documentation/swiftui/app-structure-and-behavior, pristupio srpanj 2022.

[24] Apple, SwiftUI View, https://developer.apple.com/documentation/swiftui/view/, pristupio srpanj 2021.

[25] Apple, Run Loop, https://developer.apple.com/documentation/foundation/runloop, pristupio srpanj 2022.

[26] Rens Breur, The SwiftUI render loop, https://rensbr.eu/blog/swiftui-render-loop/#:~:text=Just%20like%20UIKit%20%2C%20SwiftUI%20is,render%20loop%20of%20an%20application., 9 svibanj 2022.

[27]. Apple, Run Loops, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html, pristupio srpanj 2022.

[28] Can Balkaya, Animations in SwiftUI: Get to Know Transactions, https://betterprogramming.pub/animation-in-swiftui-get-to-know-transactions-7cd57cfb299f, 4 siječanj 2021.

SAŽETAK

U ovom diplomskom radu analizirane su mogućnosti i izazovi nativnog i višeplatformskog razvoja mobilnih aplikacija, programske arhitekture, alate i mogućnosti testiranja. U radu je naglasak na razvoju mobilne aplikacije za iOS platformu koristeći *Flutter* za višeplatformski razvoj te alat *SwiftUI* za nativni razvoj. Nadalje, cilj rada je usporediti performanse višeplatformske i nativne aplikacije te analizirati njihov utjecaj na korisnika i na odabir prikladnog pristupa od strane programera. Za ostvareno programsko rješenje u skladu s definiranim funkcionalnim i nefunkcionalnim zahtjevima, obavljeno je testiranje nativnog i višeplatfromskog programskog rješenja s naglaskom na testiranje performansi. Rezultati testiranja pokazuju da je pogodno koristiti *Flutter* pri projektima gdje je prioritet smanjenje troškova te dijeljenje koda, dok je pogodnije koristiti *SwiftUI* na projektima gdje su performanse prioritet.

Ključne riječi: iOS, mobilna aplikacija, performanse, testiranje, višeplatformski razvoj.

ABSTRACT

Analysis of the possibilities of cross-platform and native development of mobile applications

This thesis analyzed the capabilities and challenges of native and cross-platform mobile application development, programming architectures, tools and testing capabilities. In the thesis, *Flutter* was used as the cross-platform tool, and *SwiftUI* as the native tool for mobile iOS application development. The goal of this thesis is to compare the performance of the cross-platform and native application and the performance influence on the end user, for the purpose of choosing the most appropriate development approach. For the implemented solution with the appropriate functional and non-functional requirements, testing of the native and cross-platform implementation was conducted with an emphasis on performance. Results show that is more appropriate to use *Flutter* in projects where the priority is lowering the budget and code sharing, while it is more suitable to use *SwiftUI* in the projects where the performance is the priority.

Key words: cross-platform application, mobile application, performance, testing

ŽIVOTOPIS

Ivan Štajcer rođen je 26. prosinca 1998. godine u Zagrebu. Osnovnu školu je upisao i završio u Slavonskom Brodu. Nakon završetka osnovne škole upisuje srednju Tehničku školu u Slavonskom Brodu, smjer elektrotehničar, koju završava 2017. godine. Tokom iste godine upisuje preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.

_____

Potpis autora

PRILOZI

Prilog 1. Diplomski rad u datoteci docx

Prilog 2. Diplomski rad u datoteci pdf

Prilog 3. Programsko rješenje *Flutter* aplikacije: https://github.com/igniti0n/picmore_flutter

Prilog 4. Programsko rješenje nativne *SwiftUI* aplikacije: https://github.com/igniti0n/picmore_ios