SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I

INFORMACIJSKIH TEHNOLOGIJA OSIJEK (Title stil)

**Sveučilišni diplomski studij**

**ANALIZA MOGUĆNOSTI VIŠEPLATFORMSKOG NATIVNOG RAZVOJA MOBILNIH APLIKACIJA**

Diplomski rad

Ivan Štajcer

Osijek, 2022.

##### SADRŽAJ

[1. UVOD 1](#_Toc112653904)

[1.1. Zadatak diplomskog rada 2](#_Toc112653905)

[2. PRIKAZ STANJA U PODRUČJU I IZAZOVI U RAZVOJU MOBILNIH APLIKACIJA 3](#_Toc112653906)

[2.1. Izazovi u razvoju mobilnih aplikacije 3](#_Toc112653907)

[2.2. Trenutno stanje razvoja mobilnih aplikacija 4](#_Toc112653908)

[2.3. Primjeri aplikacija 5](#_Toc112653909)

[3. MODEL I GRAĐA MOBILNE APLIKACIJE 8](#_Toc112653910)

[3.1. Funkcijski i nefunkcijski zahtjevi na aplikaciji 8](#_Toc112653911)

[3.1.1. Funkcijski zahtjevi 8](#_Toc112653912)

[3.1.2. Nefunkcijski zahtjevi 9](#_Toc112653913)

[3.2. Građa mobilne aplikacije 9](#_Toc112653914)

[3.3. Testiranje u višeplatformskom razvoju mobilnih aplikacija 11](#_Toc112653915)

[3.3.1. Testni slučajevi 11](#_Toc112653916)

[4. PROGRAMSKO RIJEŠENJE NATIVNE I VIŠEPLATFORMSKE APLIKACIJE 15](#_Toc112653917)

[4.1. Korištene tehnologije, alati i jezici 15](#_Toc112653918)

[4.1.1. Flutter 15](#_Toc112653919)

[4.1.1.1. Arhitektura 15](#_Toc112653920)

[4.1.1.2. Widgeti i trenutno stanje aplikacije 16](#_Toc112653921)

[4.1.1.3. Propagacija stanja 17](#_Toc112653922)

[4.1.1.4. Prikazivanje korisničkog sučelja 18](#_Toc112653923)

[4.1.1.5. Prikazivanje promjene stanja 21](#_Toc112653924)

[4.1.2. SwiftUI 22](#_Toc112653925)

[4.1.2.1. Razlika sa *UIKit* alatom 22](#_Toc112653926)

[4.1.2.2. Struktura SwiftUI aplikacije 23](#_Toc112653927)

[4.1.2.3. Prikazivanje korisničkog sučelja 24](#_Toc112653928)

[4.1.2.4. Obnavljanje stanja na ekranu 25](#_Toc112653929)

[4.2. Implementacija u Flutteru 26](#_Toc112653930)

[4.2.1. Arhitektura Flutter aplikacije 26](#_Toc112653931)

[4.2.2. Upravljanje stanjima unutar Flutter aplikacije 30](#_Toc112653932)

[4.2.3. Beskonačna lista slika u Flutteru 31](#_Toc112653933)

[4.2.4. Filtriranje i spremanje slike u Flutteru 34](#_Toc112653934)

[4.3. Implementacija u SwiftUI 35](#_Toc112653935)

[4.3.1. Arhitektura SwiftUI aplikacije 35](#_Toc112653936)

[4.3.2. Upravljanje stanjima unutar SwiftUI aplikacije 37](#_Toc112653937)

[4.3.1. Beskonačna lista slika u SwiftUI 38](#_Toc112653938)

[4.3.1. Filtriranje i spremanje slike u SwiftUI 42](#_Toc112653939)

[5. TESTIRANJE I ANALIZA PROGRAMSKOG RIJEŠENJA 44](#_Toc112653940)

[5.1. Testiranje aplikacije 44](#_Toc112653941)

[5.1.1. Nativna aplikacija 44](#_Toc112653942)

[5.1.2. Višeplatformska aplikacija 44](#_Toc112653943)

[5.2. Analiza rezultata 44](#_Toc112653944)

[5.3. Analiza implementacije i korisničkog iskustva 44](#_Toc112653945)

[6. ZAKLJUČAK 45](#_Toc112653946)

[LITERATURA 46](#_Toc112653947)

[SAŽETAK 47](#_Toc112653948)

[ABSTRACT 48](#_Toc112653949)

[ŽIVOTOPIS 49](#_Toc112653950)

[POPIS PRILOGA 50](#_Toc112653951)

# UVOD

Ovaj rad se bavi usporedbom izrade mobilne iOS aplikacije koristeći noviju višeplatformsku tehnologiju Flutter te implementacije identične mobilne aplikacije koristeći najnoviji nativni pristup izrade iOS aplikacija, *SwiftUI*. Cilj ovog rada je što bolje usporediti sve aspekte dvaju pristupa izrade mobilnih aplikacija. S obzirom na tržite i cijene izrada nativnih aplikacije zasebno, dati dobar pregled koliko se isplati raditi posebne nativne aplikacije umjesto odabira višeplatformkog rješenja. Kako bi se rad što više fokusirao na same alate izrade aplikacija, pri samoj izradi spomenutih aplikacija neće se koristiti komuniciranje sa serverom, radi varijabilnih promjena brzina interneta i teškog uspoređivanja dobivenih rezultata. Također, logika unutar aplikacije će se svesti na što niži nivo, kako bi izbacio moguće greške programera pri samom pisanju koda te stavio fokus na same alate. U radu će se usporediti sam postupak izrade mobilne aplikacije koristeći pojedine alate, brzina izrade, korištene arhitekture, rukovanje stanjima, komponente i slično. Također, uspoređivati će se tehničke performanse aplikacije, kao i korisničko iskustvo. U drugom poglavlju će se opisati korištene tehnologije pri izradi pojedinih aplikacija te način na koji prikazuju korisničko sučelje. Treće i četvrto poglavlje će detaljno opisati čitav proces implementacije pojedinih aplikacija, kako bi se što bolje u razmatranje rezultata uzela sama implementacija aplikacije. Peto poglavlja će usporediti sve prethodno navedene stavke napravljenih aplikacija te, uz uzimanje proces izrade i programera u obzir, dati pregled i opis dobivenih rezultata.

## Zadatak diplomskog rada

Uzimajući u obzir aktualne metodologije razvoja programske podrške, programske arhitekture, programske okvire, alate, jezike i mogućnosti testiranja, u diplomskom radu potrebno je opisati i analizirati mogućnosti i izazove nativnog i višeplatformskog razvoja mobilnih aplikacija. Poseban naglasak treba dati na razvoj za iOS platformu, te na mogućnosti nativnog višeplatformskog razvoja u alatu Flutter i pripadajuće tehnologije i jezike (Dart, koji se koristi u Flutteru te Swift, koji se koristi pri izradi iOS aplikacija). Na primjerima prikladnih projekata ispitnih mobilnih aplikacija s odgovarajućim funkcionalnostima i komponentama na korisničkoj i poslužiteljskoj strani pisanima u navedenim jezicima, potrebno je prikazati postupak nativnog i višeplatformskog nativnog razvoja alatom Flutter i obaviti testiranje programske podrške s naglaskom na testiranje performansi. Na temelju toga, potrebno je napraviti usporedbu i analizu pogodnosti navedenih pristupa u razvoju mobilnih aplikacija, dati osvrt na prikladnost programske arhitekture, te analizirati rezultate testiranja s gledišta postignutih performansi.

# PRIKAZ STANJA U PODRUČJU I IZAZOVI U RAZVOJU MOBILNIH APLIKACIJA

## Izazovi u razvoju mobilnih aplikacije

Tokom stvaranja mobilnih aplikacije postoje brojni izazovi koji se razlikuju od aplikacije do aplikacije, radi zahtijeva same biznis logike pojedine aplikacije. Zahjtevi pri izradi mobilnih aplikacija ne proizlaze samo iz potrebne funkcionalnosti koje aplikacija treba izvoditi. Potrebno je implementirati ono što je korisniku najvažnije osim samog problema kojeg aplikacija rješava, korisnici žele imati ugodan izgled aplikacije sa 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. 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 jako bitnu ulogu. Dizajner može napraviti odlično korisničko sučelje sa 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 sa rješenjem problema kojeg ova aplikacija rješava. No, ako programer to ne može spojiti skupa da radi kao cjelina, onako kako je to zamišljeno, ili ne može implementirati neke animacije koje zahtijevaju da se obave programski, ili njegov algoritam ne može ispuniti očekivanja brzine proračuna, korisnici neće biti zadovoljni sa proizvodom. Osim samih vještina i koordiniranja tima pri izradi proizvoda, potrebno je prije svega izabrati alat pri izradi mobilne aplikacije. Ako se radi o aplikaciji za samo jednu platformu, onda ima smisla raditi nativnu aplikaciju u Swift programskom jeziku za iOS ili Kotlin za Anroid. No, ako aplikacija treba biti na Anroid i na iOS uređajima, moguća su dva izbora, nativan razvoj ili višeplatformski razvoj. Sa strane korisnika ili klijenta koji nema tehničku pozadinu, lako ne može uvidjeti problem kod podržavanja aplikacije na Android i na iOS uređajima. Aplikacije je ista, samo je na drugom uređaju, koliki to problem može biti, samo kopirajte aplikaciju na drugu platformu, zar ne? – Ne. Nažalost, ista apliakcija pri nativnom razvoju se mora odvojeno razvijati. Ovaj prolem rješavaju višeplatformska rješenja kao što je Flutter, pri čemu se gubi na perfomansama aplikacije Ako korisnik ne primjeti da je nativna aplikacije pet posto brža, jer se okom to ne vidi, zar je to bitno? Korisniku je više bitno korisničko iskustvo te problem koji riješava aplikacija nego kako je ona napravljena [02]. Također, razvoj višeplatformske aplikacije košta manje nego poseban nativan razvoj iste aplikacije na pojedinoj platformi. Ovaj problem je vrlo aktualan u trenutnom dobu te sa vremenom razlike u performansama višeplatformkih i nativnih aplikacija postaju sve manje. Također, postoje alternativna rješenja kao što su izgradnja višeplatformske mobilne aplikacije sa pojedinim dijelovima obavljenim nativno te ubačeni unutar aplikacije kao biblioteke.

## Trenutno stanje razvoja mobilnih aplikacija

Mobilne aplikacije se razvijaju konstanto i potrebna je stalna inovacija, 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 2 milijuna aplikacija [1, <https://www.blog.udonis.co/mobile-marketing/mobile-apps/mobile-app-market-forecast>]. Također, novac utrošen od strane korisnika za korištenje aplikacije je također tokom godina sve veći, kao što je vidljivo na slici 2.1.

IZNIMNI SLUČAJ, slike

Chart

Description automatically generated

Sl. 2.1. Broj aplikacija (izvor: https://www.blog.udonis.co/mobile-marketing/mobile-apps/mobile-app-market-forecas)

Broj skidanja aplikacija sa trgovina također raste, tako je do sada u prvoj četvrtini 2022. godine, ukupan broj skidanja dosegao iznos od 36.9 bilijuna. Ovo predstavlja povećanje za 1.4% na 2021. godinu.[2, <https://www.visualcapitalist.com/top-downloaded-apps-2022/#:~:text=According%20to%20the%20report%2C%20total,increase%20compared%20to%20Q1%202021>].

## Primjeri aplikacija

Svake godine izlazi sve veći broj odlično napravljenih aplikacija. Dobar primjer aplikacije napravljene koristeći Flutter je Google Ads. Google Ads pomaže pratiti korisničke kampanje oglašavanja preko Google korisničkog računa.

Graphical user interface

Description automatically generated

Sl. 2.2. Google Ads aplikacija

Reflectly je još jedan primjer dobre aplikacije. Sa 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.

Graphical user interface, application

Description automatically generated

Sl. 2.3. Aplikacija Reflectly

Weather apliakcija od Apple firme u svojem novijem izdanju koristi SwiftUI pri izgradi koriničkog sučelja. Weather aplikacija ma ocijenu 3.2 na App Store trgovini te preko četiri tisuće recenzija.

Graphical user interface, application

Description automatically generated

Sl. 2.4. Weather aplikacija

Corona Virus Stats & Advices App je aplikacija te za praćenje najnovijih podataka o statusu trenutne pandemije. Osim statistika, pruža korisniku korisne savijete o tome kako zaštiti svoje zdravlje. Napravljena koristeći SwiftUI.

Graphical user interface

Description automatically generated

Sl. 2.5. Weather aplikacija

# MODEL I GRAĐA MOBILNE APLIKACIJE

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

## 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 zahtijevu na aplikaciju opisuju kakva svojsta rješenja za implementiranje aplikacije imaju, kako će aplikacija raditi i zašto. [0, <https://lvivity.com/functional-and-non-functional-requirements>].

### Funkcijski zahtjevi

Funkcijski zahtijevi uključuju uključuju zahtijeve biznis logike unutar aplikacije, korisničke zahtijeve, administrativne zahtijeve, zahtijeve pri autorizaciji korisnika, vanjske zahtijeve aplikacije i slično. Funkcijski zahtijevu se dijele na slučajeve upotrebe, pri čemu se svaki slučaj dijeli na sudionike, funkcionalnost i cilj slučaja. [0, <https://theappsolutions.com/blog/development/functional-vs-non-functional-requirements/>]. 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 sa formama i slično [0, <https://clearbridgemobile.com/how-to-build-a-mobile-app-requirements-document/>].

Uzimajući u obzir najčešće funkcijske zahtjeve, implementirana aplikacija treba sadržavati:

* Korisnik ima pregled kvalitetnih slika pomoću liste može te 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 o slici.
* Korisnik može primijeniti filter izoštrenja na slici, kako bi promijenio sliku prema želji.
* Korisnik može spremiti filtriranu sliku u galeriju na mobilnom uređaju, kako bi je sačuvao.

### Nefunkcijski zahtjevi

Primjeri nefunkcijskih zahtjeva mobilnih aplikacija uključuju performanse u pogledu vremena trajanja izvršenja nekog procesa, pokretanja aplikacije, korištenje memorije, veličina aplikacije i slično. Uključuju skalabilnost, kao mogućnost primanja više korisnika ili rada sa više podataka. Pouzdanost kao nefunkcijski zahtjev zahtijeva da sa svakim korištenjem aplikacije se dobije isti rezultat pri određenim radnjama. Nefunkcijski zahjevi uključuje responzivnost, što predstavlja povratne informacije korisniku te brzinu povratne informacije [0, <https://clearbridgemobile.com/how-to-build-a-mobile-app-requirements-document/>].

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

* Kada korisnik otvori aplikaciju treba bi vidjeti prikaz slika na ekranu unutar pet sekundi
* Aplikacija mora prikazati svaku promjenu na ekranu bez vidnog zastajkivanja
* Aplikacija mora izgledati funkcionalno na uređajima različitih veličina ekrana
* 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

## Građa mobilne aplikacije

Implementirana mobilna aplikacija se sastoji od početnog ekrana koji prikazuje beskonačnu listu slika dohvaćenih sa se servera (<https://unsplash.com/>). Pojedinačni element unutar liste prikazuje sliku dobivenu sa servera, broj pozitivnih reakcija korisnika na sliku te korisničko ime autora slike, vidljivo na slici 3.1.

Graphical user interface, website

Description automatically generated

Sl. 3.1. Početni ekran aplikacije

Klikom na sliku unutar liste, aplikacije prikazuje stranicu sa detaljima slike koju je korisnik pritisnuo, vidljivo na slici 3.2. Na ekranu je prikaza 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 'slider' pomoću kojeg korisnik mijenja intenzitet filtera primijenjenog na odabranu sliku te se filtrirana slika prikazuje na mjestu orginalne slike. Na ekranu je također prikazan gumb pomoću kojeg korisnik može spremiti filtriranu sliku u galeriju slika na mobilnom uređaju

Graphical user interface

Description automatically generated

Sl. 3.2. Stranica aplikacije sa prikazom detalja odabrane slike

## Testiranje u višeplatformskom razvoju mobilnih aplikacija

Testiranje je obavljeno koristeći iste alate za testiranje višeplatformske i nativne aplikacije, kako bi vrednovanje bilo isto. Za testiranje je korišten vremenski profili 'XCode' instrumenata, kao i u [04]. Korišten je i alat 'Apptim'. 'Apptim' je alat za mjerenje performansi mobilnih aplikacija koji omogućava validiranje performansi i korisničkog iskustva mobilnih aplikacija [https://help.apptim.com/en/articles/2965623-what-is-apptim]. 'Apptim' interno koristi profile iz aplikacije instrumenata, s tim da pruža dobro korisničko sučelje. Testiranje je poželjno obaviti na što više mobilnih uređaja, kao što je napravljeno u [3] i [4]. 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. Testni uređaji

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

### Testni slučajevi

Testni slučajevi su provedeni cijajući bitne stavke s obzirom na ograničene resurse na mobilnim uređajima, zahtjevima mobilnih aplikacije te već provedenih testova performansi mobilnih aplikacija.

#### Vrijeme odziva

Vremena odziva radnji unutar aplikacije ispod 100 milisekundi se čine trenutačna sa pogleda korisnika. Vremena u trajanju do jedne sekunde ili par se smatraju dozvoljenima ako se ne događaju često. Sva odzivna vremena iznat toga narušavaju korisničko iskustvo [04]. 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 [04]. Također, provedena su mjerenja vremena odziva za primjenjivanje filtera na sliku i spremanja slike na uređaj.

#### Postotak korištenja CPU

Pri mjerenju CPU mjeri se postotak ukupnog kapaciteta CPU uređaja koji je korišten unutar određenog intervala vremena. Ako mobilna aplikacija koristi preveliki postotak CPU, postoji šansa da negativno utječe na ostale procese na uređaju [04]. Provedeno je mjerenje postotka korištenja CPU kroz čitavu aplikaciju, kao i u [04] te je provedeno mjerenje postotka korištenja CPU u određenim radnjama aplikacije, kao i u [01], [02], [03] te [04]. Mjerenje CPU nije trivijalan postupak, jer je potreban dovoljno mali vremenski uzorak, alati pri mjerenju iOS aplikacija dozvoljavaju ograničen vremenski uzorak CPU [04].

#### Memorija

Korištena memorija se odnosi 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 [04]. Uzimajući u obzir memoriju kao bitnu stavku pri izradi mobilnih aplikacija te provedena mjerenja u [02], [03] i [04], provedeno je mjerenje količie zauzeća memorije pri početnom stanju aplikacije te pri kraju testiranja.

#### Veličina aplikacije

Velike aplikacije su manje instalirane, više su vjerovatnije da će biti izbrisane sa uređaja te trebaju više vremena pri skidanju i instalaciji na uređaju [https://www.farfetchtechblog.com/en/blog/post/app-size-matters-i/#:~:text=Large%20mobile%20apps%20are%20less,big%20apps%20mean%20bad%20reputation.] Uzimajući u obzir mjerenja veličine aplikacije u [02], provedena su mjerenja veličine aplikacije pri skidanju i instaliranju na uređaj.

#### Mrežni promet

Obavljeno je mjerenje mrežnog prometa aplikacije, kao što je napravljeno i u [02]. Aplikacija ima intenzivni mrežni promet s obzirom da prikazuje više slika velike kvalitete koje preuzima sa servera. Mrežni promet unutar aplikacije nije uspoređen kao što je memorije ili postotak korištenja CPU jer ovisi o utjecajima same mreže i uvijetima u trenutku testiranja. Pregled mrežnog prometa je dan kao faktor kojeg trebauzeti u obzir pri analizi rezultata testiranja, gdje možda može biti razlog povećanog udia korištenja CPU.

# PROGRAMSKO RIJEŠENJE NATIVNE I VIŠEPLATFORMSKE APLIKACIJE

U ovom poglavlju je opisano 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.

## Korištene tehnologije, alati i jezici

### Flutter

Flutter je višeplatformski alat za razvoj softvera koji omogućava korištenje istog koda na različitim operacijskim sustavima kao što su iOS i Android, uz dopuštanje aplikacijama da direktno komuniciraju sa servisima pojedine platforme [1]. Sama 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 optimiziran jezik za razvoj aplikacija na bilo kojoj platformi uz pružanje značajki kao što su „*type saftey“* i „ *sound null saftey“* [2]. Pri višeplatformskom razvoju, Dart pruža prevođenje koda u strojni kod. To se kod nativnih platformi postiže pomoću virtualne mašine 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 prevodilac za razvojno (*dartdevc*) i produkcijsko (*dart2js*) okruženje kako bi preveo kod napisan u Dart programskom jeziku u JavaScript.

#### Arhitektura

Flutter je dizajniran kao višeslojni sustav sastavljen od više međusobno neovisnih biblioteka, pri kojoj svaka ovisi o sloju na kojem se nalazi. Svaki sloj je osmišljen kao nezavisni te ga je moguće totalno zamijeniti sa nekim drugim [3].

Graphical user interface

Description automatically generated

Sl. 2.1. Slojevi arhitekture Flutter-a , izvor: https://docs.flutter.dev/resources/architectural-overview

Svaki operacijski sustav komunicira sa „*Embedder*“ slojem te se on razlikuje ovisno o kojoj se platformi radi. Njegova svrha je komunikacija sa operacijskim sustavom te pakiranje aplikacija na način 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 sa Flutter mašinom napisanom u C/C++ programskom jeziku, koja predstavlja centralni sloj. U njoj se nalaze implementacije središnjeg programskog sučelja Fluttera. Na vrhu sloja se nalazi samo programsko okruženje Fluttera koje se koristi pri razvoju aplikacija. Komunikacija sa mašinom je omogućena pomoću „*dart:ui*“ biblioteke sastavljene od Dart klasa, koje služe kao adapteri klasama napisanim u C/C++ programskom jeziku iz same mašine.

#### Widgeti i trenutno stanje aplikacije

Flutter čitavo korisničko sučelje gardi pomoću widgeta. Widget predstavlja osnovnu jedinicu za izgradnju korisničkog sučelja te je implementiran kao nepromjenjiva klasa, koja pomoću svoje „*build()“* metode opisuje kako trenutno izgleda korisničko sučelje s obzirom na trenutno stanje unutar aplikacije [4]. Više različitih widgeta formiraju sučelje koristeći kompoziciju te tim stvaraju trenutnu sliku aplikacije s obzirom na trenutno stanje. Svaki widget od roditelja dobiva „*context“*, preko čega prima informacije o tome gdje se nalazi unutar formiranog stabla. Prilikom promjene stanja, kao na primjer pri reakciji na interakciju korisnika sa aplikacijom, pojedini widget unutar hijerarhije se može zamijeniti sa drugim te time reflektirati promijenjeno stanje unutar aplikacije.

Diagram

Description automatically generated

Sl. 2.2. Hijerarhija widget-a ,

izvor: https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple

Postoje dvije vrste widgeta:

* „*Stateless widget*“
* „*Stateful widget“*

„*Stateless widget“* ne sadrži nikakvo stanje koje je podložno promjenama. Kao na primjer „*Text“* koji služi za prikazivanje teksta na ekran. „*Stateful widget“* sadrži neki atribut koji je podložan promjeni, koja se može pojaviti usred, na primjer, interakcije korisnika s aplikacijom ili drugih faktora. Trivijalni primjer bi bio početna aplikacija dana pri postavljanju svakog novog projekta, koja sadrži gumb te tekst koji prikazuje koliko je puta gumb pritisnut. Takav widget mijenja stanje, ali s obzirom da widgeti sami po sebi nisu promjenjivi, spremaju takva stanja unutar posebne „*State“* klase. Ta klasa sadrži „*build“* metodu, koja treba biti pozvana nakon svake promjene stanja kako bi se ona reflektirala na zaslonu. Pozivajući metodu „*setState“*, Flutter dobiva informaciju o razlici trenutnog stanja widgeta i onog prikazanog na ekranu te poziva „*build“* methodu „*State“* klase.

#### Propagacija stanja

Kako je widget klasa, moguće mu je predati podatke pomoću konstruktora te pomoću tih podataka implementirati „*build“* metodu. Ovaj pristup je primjenjiv dok se aplikacija sastoji od svega par widget-a. Kako se dodaju ove funkcionalnosti i implementiraju zasloni, postupak prosljeđivanja podataka kroz konstruktore postaje vrlo težak. Stoga, kao rješenje Flutter pruža treći tip widgeta, „*InheritedWidget“*. „*InheritedWidget“* omogućuje jednostavan pristup podacima svim widgetima koji se nalaze ispod unutar hijerarhije stabla.

Diagram

Description automatically generated

Sl. 2.3. „*InheritedWidget“*,

izvor: https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple

Ako „*ExamWidget“* treba podatke od „*StudentState“* widgeta, moguće ih je dobiti preko „*context”* parametra, kojeg svaki widget pruža kao argument svoje „*build“* metode. Prilikom toga, dobiti će stanje od najbližeg widgeta koji se nalazi iznad unutar hijerarhije stabla i ima tip „*StudentState“*. 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.

#### Prikazivanje korisničkog sučelja

Nativne aplikacije, kao ona primjer u Androidu koje se pišu u Java programskom jeziku, zajedno sa svojim bibliotekama pružaju komponente koje predstavljaju što se prikazuje na ekranu. Ti elementi se koriste od strane *Skia* programa koji upravlja grafičkom karticom te ovisno o primljenim komponentama govori grafičkoj kartici što da prikaže na ekranu. Obično, višeplatformske tehnologije iskorištavaju ovo već postojeću implementaciju nativnih tehnologija te jednostavno fasada preko njih koja komunicira s obje platforme. Flutter je drugačiji po tome što totalno zamjenjuje tu već postojeću funkcionalnost nativnih tehnologija sa svojom. Time se postiže bolja i brža komunicira sa *Skia* programom te ima bolju kontrolu. Tome što se prikazuje na korisničkom sučelju.

Diagram

Description automatically generated

Sl. 2.4. Komuniciranje sa GPU

Flutter kao rješenje za određivanje što se prikazuje na korisničkom sučelju, koristi kompozicijsku strukturu stabla, i to tri različite vrsta stabla. [5]

* Stablo widgeta („*Widget tree“*)
* Stablo elemenata („*Element tree“*)
* Stablo prikaza („*Render tree“*)

Stablo widgeta predstavlja upravu onu kompoziciju widgeta koju programer napiše unutar koda. Ono predstavlja deklarativan pristup izricanja kako treba izgledati korisničko sučelje ovisno o trenutnom stanju aplikacije. Postupak prikazivanja korisničkog sučelja, započinje pozivanjem „*build“* fazom, unutar koje se izvršavaju „*build“* metode pojedinih widgeta. Iako je widget sam po sebi nepromjenjiv („*immutable“*), tokom izvođenja „*build“* metode Flutter može ubaciti nove widgete u stablo, čija uloga može biti skrivena unutar nekih od widgeta koje Flutter pruža. Tokom izvođenja „*build“* faze, Flutter. Preslikava stablo widgeta u stablo elemenata u odnosu jedan na prema jedan. Stalbo elemenata predstavlja stvarni opis komponente korisničkog sučelja za pojedini widget zajedno sa njegovim stanjem, ako je „*StatefulWidget“*, povezuje stablo widgeta i stablo prikaza te upravlja stablima. Svaki element zna koji mu je pripadajući widget iz stabla widgeta, tako što zna koji je njegov tip te koji je njegov ključ, ako postoji. Ključ služi kao jedinstveni identifikator koji se može predati svakom widgetu, kako bi element unutar stabla elemenata mogao bolje odrediti sa kojim je widgetom povezan.

Chart, bubble chart

Description automatically generated

Sl. 2.5. Stablo widgeta i stablo elemenata,

izvor: https://docs.flutter.dev/resources/architectural-overview

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

* Element komponente („*ComponentElement“*)
* Element prikaza („*RenderObjectElement“*)

Element komponente služi kao spremink 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. Sastavljeno je na principu da svaki objekt zna samo tko su mu djeca unutar stabla te koja su njihova ograničenja s obzirom na veličinu. Koristeći taj princip, Flutter prolazi korz stablo koristeći „deth-first*“* iteraciju stabla. Prilikom posjete, svaki objekt preda dijeci 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 (*O(n)*). Nakon završetka prolaska kroz stablo, određene su dimenzije svakog objekta te su spremni za prikaz na korisničkom sučelju.

Chart, bubble chart

Description automatically generated

Sl. 2.5. Stabla Fluttera,

izvor: https://docs.flutter.dev/resources/architectural-overview

#### 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 sa kojim je widgetom povezan s obzirom na njegov tip i ključ. Ako se widget zamjeni sa widgetom istog tipa, i istog ključa (ako postoji ključ), element se ponovo iskorištava i povezuje sa novim widgetom. Jedina stavka koja se mijenja, su nove informacije iz novonastalog widgeta. To mogu biti informacije, na primjer, veličini teksta, boji itd. Time, element prosljeđuje potrebne informacije objektu prikaza unutar stabla prikaza, koji se također ponovo koristi. Postupak pozicioniranja widgeta na ekranu se ponavlja te se oni oslikavaju na korisničkom sučelju koristeći nove upute o prikazu na ekran.

### 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 i godinu [6]. Kao programski jezik koristi se Swift. Swift je programski jezik namijenjen za opću namjenu pri uzimanju modernog pristupa kod sigurnosti, performansi i metodologijama dobrog dizajna [7]. Swift je namjenjen da zamjeni *Objective-C* programski jezik te se koristi za razvoj svih aplikacija na Apple platformama.

#### Razlika sa *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 deklerativan pristup pri izradi aplikacije te podržava više platformi od strane Apple firme [8]. Unutar UIKit alata, programer stvara pojedine prikaze („views*“*) te ih povezuje kako bi stvorio hijerarhiju prikaza koje skupa čine korisničko sučelje. Prilikom promjena na ekranu, 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 funkcija trenutnog stanja aplikacije. Promjenom stanja aplikacije, promjena se reflektira na ekranu. Ovaj deklarativni pristup pri stvaranju prikaza na ekranu ne zahtijeva ručno stvaranje i brisanje prikaza na ekran od strane programera, što smanjuje kompleksnost i mogućnost grešaka u kodu. *SwiftUI* se oslanja na mogućnosti samog *Swift* programskog jezika, što zahtjeva promjenu jezika kako bi se unaprijedio. Ovo je vrlo bitna stavka kod odabira alatka u kojem će programer izraditi aplikaciju. Animacije su važan dio svake aplikacije, sa SwiftUI pruža lakše rukovanje sa animacijama te dosta animacija i prijelaza se dobije direktno od samog alat. Animacije u UIKit alatu nisu jednostavne za izvest te se moraju eksplicitno navesti kako se izvode. Kako bi dobio najbolje elemente SwiftUI alata, mora koristiti noviju verziju operacijskog sustava na uređaju za kojeg se aplikacija radi. Sa iOS 16 verzijom, SwiftUI je napredovao, ali potrebno je podržavati starije verzije operacijskog sustava kako bi se pokrili svi ciljani korisnici. SwiftUI je dostupan od iOS 13 verzije, 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, sa boljim verzijama SwiftUI alata, postaje bolja opcija pri odabiru alata za izradu korisničkih sučelja.

Najaviti tablicu

Tablica 2.1. Usporedba SwiftUI i UIKit alata

|  |  |  |
| --- | --- | --- |
|  | UIKit | SwiftUI |
| Pristup rada | Imperativan | Deklarativan |
| Minimalna iOS verzija | 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 widget-e | ne | da |
| Podržava Live Preview | ne | da |

#### Struktura SwiftUI aplikacije

Struktura SwiftUI aplikacije se iskazuje na deklarativan način, kao što se iskazuju i prikazi na ekranu. Početna točka svake aplikacije kreirati tip koji nasljeđuje „App*“* protokol te ga je potrebno naznačiti sa „@main*“* ključnom riječi koja pruža osnovnu implementaciju „*main“* funkcije. Ovaj protokol predstavlja strukturu i ponašanje aplikacije [9]. „App*“* protokol zahtijeva implementiranje „body*“* izračunatog atributa koji definira sadržaj aplikacije. Tijelo aplikacije je sastavljeno od scena te je svaka scena sastavljena od kompozicije različitih prikaza. Kako je svaki element struktura, a ne klasa, poboljšavaju se performanse pri izvođenju aplikacije. Strukture su unutar Swift programskom jezika „Value type*“* tipovi podataka, što znači da su tretirane kao cijeli broj 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*“* izračunatog atributa koji mora vratiti točno jedan tip prikaza [10]. Korisničko sučelje se tvori 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 se smanjuje broj 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.

Icon

Description automatically generated

Sl. 2.6. Struktura SwiftUI aplikacije

#### Prikazivanje korisničkog sučelja

SwiftUI, kao i u UIKit alatu, je apstrakcija iznad petlje događaja koja je odgovorna za slanja poruka kodu koji ja zaslužan za korisničko sučelje. Zauzvrat kod odgovoran za korisničko sučelje može odrediti koji se dijelovi ekrana trebaju ažurirati. Unutar SwiftUI alata petlja je skrivena te nije potrebno direktno komunicirati sa niti znati da ona postoji [11]. 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 Ma OS X 10.0. Petlja događaja je dio infrastrukture povezane sa nitima. Njen zadataka unutar te infrastrukture je da rezervira poslove koje se trebaju napraviti i organizira nadolazeće događaje. Ova petlja efektivno daje niti posao kada posla ima te ju stavlja u stanje spavanja kada nema posla za obradu [12]. 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 ekranu.
* Tip 1. Događaj sinkroniziranje ažuriranja ekrana i ponovnog prikazivanja na ekranu te neki asinkroni događaji za. Dohvaćanje podataka sa interneta.
* Mjerači vremena.
* Glavni red za obrađivanje. Bilo kakav kod asociran sa ovim redom.

Osim navedenih ulaznih događaja, moguće je dodati slušatelje na instancu „*CFRunLoop“* koji budu obaviješteni kada petlja dosegne određeni dio [13]. Animacije se vrte na glavnoj niti, ali ako glavna nit zablokira, moguće je da će animacije i dalje raditi. Razlog tome je što se animacije, iako vrte na glavnoj niti, ne vrte na istom procesu. Aplikacija pomoću biblioteke „*Core Animation“* prikazuje svoje animacije na i prikaze na ekran, ali ih prikazuje na više različitih slojeva koji se nazivaju „*CALayer“*. Pomoću kompozitora aplikacija spaja više različitih slojeva te ih prikazuje na ekran, dok „*Core Animation“* komunicira sa serverom za crtanje prikaza na ekran kako bi mu dao informaciju o tome što treba nacrtati i animirati.

#### Obnavljanje stanja na ekranu

Kada se neko stanje unutar aplikacije promjeni, potrebno je reflektirati tu promjenu na ekranu tako što se mjenjaju pojedini dijelovi prikaza kao što su boja ili tekst. Nakon napravljenih promjena na pojedinim slojevima, potrebno je komunicirati tu promjenu serveru za crtanje te prikazati ih. Aplikacija ne može znati kada su sve promjene gotove i spremne za crtanje na ekran. Za signaliziranje završetka promjena koristi se transakcija, instanca „*CATransaction“*. Transakcija se mora pokrenuti kada se promjene dese i završiti kada su promjene gotove. Pomoću ovog mehanizma aplikacija zna da su sve promjene gotove za trenutni sloj te se pomoću servera za crtanje prikazuju na ekran [14]. Transakcije se mogu manualno stvoriti i završiti te su osnova unutar svake Apple, pa tako i SwiftUI aplikacije. Tokom izrade aplikacija, programer ne komunicira direktno sa 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.

Text, chat or text message

Description automatically generated

Sl. 2.6. SwiftUI render loop, izvor: https://rensbr.eu/blog/swiftui-render-loop/#:~:text=Just%20like%20UIKit%20%2C%20SwiftUI%20is,render%20loop%20of%20an%20application.

## Implementacija u Flutteru

### Arhitektura Flutter aplikacije

Arhitektura unutar višeplatformske aplikacije u Flutteru je podijeljena na dijelove:

Graphical user interface, text, application

Description automatically generated

Sl. 4.1. 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, dependency injection, ulazni widget unutar aplikacije i slično. Sama main funkcija, koja predstavlja ulaznu točku unutar aplikacije se nalazu unutar main.dart datoteke. Ona se nalazi na vrhu stabla datoteka i direktorija radi bolje preglednosti. Ostatak aplikacije je podijeljen na tri glavna sloja.

* Data layer
* Domain layer
* Ui layer

Data sloj je vanjski sloj aplikacije. On sadržava implementacije servisa/repozitorija koji komuniciraju sa nekim vanjskim izvorima podataka (serverima), lokalnim izvorima (lokalna baza podataka), servise za upravljanje hardverom te bilo kakve servise ili menadžere koji nemaju povezanosti sa biznis logikom same aplikacije te nisu vezani za aplikaciju. U tom smislu, mogli bi se izdvojiti posebno unutar posebne biblioteke te koristiti također unutar drugih aplikacija. Ovom sloju pripada ‘source\_remote’ direktorij. U ovoj aplikaciji nema komunikacija sa lokalnom bazom podataka ili hardverom uređaja. U slučaju da je to potrebno, takve servisi bi bili stavljeni unutar posebnih ‘source\_local’ i ‘device\_hardware’ direktorija.

Graphical user interface, text, application

Description automatically generated

Sl. 4.2. Data sloj aplikacije

Domain sloj je srednji sloj aplikacije te sadrži apstrakciju servisa/repozitorija i menadžera koje aplikacije treba koristiti za potrebe izvođenja biznis logike unutar aplikacije. Ove apstrakcije se odnose na implementacije iz data layer sloja. Time se servisi iz vanjskog sloja aplikacije mogu lako zamijeniti drugim servisom, ako je to potrebno. Pri tome nije potrebno mijenjati ništa što direktno utječe na biznis logiku. Domain layer sloj također sadrži apstrakciju i konkretnu implementaciju upravljača (interactors). Svrha upravljača je koordiniranje obavljanja nekog zadatka, kojeg zahtijeva biznis logika aplikacije. 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, 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 služe za lak pristup podacima koji su raspoloživi svim repozitorijima.

Graphical user interface, application

Description automatically generated with medium confidence

Sl. 4.3. Domain sloj aplikacije

UI sloj sadržava sve stranice koje aplikacija treba prikazati, pri čemu je svaka stranica idealno razbijena na više manjih widgeta. 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, sama konfiguracija teme aplikacije na osnovu danih podatka i slično. Osim samog izgleda apliakcije, ovaj sloj sadrži i 'state management' aplikacije. State management se odnosi na upravljanje stanjima aplikacije te predstavlja dio koji povezuje domain sloj sa UI slojem. Trenutni izgled aplikacije je funkcija trenutnog stanja aplikacije, time se upravljanjem stanja aplikacije mijenja korisničko sučelje i određuje što će se prikazati na ekranu. 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 mogu reagirati na razne događaje sa korisničkog sučelja, kao što su pritisak na gumb, listanje kroz listu na ekranu ili bilo kakvu akciju od strane korisnika. Time, prezenteri su jedina komponenta koja komunicira sa komponentama korisničkog sučelja. Kao izlaz daju nekakvo stanje koje opisuje to sučelje, a kao ulaz reagiraju na razne događaje.

Text

Description automatically generated

Sl. 4.4. UI sloj aplikacije

Common direktorij na samom vrhu stabla, sadrži stvari koje koriste više od jedan sloj apliakcije. Unutar njega se nalaze stvari kao što su errori unutar aplikacije, različita okruženja apliakcije ( na primjer produkcija i razvojno okruženje), modeli podataka koji se koriste na više slojeva aplikacije, pomoćne stvari i slično. Također svaki sloj može sadržavti 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 apliakcije s emogu nalazi unuat posebnih slojeva. Time, model podatka koji predstavlja odgovor pri komunikaciji sa serverom bi se nalazi unutar 'data' sloja, u 'domain' sloju poseban model podatka koji se dobiva iz modela dobivenog sa odgovora sa servera te poseban model podatka na korisničkom sloju, s obzirom koji podaci su potrebni tom sloju.

Graphical user interface, text

Description automatically generated

Sl. 4.4. Common direktoriji

### Upravljanje stanjima unutar Flutter aplikacije

Komunikacija unutar aplikacije je prikazana na slici ispod.

Diagram

Description automatically generated

Sl. 4.5. Upravljanje stanjima unutar Flutter aplikacije

Tok se izvodi tako da widget (korisničko sučelje) obavijesti prezentera o nekom događaju, na primjer da dohvati listu slika. Prezenter potom poziva određeni interactor koji je zadužen za koordiniranje obavljanja tog zadatka. Upravljač (interactor) poziva repozitorije odgovorne za određene resurse, potreban za izvršavanje tog zadatka. Potom repozitorij poziva servis za komuniciranje sa serverom za slike. Čeka se na odgovor od servera. Nakon što odgovor dođe, 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 samih prezentera koristila se biblioteka 'Riverpod', koja predstavlja jedno od mogućih rješenja upravljanja stanjima unutar aplikacije.

### Beskonačna lista slika u Flutteru

Implementacija beskonačne liste slike se izvela pomoću 'infinite\_scroll\_pagination' biblioteke, koja je proizašla iz Flutter zajednice te je dostupna svima za korištenje. Ova biblioteka omogućuje definiranje izgleda korisničkog sučelja liste kao što je jedan element unutar liste, čekanje na dodatne elemente koje se trebaju nadodati na listu, greška pri dohvaćanju novih elemenata liste. Za prikaz elementa liste koristi se widget 'ImageListItem', za učitavanja se prikazuje indikator učitavanja podataka, dok se za stanje greške prikazuje 'ListeError' widget koji prikazuje korisniku grešku na ekranu.

Text, chat or text message

Description automatically generated

Sl. 4.5. Korisničko sučelje liste

Osim vizualnih reprezentacija, biblioteka pruža način slušanja na obavijesti o tome kada je potrebno dohvatiti nove podatke za prikaz u listi. Za slušanje na dogašaje 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 pozivamo 'fetchImages' metodu nad prezenterom sa datim 'pageKey' parametrom.

Text

Description automatically generated

Sl. 4.6. Zahtjev za dohvaćanjem dodatnih slika

Prezenter zatim poziva upravljača (interactora) da obavi radnju dohvaćanja potrebnih slika. Potrebno je poslati broj stranice sa koje želimo dohvatiti slike. Prezenter nasljeđuje klasu 'RequestProvider' koja služi kao fasada oko izvođenja asinkronih zahtijeva te vraća određeno stanje ovisno kako je se zahtijev izvršio. To stanje moće biti uspiješno i vratiti neke podatke (u ovom slučaju listu modela 'UnsplashImage'), u početnom stanju, neuspješno ili u procesu učitavanja. Zahtjev koji se ćeli izvesti se izvršava pomoću metode 'executeRequest' koja kao parametar prima asinkronu funkciju koju želimo izvršiti.

Text

Description automatically generated

Sl. 4.7. Prezenter za listu slika

Kako bi se lista nadopunila sa 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.

Text

Description automatically generated

Sl. 4.8. Slušanje na nova stanja prezentera

### Filtriranje i spremanje slike u Flutteru

Slika se prikazuje pomoću 'Image' widgeta. Filter se primjenjuje tako da se slici kao roditelj postavi 'ColorFiltered' widget. Pomoću 'ColorFiltered' widgeta se primenjuje filter koji izoštrava sliku.

Text

Description automatically generated

Sl. 4.8. Filtriranje slike

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. Za dovaćanje filtrirane slike, kao roditelj slike se postavlja 'RepaintBoundary' widget, pomoću kojeg se sva dijeca od 'RepaintBoudnary' widgeta stavljaju na svoj sloj za bojanje na ekran te time promjene na dugim mjestima korisničkog sučelja nemaju utjecaja. Dodjeljuje se ključ 'RepaintBoundary' widgetu te se pomoću ključa dobija referenca na željeni widget. Nakon toga dohvaća se slika sloja kojeg 'RepaintBoundary' widget obuhvaća te slika pretvara u niz bajtova koji reprezentiraju tu sliku. Pomoću bajtova slike, stvara se PNG datoteka unutar direktorija dokumenata za aplikacije te se time slika sprema na uređaj. Zatim, slika se sprema u galeriju na uređaj korisnika, referencirajući se na prethodno spremljenu PNG datoteku. Spremanje u galeriju se obavlja pomoću 'ImageGallerySaver' klase, koja proizlazi iz vanjske biblioteke.

Text

Description automatically generated

Sl. 4.8. Spremanje filtrirane slike

## Implementacija u SwiftUI

### Arhitektura SwiftUI aplikacije

Arhitektura korištena u SwiftUI aplikacije je identična aplikaciji napravljenoj u Flutter-u. Aplikacije se sastoji također od tri prijašnje spomenuta sloja te struktura direktorija je slična Flutter aplikaciji.

Graphical user interface, application

Description automatically generated

Sl. 4.12. Arhitektura SwiftUI aplikacije

App direktoriji sadržava stvari vezane za aplikaciju, kao greške unutar aplikacije, dependency injection i slično. Ulazna točna unuar aplikacije je 'App' struktura, koja se nalazi unutar 'picmore\_iosApp' direktorija. Aplikacije se dijeli na prethodno spomenuta tri sloja (dana, domain ui). Dana sloju pripada direktoriji 'remote\_services', koji sadrži implementaciju repozitorija zaslužnog za upravljanje slikama.

Graphical user interface, text, application

Description automatically generated

Sl. 4.12. Data sloj iOS aplikacije

Domain sloju pripada 'domain' direktorij, unutar kojeg se nalazi apstrakcija repozitorija, apstrakcija i implementacija upravljača (interactor) te implementacija klase za računanje Fibonacci broja.

Graphical user interface, text

Description automatically generated

Sl. 4.13. Domain sloj iOS aplikacije

UI sloju aplikacija pripada 'ui' direktorij. Unutar njega se nalze implementacija zaslona liste slike te detalja slike, zajendo sa sučeljem za računanje broja. Osim korisničkog sučelja, nalaze se klase koje upravljaju stanjima korisničkog sučelja, nazvanih 'View Models'. Oni imaju istu ulogu kao što prezenter ima unutar Flutter aplikacije. Jedina razlika imeđu njih je njihov naziv, koji se i ne mora razlikovati, te sam način upravljanja stanja. 'View Model' je zaslužan za reagiranje na događaje sa korisničkog sučelja te izmjenu stanja na korisničkom sučelju.

Graphical user interface, application

Description automatically generated

Sl. 4.13. UI sloj iOS aplikacije

### Upravljanje stanjima unutar SwiftUI aplikacije

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

Diagram

Description automatically generated

Sl. 4.14. Upravljanje stanjima unutar iOS aplikacije

Korisično sučelje obavijti View Model o nekom događaju, kao na primjer klik na gumb ili dolazak na ekran. 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 servise koje odrađuju posao ili menadžere koji su upravljaju za nekim dijelom hardvera.

### Beskonačna lista slika u SwiftUI

'InfiniteList' je prikaz koji je zaslužan za prikazivanje beskonačne liste slike. Kao atribute koji mijenjaju stanje ovog prikaza, ovaj prikaz ima 'data' atribut, koji predstavlja podatke koji se prikazuju unuar liste. U ovom slučaju, to je lista modela 'UnsplashImage'. Atribut 'isLoading', čijim mijenjanjem se prikazuje stanje učitavanja novih slika u listi. 'isError' atribut, čijim mijenjanjem se prikazuje greška 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. Sadrži 'content' atribut, koji 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.

Text

Description automatically generated

Sl. 4.15. Atributi prikaza beskonačne liste

Kao i svaki prikaz koji se koristi unutar SwiftUI, potrebno je da da prikaz nasljeđuje protokol 'View' te u 'body' atributu definira izgled svoga prikaza. 'InfiniteList' prikaz implementira body atribut na način 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.

Text

Description automatically generated

Sl. 4.16. Implementiranje prikaza beskonačne liste iOS aplikacije

'ImagesPage' prikaz je glavni prikaz na stranici gdje se prikazuje lista slika, u sebi koristi 'InfiniteList' prikaz te ga konfigurira sa View Model-om koji je odgovoran za upravljanje ovim dijelom korisničkog sučelja.

Text

Description automatically generated

Sl. 4.17. Implementiranje prikaza beskonačne liste

Prilikom zahtijeva novih slika za prikaz u listi, 'InfiniteList' prikaz će pozvati 'loadMore' metodu nad View Model-om. Time View Model mjenja stanje u stanje učitavanja novih slike te poziva upravljač (interactor) da obavi dohvaćanje novih slika sa servera. Nakon što upravljač dohvati nove slike, View Model promjeni stanje sa ovim slikama te se to reflektira na korisničkom sučelju. Kod mjenjanja stanja, unuar SwiftUI prikaz može promatrati objekt koji nasljeđuje 'ObservableObject' protokol te taj objekt objavljuje atribute na koje želi da slušatelji ponovo procjene trenutno stanje, tako da dodaju ključnu riječ '.@Published' ispred atributa. 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.

Text

Description automatically generated

Sl. 4.18. Implementacija View Model-a beskonačne liste

### Filtriranje i spremanje slike u SwiftUI

Pomoću 'Slider' prikaza, danog od SwiftUI biblioteke, mjenja se vrijednost 'filterIntensity' koja diktira jačinu filtera. Prilikom svake promjene poziva se metoda 'applyProcessing' koja primjenjuje filter izoštrenja i sprema reuzultat unutar 'processedImage' variable. Varijabla 'processedImage' se pomoću 'Image' prikaza, danog iz SwiftUI biblioteke, prikazuje na korisničkom sučelju, prikazano na slici 4.19.

Text

Description automatically generated

Sl. 4.19. Implementacija prikaza filtriranja u SwiftUI

Metoda 'applyProcessing' implementira filter na način da mjenja pojedine elemente na dijagonali matrice, koja se potom množi sa slikom. Implementacija filtera je vidljiva na slici 4.20.

Text

Description automatically generated

Sl. 4.20. Implementacija filtriranja slike u SwiftUI

Korisnik pritiskom na gumb sa tekstom 'Save image to gallery', sa slike 4.19, sprema filtriranu sliku u galeriju slika na uređaju. Klasa 'ImageSaver' enkapsulira postupak spremanja slike u galeriju te je vidljiva na slici 4.21. Koristeći metodu 'UIImageWriteTosavedPhotosAlbum' iz 'UIKit' biblioteke, sprema se filtrirana slika u galeriju. Prilikom završetka spremanja slike, izvršava se metoda 'saveComplete' koja je poziva funkcije za obavjest o uspješnom ili neuspješnom spremanju slike.

Text

Description automatically generated

Sl. 4.21. Implementacija filtriranja slike u SwiftUI

# TESTIRANJE I ANALIZA PROGRAMSKOG RIJEŠENJA

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 sa rezultatima testiranja iz drugih radova za slučajeve gdje su ti podaci dostupni.

## Testiranje vremena odziva

## Analiza rezultata

## Analiza implementacije i korisničkog iskustva

Beyond web native hybrid…

Approaches to Mobile Application Development: Comparative Performance Analysis, [01]

* A comparative study of the processing time of software applications for mobile devices, generated using different development approaches, was presented.
* To assess processing speed, a simple calculation including several iterations, mathematical operations and floating point arithmetic was proposed, which is summarized in the following series:

**A Comprehensive Comparison Between Hybrid and Native App Paradigms** , [02]

* The cross-platform feature of hybrid app is the biggest advantage to native app as it can save much time and cost.
* So from the aspect of compatibility, hybrid app is more easily programmed than native app.
* From users’ viewpoint, they care more about the experience rather than the developing process of an app.
* Messured CPU Occupancy Ratio (%) native request so it costs more CPU resources (106% higher),
* Mermory Occupancy (MB) , memory space (73.0% higher)

Conclusion: Both hybrid and native paradigms are ways to fulfill the different needs and preferences of developers and users, but none of then can be thought as a perfect solution. So the recommendation is, which to use depends on the actual demand, in case of UE-oriented or performance is emphasized more, native app paradigm would be a better choice, and if code reusability and cross-platform feature is necessary hybrid app paradigm would be optimal.

**A Performance Evaluation of Cross-Platform Mobile Application Development Approaches** , [03]

* Each of the benchmark apps consists of two screens: the initial screen is a configuration screen for setting the parameters of the benchmark app. The second screen is the main screen which displays a screenful of contents of variable sizes. The size of the contents is controlled by one or more parameters of the benchmark app.
* *Observations on memory usage*. The memory usage of all apps generally increases proportionally to the size of the UI,
* *Observations on app size*. The sizes of the native apps, NA and NI, are the smallest on both platforms. The sizes of cross-platform apps are significantly larger than the sizes of the native apps on both Android and iOS platforms.

Conclusion: Our experiment shows that there are significant differences in the performance characteristic of the benchmark apps developed using different approaches. The data collected from the experiment offers insights to the designs and trade-offs of the different approaches. These insights offer guidance to the practitioners when selecting among the different approaches.

**Comparing performance parameters of mobile app development strategies** , [04]

* In iOS, the Time Profiler feature of the Instruments tool is used for all response time measurements.
* Response times under 100 milliseconds will feel instantaneous to the user. Response times up to one (or a few) second(s) are acceptable to the user, if they rarely occur. Delays greater than a few seconds will significantly degrade the user experience.
* *Launch time : HE:191 LE:611*
* Overall the native implementations have a consistently low memory requirement.

Displaying CPU usage in tables as numerical values is not a trivial operation. In order to draw conclusions regarding the CPU use of specific actions in the mobile application, the sample time for the measurement intervals must be suf- ficiently small. --as the iOS and Windows Phone 8 measurement tools do not allow sufficiently small sample intervals.

* The advantage of using an existing application instead of making a new one is twofold. First of all, learning tools and implementing applications is very time consuming.
* In Table 1, an overview of the devices used for the per- formance analysis is given. For Android and iOS both a high-end and a low-end device was selected.
* Generally speaking, response times are an important fac- tor regarding the user experience. This study measures the response times for three different actions: The launch time of the application, the time to load a new page of the ap- plication and the time to return to the previous page.
* The CPU usage is the percentage of the total CPU ca- pacity of the device used by the application in the measured time interval. CPU intensive applications may negatively impact other processes running on the device, decreasing user experience. This research considers the overall CPU usage of an application, as well as the CPU usage during two specific actions in the PropertyCross application: launching the application and using the GPS to search and list prop- erties.
* The memory usage parameter reflects the amount of RAM memory allocated by the application. Measuring the mem- ory footprint of the application is especially important for low-end devices as the performance of the platform can be significantly degraded if a high percentage of the available RAM is allocated.
* The first measure- ment is the amount of RAM memory used when the app is fully launched. After launching the application, each page of the application was visited in a specified order. The second value is the maximal memory usage measured while going through the application. The final value is the memory us- age measured after each page of the application was visited.
* make intensive use of hardware resources such as wireless communication and sensors will have a higher energy con- sumption. Battery usage is, hence, not considered in this paper.

adv:

• Existing infrastructure. Developing new applica- tions from scratch is time consuming. In many cases, existing code for other projects can be reused. For instance, existing GWT projects are easily converted

Platform-specific code. Some cross-platform tools (e.g. JavaScript frameworks) use a 100% shared code base for all targeted platforms. Therefore, not all na- tive features can be accessed. These tools are, hence, not suitable for the development of applications with very specific requirements regarding user interfaces and hardware access.

conclusion:

* The performance penalty resulting from the use of cross-platform tools is generally acceptable, especially on high-end devices.
* In order to select a suitable cross-platform tool, several criteria other than performance have to be considered. Some of these parameters are often discussed (e.g. license cost, development environment/support, maturity and stability)

Beyond Web/Native/Hybrid: A New Taxonomy for Mobile App Development [05]

Code Sharing. The amount of code sharing is at its low- est in Endemic Apps. A major point of using pandemic or ecdemic approaches is to share code. Code sharing is also possible for En- demic Apps, however. You can, for example, share test code with a cross-platform UI testing tool such as Appium12 or by generating parts of the code from models.

In practice, code sharing in pandemic apps tends to be more common than in ecdemic apps. The reason for this is that business logic and UI are typically written with shared code in pandemic apps, while ecdemic frameworks often offer to mix in more en- demic code (especially for the UI). For apps of lesser complexity, however, code sharing could be nearly 100 % for both approaches.

* User Experience. From the perspective of UX, Endemic Apps offer the best performance, full native UI and all of the pos- sibilities of the OS. Pandemic apps (with the exception of Hybrid Bridged Apps), on the other hand, lack the possibility of using a native UI which may reduce UX. Performance may also be a prob- lem for Web Apps and Hybrid Apps while System Language Apps may in some cases even outperform Endemic Apps.

# ZAKLJUČAK

# LITERATURA

[1] Flutter, https://flutter.dev/

[2] Dart, https://dart.dev/overview#platform

[3] Flutter docs, arhitectular overview, https://docs.flutter.dev/resources/architectural-overview

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

[5] Flutter docs, inside Flutter, https://docs.flutter.dev/resources/inside-flutter

[6] Introducing SwiftUI, https://developer.apple.com/tutorials/swiftui

[7] About Swift, https://www.swift.org/about/

[8] What is SwiftUI, https://cocoacasts.com/swiftui-fundamentals-what-is-swiftui#:~:text=SwiftUI%20is%20Apple's%20brand%20new,is%20a%20cross%2Dplatform%20framework.

[9] App Structure and Behavior, https://developer.apple.com/documentation/swiftui/app-structure-and-behavior

[10] SwiftUI View, https://developer.apple.com/documentation/swiftui/view/

[11] Apple documentation, Run Loop, https://developer.apple.com/documentation/foundation/runloop

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

[13]. Run Loops, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html

[14] Animations in SwiftUI: Get to Know Transactions, https://betterprogramming.pub/animation-in-swiftui-get-to-know-transactions-7cd57cfb299f

# SAŽETAK

# ABSTRACT

# ŽIVOTOPIS

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Potpis autora

# POPIS PRILOGA