Razvoj mobilnih aplikacija pomoću Fluttera

Turić, Danijel

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Danijel Turić

Razvoj mobilnih aplikacija pomoću Fluttera

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Danijel Turić

Matični broj: 0016118101

Studij: Informacijsko i programsko inženjerstvo

Razvoj mobilnih aplikacija pomoću Fluttera

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Zlatko Stapić

Varaždin, rujan 2020.

Danijel Turić

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema ovoga rada je razvoj prirodnih mobilnih aplikacija i aplikacija napisanih pomoću razvojnog okvira Flutter. Pojavom Fluttera omogućuje se izrada mobilnih aplikacija temeljenih na jednom izvornom kôdu koja se izvodi na više platformi. Postavlja se pitanje da li je Flutter dovoljno dobra zamjena prirodne aplikacije i koliko je zahtjevan proces izrade takvih aplikacija. Unutar rada ukratko je opisana svaka od trenutno dostupnih tehnologija koje se koriste za izradu mobilnih aplikacija te je napravljena usporedba tehnologija po elementima arhitekture platforme i životnog ciklusa glavnih elemenata koji se koriste za izradu aplikacija. Za praktični dio rada izrađene su tri mobilne aplikacije, jedna za svaku navedenu platformu. Za izradu pojedine mobilne aplikacije koristi se jednostavna MVC arhitektura koja je prilagođena svakoj od platformi za koju je aplikacija izrađena. Za usporedbu, uspoređuju se elementi izrade aplikacija i aplikacije kao finalnog proizvoda.

Ključne riječi: Flutter, Android, iOS, Mobilne aplikacije, Mobilne platforme, Arhitekture sustava

Sadržaj

1. Uvod

U današnje doba kada su pametni uređaji postali navika i svakodnevni predmet bez kojega je teško zamisliti današnji život, pojavljuje se prostor za izradu aplikacija koje pokrivaju razne potrebe korisnika pametnih uređaja. Glavni problem koji se pojavljuje je proces izrada aplikacija i odabir dostupnih tehnologija za samu izradu.

Kroz kratku povijest razvoja mobilnih aplikacija dogodilo se da su i velikani pali kao što je Windows Phone (Tung, 2019) gdje je jedan od razloga pada platforme bio taj što nije bio dovoljan broj aplikacija koje bi korisnici koristili na svojim uređajima.

Danas su na tržištu mobilnih platformi dostupne dvije mobilne platforme. Mobilne platforme su Android i iOS, radi toga postavlja se pitanje načina izrade mobilnih aplikacija. Jedno od rješenja je izrada prirodnih aplikacije za svaku od platformi zasebno, prilikom čega je potrebno pisati dva različita kôda. Drugo rješenje je izrada aplikacija koja dijeli isti kôd, te se omogućava izrada aplikacije koja se istovremeno može izvršavati istovremeno na obje platforme. Glavna prednost takvih aplikacija je ta da se smanjuje vrijeme izrade aplikacija i smanjuje se broj potrebnih resursa za izradu aplikacije, dok se smanjuje kvaliteta aplikacija i korisničko iskustvo korištenja aplikacije u odnosu na prirodne aplikacije.

Prilikom izrade prirodnih aplikacija za dostupne mobilne aplikacije koriste se službeni paketi za razvoj programa (eng. *Software Development Kit, SDK*) i razvojna okruženja koja omogućuju jednostavniju i bržu izradu aplikacija. Uz navedene alate postoji pitanje jeli je moguće izraditi jednu aplikaciju koja će se izvršavati na više platformi kako bi se još više pojednostavio i ubrzao proces izrade mobilnih aplikacija. Kao odgovor na to pitanje, pojavile su se alternative koje omogućuju izradu mobilnih aplikacija za više platformi izrađenih jednim kôdom. Te alternative su JavaScript, Xamarin i Flutter. JavaScript i Xamarin su opcije koje se ponašaju kao omotači (*eng. Wrapper*) oko službenog SDKa, što u naposljetku utječe na performanse odnosu na prirodne mobilne aplikacije. Flutter, za razliku od prethodno navedenih rješenja se ne ponaša kao omotač oko prirodnog SDKa, već koristi vlastitu implementaciju stroja za izvođenje aplikacija (*eng. Engine*) koji je direktno se direktno povezuje na resurse platforme, a ne na metode koje pruža SDK.

Pojavom Fluttera kao rješenja za izradu više-platformskih aplikacija, postavlja se pitanje da li je Flutter dovoljno dobro rješenje kako bi se ubrzao i smanjio trošak razvoja mobilnih aplikacija, a da se pri tome ne izgubi na kvaliteti izrađenih aplikacija. U nastavku slijedi upoznavanje sa prirodnim platformama i sa Flutter razvojnim okvirom (eng. Framework) kako bi usporedba bila jednostavnija.

2. Metode i tehnike rada

Kroz rad će se govoriti o mobilnim platformama Android i iOS, te o razvojnom okviru Flutter. Prvo će se proučiti arhitekture, točnije na slojeve od kojih se građene platforme, odnosno razvojni okvir. Uz arhitekture će se također proučiti životni ciklusi glavnih elemenata koji se koriste tijekom izvođenja aplikacija. Nakon upoznavanja s platformama i razvojnim okvirom, proučit će se MVC arhitektura, kako bi se upoznalo sa samim procesom izrade mobilnih aplikacija i kako bi bilo lakše usporediti proces izrade aplikacija. Nakon upoznavanja s arhitekturom, na osnovu prethodno pojašnjenih pojmova napravit će se priprema kojom će se izraditi prirodne mobilne aplikacije i Flutter aplikacija. Nakon izrade aplikacija, usporedit će se proces izrade aplikacija te aplikacija kao finalni proizvod.

Prilikom izrade ovog rada će se koristiti programski alati kao što su programska razvojna okruženja, emulatori i simulatori koji se koriste za izvođenje mobilnih aplikacija. Za izradu Android prirodne aplikacije bit će korišten Android studio razvojno okruženje, programski jezik izrade biti će Kotlin, dok će se aplikacije izvršavati na Android emulatoru. Prirodna iOS aplikacija će bit izrađena unutar xCode razvojnog okruženja, programski jezik koji će se koristiti za izradu aplikacije je Swift i aplikacija će se izvoditi na simulatoru. Flutter aplikacija će se izrađivati unutar Android studio, prilikom čega je potrebno koristiti dodatak koji će omogućiti izradu Flutter aplikacije unutar razvojnog okruženja, za izradu aplikacije koristit će se Dart programski jezik, dok će se aplikacija izvršavati uz pomoć emulatora za Android i simulatora za iOS.

U nastavku će se ukratko napisati ponešto o svakoj od platformi koje će se dalje, detaljnije opisivati unutar ovoga rada, tako da će unutar ovoga poglavlja biti opisani elementi pojedine platforme kao što su početci svake platforme, struktura arhitekture platforme i životni ciklusi glavnih elemenata platforme. Uz platforme bit će obrađen i sam Flutter razvojni okvir koji omogućuje izradu jedne aplikacije za obje platforme.

3. Arhitekture platformi

Kako su prethodno navedene prirodne mobilne platforme i Flutter kao potencijalno rješenje za razvoj više-platformskih mobilnih aplikacija, u nastavku ovoga poglavlja bit će opisani slojevi arhitekture prirodnih mobilnih platformi i Fluttera kako bi se lakše upoznalo s načinom izvođenja aplikacija. Uz strukturu arhitektura opisat će se i životni ciklusi glavnih elemenata od koji se sastoji svaka aplikacija napisana za određenu platformu.

Svaka platforma se sastoji od više modula, koji su raspoređeni unutar slojeva kako bi se smanjila nepotrebna međusobna kohezija između samih modula. Svaki od slojeva pruža određeni skup funkcionalnosti koji se nude programeru (*eng. Developer*) aplikacija.

3.1. Android

Jedna od dvije mobilne platforme koje imaju veliki udio na tržištu mobilnih platformi je mobilna platforma Android. Platforma je trenutno najrasprostranjenija platforma što se tiče pametnih uređaja. Prema zadnjim podacima (*Mobile Operating System Market Share Worldwide | StatCounter Global Stats*, 2020) udio pametnih telefona koji koriste Android kao operativni sustav iznosi 74.4%.

Prema podacima (*Android (operacijski sustav) – Wikipedija*, 2020) Android kao platforma počela se razvijati 2003 godine pod vlasništvu Android Inc. U početku operativni sustav je bio namijenjen za digitalne kamere ali je to tržište bilo malo te su se odlučili promijeniti plan i razvoj platforme nastaviti u smjeru izvršavanja na pametnim telefonima. Prva verzija koja je službeno predstavljena je bila Android 1.0 2007. godine koja je dolazila na HTC T-Mobile G1 mobilnom uređaju.

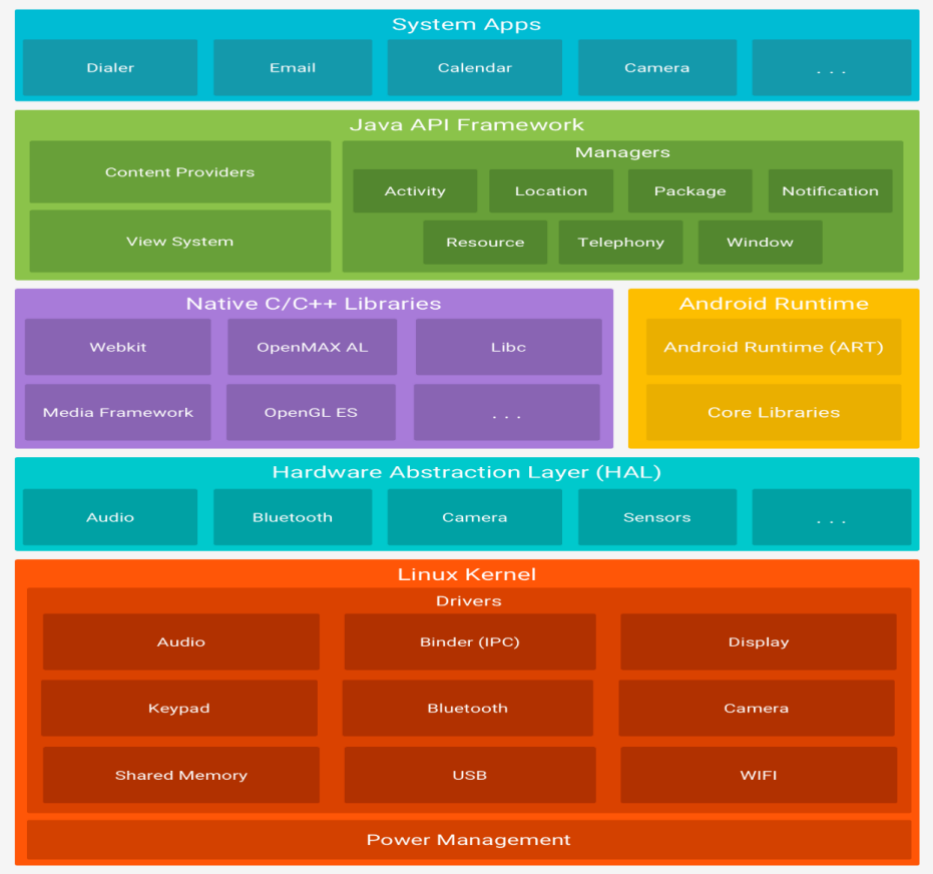
Mobilna platforma Android je platforma čiji je kôd otvorenog tipa (*Platform Architecture | Android Developers*, 2020), što znači da je omogućen pristup do samog programskog kôda, te je omogućeno da ga svatko može slobodno pregledavati i koristiti. Samim tim lakše je proučavati slojeve od kojih se sastoji arhitektura platforme, te način raspoređivanja i iskorištavanja dostupnih resursa za izvođenje mobilne aplikacije.

3.1.1. Slojevitost arhitekture

Android platforma kao i svaka druga platforma podijeljena unutar slojeva koji u konačnici grade arhitekturu platforme. Na slici 1 prikazana je arhitektura mobilne platforme Android iz koje se vidi da je arhitektura podijeljena unutar šest slojeva. Glavni i najniži sloj arhitekture platforme je jezgra platforme koja je bazirana na Linux jezgri (*eng Linux kernel*).

Nad slojem jezgre nalazi se sloj koji pruža pristup metodama koje se koriste za pristup sklopovlju uređaja na kojemu se izvodi Android platforma. Nad slojem sklopovske podrške nalaze se sloj koji sadrži biblioteke koje su napisane u C/C++ programskome jeziku (*eng. Native C/C++ Libraries*) i sloj radnog okruženja uz pomoć kojega se izvodi Android aplikacija (eng. Android Runtime). Nad tim slojevima se nalazi sloj koji omogućuje pisanje Android aplikacija u Java programskome kôdu i zadnji sloj je unutar kojeg se nalaze sistemske aplikacije. Prema prethodno opisanome, slojevi mobilne platforme Android su sljedeći:

- Linux Kernel kao najosnovniji sloj arhitekture platforme,
- HAL (Hardware Abstraction Layer) sloj koji pruža metode, koje se koriste za povezivanje s fizičkim sklopovljem uređaja,
- Native C/C++ Libraries sadrži biblioteke koje su napisane u C/C++ programskome jeziku,
- Android runtime sloj koji omogućuje izvršavanje prirodnih Android aplikacija,
- Java API Framework sloj koji omogućuje izvršavanje Java programskog kôda nad ostatkom arhitekture koja je bazirana na C/C++ programskome jeziku, te
- System Apps je sloj koji se sastoji od najosnovnijih aplikacija za rad svakog uređaja na kojemu se nalazi mobilna platforma Android.



Slika 1 Arhitektura Android platforme (izvor: https://developer.android.com/guide/platform, 2020.)

3.1.1.1. Linux kernel

Kao i svaka platforma, pa tako mobilna platforma Android sastoji od jezgre čiji je zadatak raspodjela dostupnih resursa između procesa koji se izvršavaju na platformi, te je temelj nad kojima se grade ostali slojevi arhitekture.

Jezgra mobilne platforme Android bazirana je na Linux jezgri. Funkcije sloja su upravljanje memorijom, upravljanje sustavom izvođenjem dretvi i upravljanje potrošnjom energije. Unutar sloja također se nalaze kontroleri (eng. Driver) za module kao što su Bluetooth, kamera, USB, WiFi, tipkovnica, Audio itd.

3.1.1.2. Hardware Abstraction Layer (HAL)

Sloj sklopovske apstrakcije (*eng. Hardware Abstraction Layer*) je sloj unutar arhitekture mobilne platforme Android unutar kojeg su sadržane metode koje koriste razni proizvođači komponenata kako bi jednostavnije mogli povezati sklopovlje s ostatkom platforme. Dakle sloj sklopovske apstrakcije je sloj koji je omogućuje direktnu komunikaciju platforme s komponentama fizičkog sklopovlja koje je dostupno na uređaju na kojemu se izvršava mobilna platforma Android. Apstrakcijom metoda omogućuje se izmjena komponenata, a da pri tome nije potrebno mijenjati strukturi ostalih slojeva arhitekture.

3.1.1.3. Android Runtime

Sloj izvršavanja Android aplikacija *(eng. Android runtime)*, u nastavku ART, je sloj unutar kojega se nalaze metode koje se koriste za izvođenje aplikacija na mobilnoj platformi Android. ART sloj je svojevrstan virtualni stroj iz razloga mu je zadaća da svaka aplikacija koji se izvodi na platformi ima svoj proces koji je izdvojen od ostalih procesa, također uloga ARTa je da se pobrine za oslobađanje resursa koji su se prestali koristiti (eng, Garbage collection) i slične funkcionalnosti kakve ima i svaki virtualni stroj. ART za izvršavanje koristi DEX datoteke. DEX datoteka je datoteka koje sadrži binarni zapis kôda aplikacije i namijenjen je za izvođenje na ARTu. ART svoj kôd može kompilirati u dvije varijante, a to su kompiliranje unaprijed (*eng. Ahead-of-time*) i kompiliranje u trenutku korištenja (*eng. Just-in-time)*. Kompiliranje unaprijed kompilira kôd bez obzira na to hoće li se kompilirani kôd izvršavati i koliko će se puta izvršava, te kao takav postoji u radnoj memoriji, dok se kompiliranje po potrebi razlikuje u tome što se kôd ne kompilira odjednom, već se kompilira u trenutku kada se pojavi potreba za korištenjem samog dijela kôda.

ART također je zaslužan za alat za otklanjanje pogrešaka (*eng. Debugger*) koji se koristi prilikom otkrivanja i otklanjanja pogrešaka. ART omogućuje korištenje alata za kontrolu resursa koji si koriste za izvođenje aplikacije (eng. Profiler) koji omogućava trenutno praćenje stanja CPU, GPU, baterije, RAM te prati mrežni promet aplikacije.

3.1.4. C/C++ Native Libraries

Kako su prethodno opisani slojevi arhitekture pisani u C/C++ programskim jezicima, koristi sloj koji sadrži osnovne biblioteke koje omogućuju komunikaciju viših slojeva s nižim slojevima arhitekture. Unutar sloja se također nalaze proširene biblioteke koje olakšavaju korištenje osnovnih biblioteka.

Glavna karakteristika C/C++ programskih jezika je ta da omogućuje programiranje na niskoj razini i time omogućuju optimiziranije izvršavanje aplikacija s dostupnim resursima sustava. Radi tog se C/C++ programski jezik koristi za izradu jezgre operacijskog sustava i raznih kontrolera koji služe za upravljanje sklopovljem. ART je također pisan u C/C++ programskom jeziku.

3.1.1.5. Java API Framework

Sloj Java razvojnog okruženja je sloj s kojim programer ima najviše interakcije prilikom izrade prirodnih mobilnih aplikacija za mobilnu platformu Android.

Sloj je zaslužan za prevađanje kôda i metoda koje prirodno pisane u C/C++ programskom jeziku, u Java programski kôd i obrnuto. Unutar ovoga sloja se omataju C/C++ biblioteke unutar Java programskog jezika. Kako je Java viši programski jezik od C/C++ programski jezika, programeru se olakšava izrada aplikacije jer ima nema potrebe za optimizacijom kôda jer su već optimiziran unutar samog Java programskog jezika. U novije vrijeme se za izradu prirodnih Android aplikacija koristi i Kotlin programski jezik koji u pozadini kompilira u Java programski jezik (*Compiling and Running - Kotlin Programming Language*, 2020).

Unutar sloja Java razvojnog okvira sadržane su komponente koje se iznova iskorištavaju za izradu prirodnih mobilnih aplikacija za platformu Android. Te komponente su:

- **Sustav View komponenata (*eng. View System*)**. View je osnovni element unutar prirodnih Android mobilnih aplikacija koji se iscrtavaju na ekran uređaja te omogućuje korisniku interakciju s aplikacijom. Neki od elemenata View sustava su Button, TextView, EditText, ImageView i neke složenije kao što su RecyclerView, MapView, WebView i slično,
- **Upravitelj resursima (*eng. Resource Manager*)**, omogućuje korištenje resursa koji nisu napisani u programskome kôdu već su nekog drugog oblika. To su na primjer slike, ikone, zvučne datoteke koje se koriste unutar same aplikacije.
- **Upravitelj notifikacijama (*eng. Notification Manager*)** je komponenta koja se koristi prilikom pripreme i prikaza notifikacija korisniku. Upravitelj sadrži metode koje omogućuju kreiranje, prikazivanje i upravljanje notifikacijama na uređaju korisnika,

- **Upravitelj aktivnostima (*eng. Activity Manager*)**, je komponenta koja je zadužena za upravljanje aktivnostima (*eng. Activity*) koja se koristi unutar aplikacije.
- **Poslužitelj sadržaja (*eng. Content provider*)**, služi za dohvaćanje podataka koji se kreiraju i nalaze unutar drugih aplikacija, tako na primjer poslužitelj sadržaja omogućuje preuzimanje svih kontakata iz aplikacije kontakata.

3.1.1.6. System Apps

Sustavne aplikacije (*eng. System Applications*) su aplikacije koje dolaze sa samom mobilnom platformom. Same aplikacije nisu krucijalne za rad samog uređaja već olakšavaju korištenje uređaja. To su aplikacije koje su sama srž mobilnog uređaja kao što su Pozivi, poruke, web preglednik. Pred instalirane sistemske aplikacije se mogu promijeniti na tako da se instalira nova aplikacija koja zamjenjuje sve funkcionalnosti sistemske aplikacije.

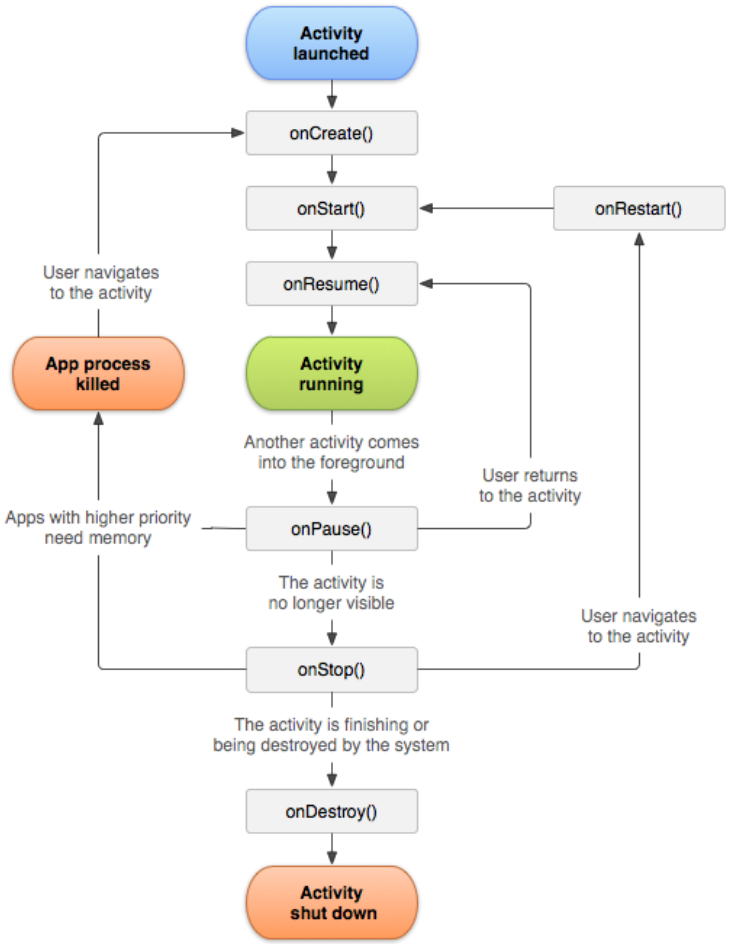
3.1.2. Životni ciklus aktivnosti i fragmenta

Prilikom razvoja prirodnih aplikacija napisanih za Android platformu koriste se dvije važnije komponente, a to su aktivnost i fragment. U nastavku će biti opisani životni ciklusi svake od njih.

3.1.2.1. Životni ciklus aktivnosti

Prilikom izvođenja aplikacije u svakom trenutku kada je aplikacija u prvome planu (*eng. Foreground*) tada je aktivna (*Understand the Activity Lifecycle | Android Developers*, 2020) jedna od aktivnosti od kojih je izrađena prirodna Android aplikacija. Glavna zadaća svake aktivnosti je upravljanje komponentama grafičkog sučelja i obrada korisničkih akcija koje se ostvaruju prilikom interakcije s grafičkim sučeljem aplikacije, te kontrola privremenim podacima koji se u tom trenutku nalaze unutar aplikacije.

Zato što je aktivnost jedna od glavnih komponenata svake aplikacije i ona zadužena za izvođenje pojedinih cjelina aplikacije, aktivnost ima svoje životne cikluse koji omogućavaju programeru bolju i lakšu kontrolu nad aktivnošću kako bi se mogli pripremiti potrebni resursi koji su tom trenutku potrebni za neometano izvođenje aplikacije. Događaji unutar životnog ciklusa jedne aktivnosti su prikazani su slikom 2.



Slika 2 Životni ciklus aktivnosti (izvor: https://developer.android.com/guide/components/activities/activity-lifecycle, 2020.)

Događaj *onCreate()* je prvi događaj u životnom ciklusu svake od aktivnosti. Događaj se poziva u trenutku kada je aktivnost kreirana i inicijalizirani su svi resursi koji su neophodni za izvršavanje aktivnosti. Unutar ovog događaja se preporučuje priprema svih resursa koji su potrebni za izvođenje trenutne aktivnosti, pod time ne misli na resurse koji su neophodni za rad aktivnosti već za resurse koji su potrebni programeru kako bi mogao izvesti potrebne zadatke svake aktivnosti.

Sljedeći događaj koji označava promjenu faze životnog ciklusa svake od aktivnosti je događaj *onStart().* Događaj označava da će se aktivnost prikazati na zaslonu uređaja. Unutar ovog događaja se preporučuje ponovna inicijalizacija svih resursa koji su oslobođeni tijekom kasnijih događaja životnog ciklusa aktivnosti.

Događaj *onResume()* je posljednji događaj životnog ciklusa unutar kojega se vrše pripreme za izvođenje aktivnosti. Događaj označava trenutak kada komponente grafičkog sučelja aktivnosti počinji osluškivati korisničke akcije. Po pozivu ovog događaja se preporučuje osvježavanje podataka koji se koriste za izvođenje aktivnosti.

Prethodno opisani događaji su događaji koji označavaju korake pripreme aktivnosti za rad, dok sljedeći događaji označavaju zaustavljanje životnog ciklusa aktivnosti.

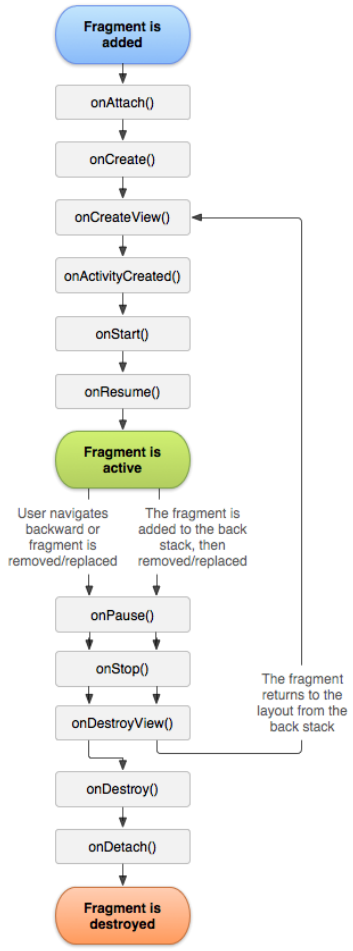
Događaj *onResume()* označava prelazak aktivnosti iz stanja aktivnog slušanja korisničkih akcija u stanje mirovanja. Aplikacija je i dalje vidljiva korisniku, ali korisnik ne može uspostaviti komunikaciju s aktivnošću. Tijekom poziva *onResume()* događaja preporučuje se oslobađanje resursa koji zahtijevaju veliku količinu memorije, te njihovo izvođenje ovisni samo o korisničkim akcijama. Nakon pozivanja ovog događaja životni ciklus može prijeći u jedno od dva stanja. To je povratak aktivnosti u aktivno stanje i tad je sljedeći događaj *onResume()* koji označuje da će aktivnost ponovo biti spremna slušati korisničke akcije ili pozivanje događaja *onStop()* koji označava da će aktivnost biti uklonjena sa zaslona uređaja.

Događaj *onStop()* označava trenutak u kojem se aktivnost uklanja sa zaslona ekrana i kao takva više nije vidljiva korisniku. Tijekom ovog događaja se preporučuje oslobađanje svih resursa koji se koriste unutar aktivnosti. Prelazak iz faze koja je uvjetovana događajem *onStop()* životni ciklus aktivnosti se može promijeniti u jedno od dva stanja. Prvo stanje je stanje kojemu prethodi događaj *onStart()* tijekom čijeg se prijelaza također okida i događaj *onRestart()* koji označava da će aktivnost ponovo prikazati korisniku te da će biti spremna slušati korisničke akcije, dok je drugi prijelaz, prijelaz koji označava događaj *onDestroy()*, odnosno označava da će aktivnost kao takva biti uništena i da se više neće moći prikazati na zaslonu ekrana uređaja.

onDestroy() događaj označava završnu fazu svake aktivnosti. Unutar ove faze operativni sustav uništava aktivnost i oslobađa resurse koje je aktivnost zauzimala za svoj rad, te oslobođene resurse preusmjerava drugim aktivnosti koje se izvode ili će se početi izvoditi.

3.1.2.2. Životni ciklus fragmenta

Druga komponenta koja se koristi prilikom izrade Android aplikacija je Fragment. Fragment kao samostalna komponenta nema svoj životni ciklus već se nadovezuje nad životni ciklus aktivnosti za koju je fragment u tom trenutku zakačen. Slika 3. prikazuje životni ciklus jednog fragmenta.



Slika 3 Životni ciklus Fragmenta (Izvor: https://developer.android.com/guide/components/fragments, 2020.)

Kao što je moguće vidjeti sa slike 3. neki od događaja iz životnog ciklusa fragmenta imaju isti naziv kao što je to slučaj kod aktivnosti. Nazivi događaja su isti zato što se izvršavaju slični procesi kao i kod okidanja istih događaja unutar životnog ciklusa aktivnosti. Iz tog razloga ti događaji neće biti ponovo opisivani, a to su događaji *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()* i *onDestoy().*

Prvi događaj životnog ciklusa svakog fragmenta je *onAttach()* koji označava da se fragment zakačio na aktivnost, te kao takav počinje postojati. Nakon ovog događaja fragment poprima životni ciklus tako da dijeli isti životni ciklus aktivnosti na koju je fragment zakačen. Tijekom ovog događaja se preporučuje priprema svih resursa koji zahtijevaju kontekst (*eng. Context*) same aplikacije.

Sljedeći događaj je događaj *onCreate()* nakon kojega dolazi *onCreateView*(). To je događaj u kojemu fragment dohvaća informaciju o tome kakvu će hijerarhiju grafičkih komponenata (*eng. View hierarchy*) imati fragment. Tijekom te faze preporučuje se jedino konstrukcija hijerarhije grafičkih komponenata, to se odvija na način da se "napuše" (*eng. Inflate*) jedna od XML datoteka koje sadrže opis rasporeda komponenata unutar hijerarhije grafičkog sučelja.

Sljedeći od događaja životnog ciklusa fragmenta je *onActivityCreated()* događaj koji označava da se aktivnost na koju je zakačen fragment završila sa izvođenjem događaja *onCreate()*.

Sljedeći događaj koji se okida, a da se razlikuje od događaja životnog ciklusa aktivnosti je *OnDestroyView*() događaj i on se poziva u trenutku kada je hijerarhija grafičkih komponenata fragmenta uništena, točnije fragment kao takav više nema svoje grafičko sučelje. Sljedeći događaj je *onDestroy()* koji označuje da će fragment osloboditi preostale resurse i time da više mijenjati stanja životnog ciklusa.

Posljednji događaj koji se događa unutar životnog ciklusa svakog fragmenta je događaj *onDetach()*. Tijekom tog događaja fragment se otkvačuje od aktivnosti na koju je fragment prije bio zakačen te se time uništava životni ciklus fragmenta i u konačnici sama referenca na taj fragment.

3.2. iOS

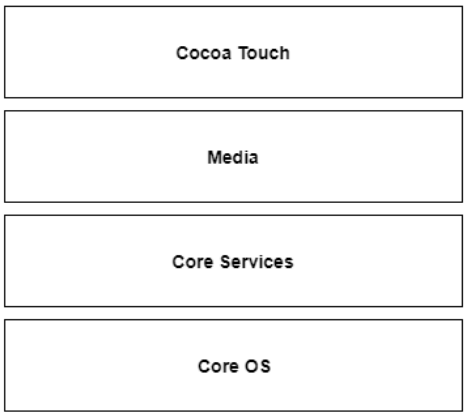
Uz mobilnu platformu Android, značajniji udio tržišta zauzima mobilna platforma iOS. Prema podacima (*Mobile Operating System Market Share Worldwide | StatCounter Global Stats*, 2020) udio platforme iOS na tržištu mobilnih platformi iznosi 25%.

Prema podacima s („IOS", 2020) iOS je platforma koja se danas koristi za Apple mobilne uređaje. Začetci platforme sežu u 2005. Godinu, kada je Steve Jobs dobio ideju ugraditi Mac unutar iPoda, točnije ideju razvijanja iPhonea uređaja. Tijekom razvijanja uređaja došli su do problema pitanja operativnog sustava za iPhone, stoga je i nastao sam operativni sustav odnosno platforma. Sama platforma postaje aktivna 2007. godine kada je i službeno u prodaju pušten prvi iPhone mobilni uređaj.

Isto kao i prethodno opisana mobilna platforma Android tako i iOS platforma sadrži više slojeva od kojih gradi arhitektura platforme. Jedna od razlika između kôd platformi Android i iOS je ta da je kôd iOS platforme zaštićen, te ga nije moguće čitati ili uređivati, kao što je to primjer kod kôda mobilne platforme Android.

3.2.1. Slojevitost arhitekture iOS platforme

Kao što je to slučaj kod Android platforme i iOS platforma je načinjena od slojeva unutar kojih se nalaze određene metode koje omogućuju izradu prirodnih iOS aplikacija. Jedna od većih razlika između platformi Android i iOS je ta da Android sadrži sloj Java razvojnog okvira koji omogućuje pisanje aplikacija koji nije C/C++ programski jezik koji se koristi u nižim slojevima arhitekture, dok iOS nema takav zato što je programski jezik korišten za pisanje iOS aplikacija Objectiv-C koji se lako može izvršavati nad C/C++ programskim jezikom (Koneva i ostali, 2014). Zbog čega nema potrebe za slojem koji prevađa jedan programski jezik u drugi i obrnuto kao što je to slučaj kod platforme Android.



Slika 4 Arhitektura iOS platforme (Izvor: https://www.tutorialspoint.com/apple-ios-architecture, 2020.)

Arhitektura iOS platforme (*Apple iOS Architecture*, 2018) je prikazana na slici 4. Sa slike se može vidjeti da je platforma podijeljena je u četiri sloja. Svaki sloj se sastoji od takozvanih razvojnih okvira koji su ubiti skup dostupnih metoda svakog sloja. To su sljedeći slojevi:

- **Core OS**, sloj operativnog sustava,
- **Core services**, sloj osnovnih servisa,
- **Media layer**, sloj koji omogućuje reprodukciju multimedijalnih elemenata aplikacije, te
- **Cocoa Touch**, sloj koji sadrži sve ostale biblioteke za rad aplikacije.

Glavna razlog između broja slojeva platformi Android i iOS je taj što prilikom izvođenja prirodnih iOS aplikacija nije potrebno prevađati programski jezik. Kod iOS arhitekture se samo izvođenje aplikacija izvršava unutar najnižeg sloja, odnosno same jezgre, dok ostali slojevi pružaju razne biblioteke i metode koje se koriste prilikom izrade prirodnih iOS aplikacija.

3.2.1.1 Sloj operativnog sustava

Osnovni sloj operativnog sustava (*eng. Core OS layer*) je sloj u kojemu se nalaze osnovne metode koje se koriste za pristup raznim komponentama sklopovlja uređaja na kojemu se izvršava mobilna platforma iOS. Sloj također ima ulogu izvršavanja procesa aktivnih aplikacija. Iz tog razloga programeri prirodnih iOS mobilnih aplikacija nemaju potpuno pravo pristupa do metoda unutar tog sloja, već je pristup ograničen s nekolicinom biblioteka.

Tako da se iz osnovnog sloja operacijskog sustava može pristupiti do sljedećih biblioteka:

- **Core Bluetooth Framework**, pruža metode koje se koriste za upravljanje bluetooth komponentom uređaja kao što su povezivanje, slanje te primanje paketa bluetooth prometa,
- **Accelerate Framework** (*Introduction to the Accelerate Framework in Swift | AppCoda*, 2017) pruža mogućnost obrade kompleksnih matematičkih operacija, obradu slike, obradu digitalnog signala uz manju složenost te time ubrzava samo izvođenje matematičkih operacija,
- **External Accessory Framework** (*External Accessory | Apple Developer Documentation*, 2020) pruža mogućnost spajanja aplikacije s vanjskim uređajima koji se spajaju preko Apple Lighting ili 30-pin connector ili bežično preko Bluetootha,
- **Security services Framework** pruža osnovne operacije koje se koriste za digitalnu sigurnost, a to su na primjer sinkrona i asinkrona kriptografija, izrada sažetaka, uspoređivanje sažetaka i slično, te
- **Local Authentication Framework** pruža metode kojima se može upravljati raznim senzorima koji se koriste prilikom autentifikacije korisnika uređaja.

3.2.1.2. Osnovni servisi

Sloj Osnovnih servisa (eng. Core services) je sloj koji proširuje sloj osnovnog operacijskog sustava, tako da omogućuje programeru pristup do nekih metoda i biblioteka iz nižeg sloja, koje nisu direktno dostupne programeru. Samim time sloj pruža veći broj biblioteka koje se omogućavaju programeru izradu prirodnih iOS aplikacija.

U nastavku će biti opisani neke od biblioteka koji se najčešće koriste prilikom razvoja prirodnih aplikacija za iOS platformu.

- **Address book Framework** je biblioteka platforme koja omogućava pristup developeru do kontakata korisnika,
- **Cloud Kit Framework** je biblioteka koja pruža potrebne akcije kako bi se aplikacija povezala s iCloud servisom, te kako bi se omogućio prijenos podataka u Cloud,

- **Core Foundation Framework** je biblioteka koja developeru omogućava pristup do osnovnih komponenata, kao što su upravitelj memorijom i raznim ostalim servisima,
- **Core Location Framework** je biblioteka koja pruža developeru informacije o lokaciji i sličnim informacijama koje su usko vezane uz lokaciju,
- **Core Motion Framework** je biblioteka koja developeru omogućuje korištenje raznih senzora na uređaju, koji se koriste unutra aplikacije,
- **Foundation Framework** je biblioteka koja se nadovezuje nad Core Foundation Framework tako da developeru omogućuje pristup do osnovnih metoda preko Objective-C jezika,
- **HealthKit Framework** je biblioteka koja sadrži sve potrebne metode kako bi se moglo koristiti podaci koji su vezani uz zdravlje korisnika.
- **HomeKit Framework** je biblioteka koja developeru pruža osnovne metode kojima se olakšava spajanje i kontrole vanjskih uređaja koji su povezani u mrežu,
- **Social Framework** je sučelje koje olakšava pristup do korisnikovih podataka koji su vezani uz razne društvene mreže.
- **StoreKit Framework** je biblioteka koja se koristi za povezivanje Apple store, te isto tako omogućuje plaćanje unutar aplikacije (eng. *In-App Purchase*).

3.2.1.3. Sloj reprodukcije multimedijalnog sadržaja

Sloj reprodukcije multimedijalnog sadržaja (*eng. Media layer*) je sloj koji sadrži biblioteke koje omogućuju i kontroliraju renderiranjem i reprodukcijom multimedijskog sadržaja na uređaju, kao na primjer crtanje grafičkog sučelja (UIKit graphics), prikaz grafike i animacija te reprodukcija audio i video sadržaja.

Sloj je podijeljen u dvije veće cjeline. Prva cjelina je skup biblioteka koje pružaju metode za upravljanjem grafičkih sadržajem (eng. Graphics Framework), dok je druga cjelina skup biblioteka koje omogućuju manipulaciju i reprodukciju audio i video sadržaja (eng. Audio Framework).

Skup biblioteka za reprodukciju grafičkog sadržaja:

- **UIKit Graphics** je biblioteka koja je namijenjena za manipulaciju prikazom i animacijama UIKit elementima grafičkog sučelja,
- **Core Graphics Framework** je biblioteka koja je namijenjena za renderiranje standardnih elementima.2D vektora i slika. je sučelje koje je namijenjeno za manipulaciju vektorskim elementima,
- **Core Animation** je biblioteka koja je zadužena za upravljanje animacijama koje se pojavljuju unutar samih aplikacija,

- **Core Images** je element koji je zadužen za procesiranje i analizu statičnih slika i za slike video sadržaja,
- **OpenGl ES i GLKit** su biblioteke koje su zadužene za obradu složenih 2D i 3D objekata koji se izvršavaju na fizičkim komponentama uređaja, te
- **Metal** sučelje koje omogućava izvršavanje složenih renderiranja i operacija na niskoj razini, točnije direktno izvršavanje na A7 procesoru uređaja.

Skup biblioteka za reprodukciju audio i video sadržaja:

- **Media Player Framework** je biblioteka koja se koristiti za reprodukciju audio sadržaja, te isto tako olakšava korištenje korisnikove iTunes kolekcije,
- **AV Foundation** je biblioteka koja je napisana u Objective C programskom jeziku te je namijenjena za reprodukciju i za kreiranje audio i video sadržaja, te
- **OpenAL** je standardizirana biblioteka za manipulaciju audio sadržajem.

3.2.1.4. Cocoa Touch Sloj

Cocoa Touch Layer je najviši sloj mobilne platforme iOS. Unutar tog sloja se spadaju neke osnovne biblioteke koje se često koriste za izradu prirodnih iOS aplikacija, i ostale biblioteke koje se ne nalaze unutar sloja, već dodaju unutar projekta naknadno, ovisno o potrebi.

Neke od biblioteke koje se nalaze unutar sloja su:

- **UIKit Framework** sadrži osnovne elemente korisničkog sučelja koji se koriste za razvoj mobilnih aplikacija te je zaslužan za životni ciklus komponenata, omogućuje upravljanje događajima kao što su geste, dodiri i slično,
- **AddressBook Framework** je biblioteka koja omogućuje pristup kontaktima na uređaju te upravljanje samim kontaktima koje pruža razvojni okvir kao što su korištenje, kreiranje, uređivanje i brisanje,
- **EventKit Framework** je biblioteka koja omogućuje rad s kalendarom i raznim događajima koji su zabilježeni unutar kalendara,
- **GameKit Framework** je biblioteka koja pruža osnovne metode koje se koriste prilikom izrada mobilnih igara, te samo metoda za povezivanje igrača u mrežu,
- **iAdKit Framework** je biblioteka koja sadrži osnovne metode kojima se omogućuje korištenje i upravljanje oglasima unutar aplikacije,
- **MapKit Framework** je biblioteka koja sadrži osnovne metode i elemente koji su potrebni za prikaz i korištenje Apple apa unutar aplikacije, te
- **MessageKit Framework** je biblioteka sadrži osnovne metode za primanje, slanje i kreiranje raznih vrsta poruka kao što su SMS, Email i slično.

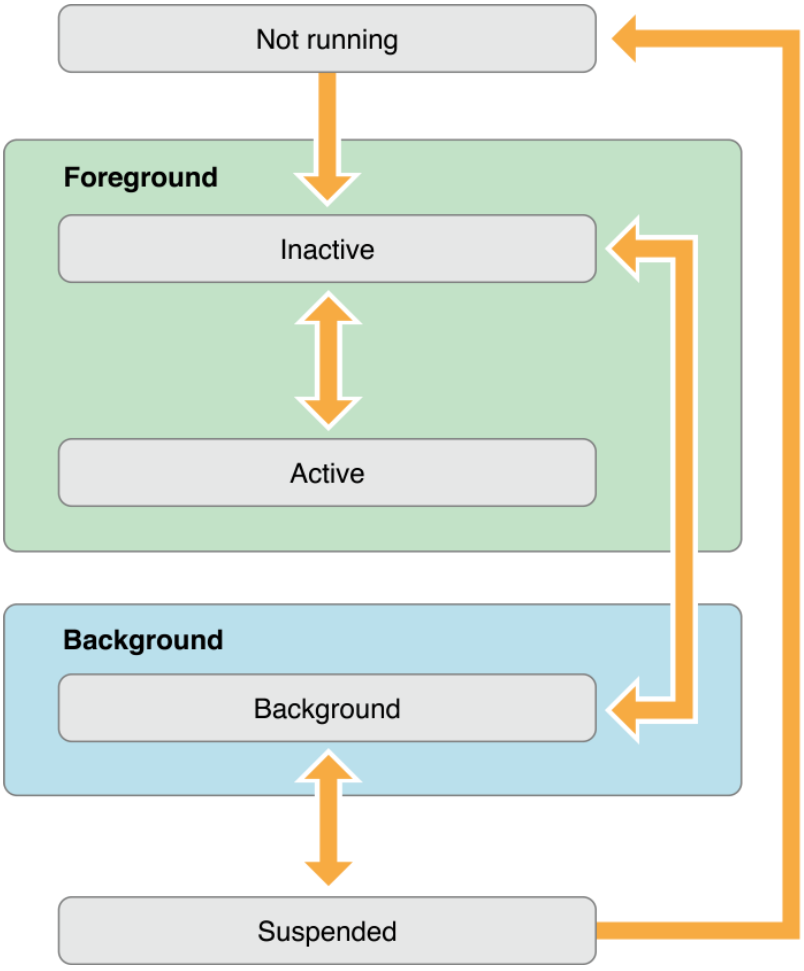
3.2.2. Životni ciklus Aplikacije i ViewControllera

Životni ciklus iOS aplikacije se može podijeliti na dva životna ciklusa, a to su životni ciklus cjelokupne aplikacije (AppDelegate) i životni ciklus jedne komponente točnije ViewControllera.

3.2.2.1. Životni ciklus aplikacije

Prilikom izvođenja jedne prirodne iOS mobilne aplikacije, koristi se AppDelegate komponenta koja služi za osluškivanje događaja životnog ciklusa aplikacije. Prilikom pokretanja aplikacije prvi događaj koji se okine prije nego se aplikacija pojavi na zaslonu uređaja je AppDelegate i iz koje se pokreče korijenski ViewController te aplikacije.

Tijekom životnog ciklusa aplikacije, aplikacija se može nalaziti u nekoliko stanja. Prema (*iOS Application Life Cycle Example*, 2018) stanja životnog ciklusa jedne iOS mobilne aplikacije (slika 5) su:



Slika 5 Životni ciklus iOS aplikacije (Izvor: https://medium.com/@neroxiao/ios-app-life-cycle-ec1b31cee9d , 2020.)

- **DidFinishLaunchingWithOptions()** je događaj koji se poziva nakon inicijalizacije korijenskog ViewControllera koji je zadužen za cjelokupnu aplikaciju. U tome trenutku aplikacija postaje živa unutar operacijskog sustava,
- **applicationDidBecomeActive()** događaj koji se okida u trenutku kada je aplikacija spremna za korištenje, odnosno kada aplikacija iz stanja mirovanja prelazi u aktivno stanje u kojemu se odvija aktivni dio rada aplikacije.
- **applicationWillResignActive()** je događaj koji se poziva u trenutku kada aplikacija prelazi iz aktivnog stanja u neaktivno stanje, odnosno kada se aplikacija prestaje prikazivati na zaslonu uređaja,
- **applicationDidEnterBackground()** događaj koji se poziva u trenutku kada aplikacija više nije vidljiva korisniku i sprema se biti neaktivna. Kao takva aplikacija je i dalje aktivna i nije uništena od strane operacijskog sustava platforme,
- **applicationWillEnterForeground()** događaj koji poziva u neposredno prije nego se aplikacija počine ponovo prikazivati na zaslonu uređaja. U trenutku poziva događaja aplikacija se nalazi u pozadinskome radu i prelazi u stanje aktivnog izvođenja.
- **applicationWillTerminate()** događaj koji signalizira finalnu fazu životnog ciklusa AppDelegate komponente. Tijekom tog događaja se oslobađaju svi resursi koji su bilo korišteni za aplikaciju kako bi ih operacijski sustav mogao dodijeliti drugim aplikacijama.

3.2.2.2. Životni ciklus ViewControllera

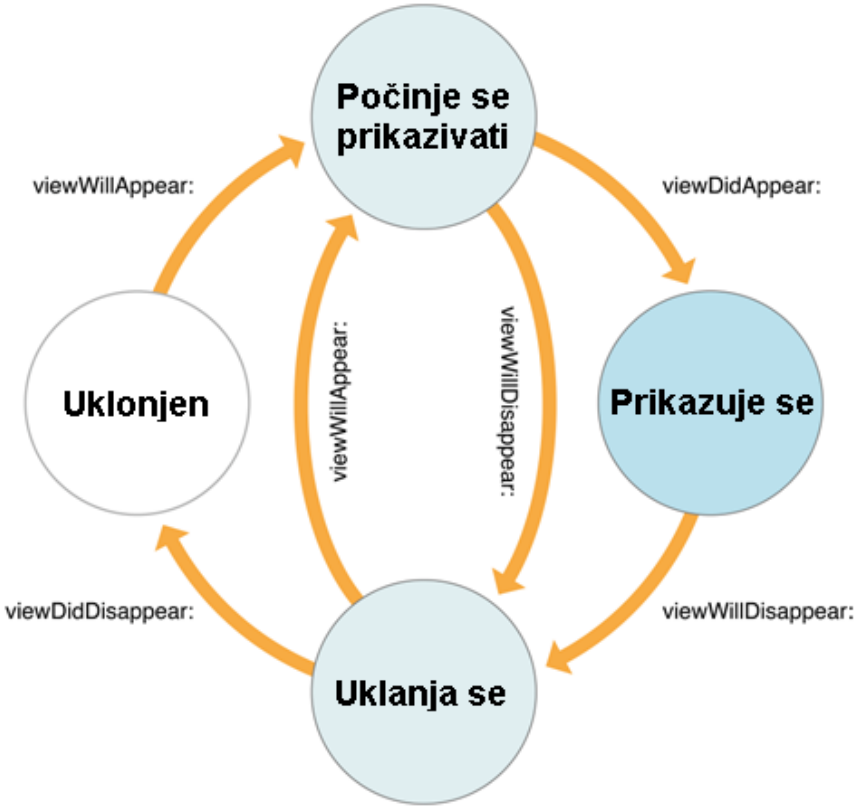
Komponenta na kojoj je bazirana svaka prirodna iOS aplikacija, uz AppDelegate je ViewController. ViewController je komponenta koja je zadužena za slušanje i obradu korisničkih akcija koje se ostvaruju preko grafičkog sučelja aplikacije.

Prema podacima sa (*Start Developing iOS Apps (Swift): Work with View Controllers*, 2020) životni ciklus UIViewController-a se može podijeliti u četiri stanja koji su:

- Stanje pripreme prikaza (*eng, Appearing*), tijekom tog stanja ViewController priprema potrebne resurse kako bi se omogućio prikaz grafičkih elemente na zaslonu uređaja
- Stanje prikazivanja (*eng. Appeared*) je stanje kada je ViewController preko svojih grafičkih elemenata prikazan na zaslonu uređaja,
- Stanje gašenje (*eng. Disappearing*) tijekom kojeg stanja ViewController oslobađa nepotrebne resurse i uklanja grafičke elemente, i
- Skriveno stanje (eng. Disappeared) tijekom kojeg ViewController nema nikakve veze za korisnikom aplikacije, točnije ViewController nije prikazan na zaslonu uređaja.

i sadrži sljedeće događaje životnog ciklusa (Slika 6):

- **ViewDidLoad()** događaj koji se okida prilikom inicijalnog pokretanja ViewControllera i označava da je ViewController spreman za korištenje i da su svi njegovi resursi spremni. Tijekom ovog događaja se preporučuje kreiranje svih potrebnih resursa koji će se koristiti unutar View controllera.
- **ViewWillAppear()** je događaj koji se okida neposredno u trenutku kada će se grafičke komponenta ViewControllera prikazati na zaslonu uređaja. Prilikom poziva ovog događaja se preporučuje osvježavanje stanja podataka koji se stalno ažuriraju te je potrebno najnovije stanje istih, također se preporučuje priprema svih animacija koje se odvijaju unutar ViewControllera.
- **ViewDidAppear()** događaj koji se okida u trenutku kada je ViewController i njegove pripadajuće komponente grafičkog sučelja uspješno prikazale na zaslonu uređaja i spremne su za slušanje i obradu korisničkih akcija. Tijekom ovog događaja se preporučuje priprema svih slušača koji se koriste za slušanje i obradu korisničkih akcija.
- **ViewWillDisappear()** je događaj koji se okida neposredno prije samog uklanjanja View Controllera i njegovih komponenata grafičkog sučelja sa zaslona uređaja. Tijekom ovog događaja preporučuje se zaustavljanje svih animacija i oslobađanje svih resursa koji zahtijevaju veliku količinu resursa za njihovo izvođenje, a njihov rezultat se koristi samo za elemente grafičkog sučelja. Npr. Ako se koriste mape po pozivu ovog događaja preporuča se pauziranje operacija kojima se dohvaća trenutna lokacija korisnika, a da se ta lokacija koristi samo kako bi se prikazala na zaslonu uređaja,
- **ViewDidDisapear()** je događaj koji se okida nakon što je ViewController zajedno sa pripadajućim komponentama grafičkog sučelja uklonjen sa zaslona uređaja. Tijekom ovog događaja preporučuje se oslobađanje svih resursa koji se koriste tijekom rada View controllera.



Slika 6 Životni ciklus ViewController-a (Prema: https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/WorkWithViewControllers.html, 2020.)

3.3. Flutter

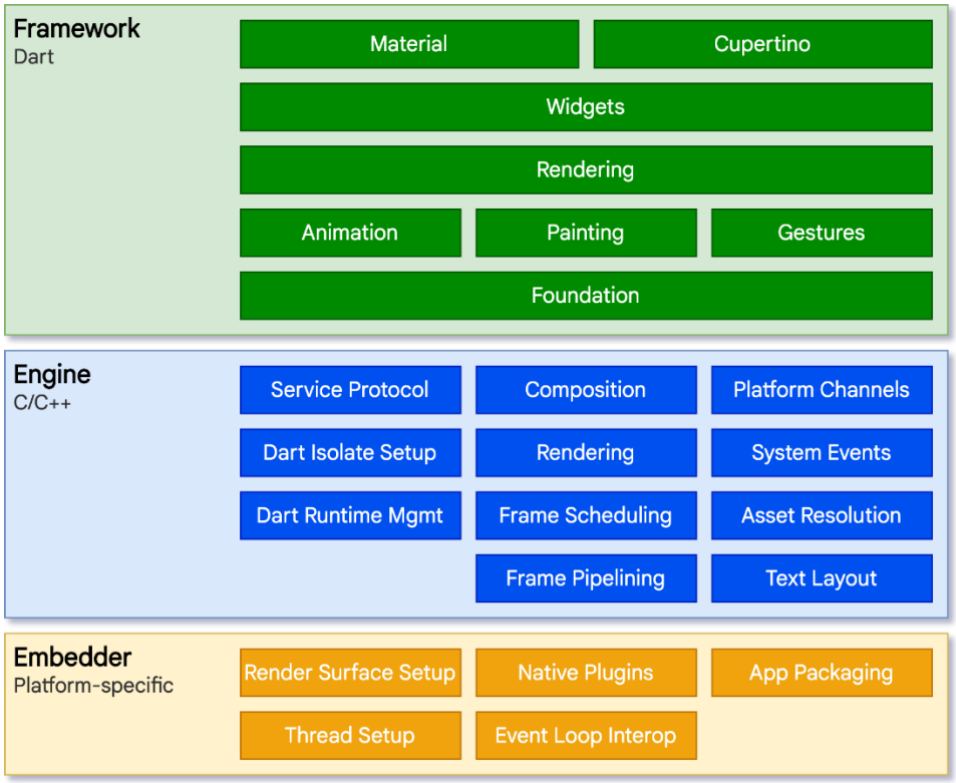
U prethodna dva poglavlja su opisane arhitekture mobilnih platformi Android i iOS. Unutar ovog poglavlja biti opisivan Flutter, koji za razliku od Androida i iOSa nije mobilna platforma, već je razvojni okvir (*eng Framework*) koji omogućava izradu više-platformskih aplikacija.

Prema podacima (*A brief Introduction to Flutter | Dream Squad - We Develop*, 2019) Flutter se prvi puta spominje 2015. godine na Ljetnoj konferenciji Dart, gdje je u sklopu te konferencije predstavljen pod kodnim imenom “SKY” (hrv. Nebo). Prva verzija koja je bila dostupna developerima izašla je 2017. godine, dok je prva službena verzija Fluttera izašla 2018. godine. Glavna razlika između Fluttera i ostalih razvojnih okvira za razvoj mobilnih aplikacija je ta da se okvir ne ponaša kao omotač oko sistemskih funkcija platforme kao na primjer razvojni okviri napisani unutar JavaScript programskog jezika već koristi svoj Flutter stroj za interpretaciju aplikacija koji direktno koristi resurse same platforme, a ne SDK platforme na kojoj se izvršava Flutter aplikacija.

3.3.1. Slojevitost arhitekture

Flutter (*Flutter Architectural Overview*, 2020) nije mobilna platforma, te kao takav nema jezgru koja i bila osnova operacijski sustava koja bi bila zadužena za izvršavanje aplikacija. Zbog toga što Flutter nije operacijski sustav nema direktnu vezu prema fizičkom sklopovlju na uređaju na kojemu se trenutno izvodi. Zbog toga što Flutter nema elemente kao što su jezgra ili sloj sklopovske podrške slojevitost Fluttera je mnogo manja od prethodno opisanih mobilnih platformi.

Arhitektura Fluttera je prikazana na slici 7. Sa slike se može vidjeti da je arhitektura Fluttera podijeljena unutar tri sloja, gdje je prvi i glavni sloj sa sama platforma (*eng. Embedder*) na kojoj se izvršava stroj za izvođenje Flutter aplikacija (*eng. Flutter engine*). Nad slojem platforme izvoditelja Flutter stroja, nalazi se sloj samog Flutter stroja, koji se koristi za izvođenje Flutter aplikacije. Nad slojem Flutter stroja nalazi Dart razvojni okvir (*eng. Dart Framework*) koji pruža osnovne metode i biblioteke od kojih se gradi Flutter aplikacija



Slika 7 Arhitektura Flutter programskog okvira (Izvor: https://flutter.dev/docs/resources/architectural-overview, 2020.)

Prema slici 7 koja prikazuje slojeve arhitekture Flutter razvojnog okvira su slojevi su podijeljeni u tri sloja i to su slojevi:

- Platforma koja izvršava stroj za izvršavanje Flutter aplikacija, (eng. Embedder),

- Sloj stroja za izvršavanje Flutter aplikacija, (eng. Engine), te
- Dart razvojni okvir koji pruža osnovne biblioteke za izradu Flutter aplikacija. (eng. Dart Framework)

3.3.1.1. Platforma izvođača

Važno je napomenuti kako se ovaj sloj neće detaljno obraditi, već će se spomenuti koje se komponente unutar sloja koriste kako bi se Flutter stroj povezao s resursima platforme. Za detaljniju podjelu arhitekturu platformi može se pročitati unutar poglavlja razrade arhitektura platformi Android i iOS.

Glavne značajke sloja platforme izvršitelja su te da omogućavaju stroju za izvođenje Flutter aplikacija spajanje na platformu kako bi se iskoristili potrebni resursi za izvođenje Flutter aplikacije.

- **Postava pripreme zaslona uređaja (eng. Render Surface Setup)** unutar ovoga elementa sloja nalaze se metode koje su zadužene za povezivanje površine na kojoj se iscrtava grafičko sučelje (*eng. Canvas*) na zaslonu uređaja, s Flutter strojem kako bi se elementi grafičkog sučelja Flutter aplikacije mogli direktno iscrtavati na zaslon uređaja,
- **Prirodne značajke platforme (*eng. Native plugins*)** je komponenta sloja koja omogućuje priključivanje Flutter stroja sa specifičnostima pojedine platforme koje je. Potrebno iskoristiti unutar Flutter aplikacije,
- **Upravitelj aplikacijskim paketima (eng. App Packing)** je komponenta sloja koje je kreirana razloga što Flutter nije platforma već razvojno okruženje te radi toga postoji problem upravljanja paketima Flutter aplikacije na platformi. Tako da je ovaj sloj zaslužan za povezivanje upraviteljem paketima platforme kako bi se mogle izvršavati operacija kao što su spremanje, pokretanje i upravljanje izvođenje aplikacija,
- **Postava sustava dretvi (*eng. Thread Setup*)** je komponenta sloja koja upravlja dretvama koje se koriste za povezivanja Flutter stroja s resursima platforme koje su potrebne stroju za njegovo neometano izvršavanje.
- **Upravitelj prekidima (eng. Event Loop Interop)** je komponenta sloja koja se koristi za upravljanje raznim prekidima platforme i prekidima koji se ostvaruju unutar Flutter stroja, te njihovo međusobnu razmjenu između platforme i stroja.

3.3.1.1. Flutter stroj za izvršavanje Flutter aplikacija

Flutter stroj za izvršavanje Flutter aplikacija (*Making the Most of Flutter: From Basics to Customization | by Alibaba Tech | HackerNoon.com | Medium*, 2018) je sloj razvojnog okvira koji omogućuje izvođenja aplikacije na bilo kojoj platformi za koju je dostupan Flutter stroj.

Prema podacima (*The Engine architecture · flutter/flutter Wiki · GitHub*, 2020) sam Flutter stroj nema direktnu vezu s dretvama koje pruža platforma za izvršavanje stroja. Iz tog razloga Flutter stroj ovisno o platformi na kojoj se nalazi može se izvršavati uz pomoć jedne ili više dretvi. Ako se Flutter stroj izvršava na više dretvi tada je svaka dodijeljena dretva zaslužna za izvršavanje jednog od zadataka kao što su renderiranje, izvršavanje prekida i ulazno izlaznih signale.

Flutter stroj za izvođenje Flutter aplikacija napisan je u C/C++ programskome jeziku i koristi se kao veza između platforme i Dart razvojnog okvira. Glavna zadaća stroja je upravljanje ulaznim i izlaznim signalima, tekstualnim podacima, upravljanje datotekama i mrežnim podacima, te najvažniji zadatak mu je samo renderiranje i prikazivanje grafičkih elemenata. Komponente unutar Flutter stroja su sljedeći:

- **Servisni protokol (eng. Service protocol)** je komponenta koja služi za uspostavljanje komunikacije između Flutter stroja i aplikacije, tako na primjer uz pomoć servisnog protokola moguće je dohvatiti trenutne postavke, verziju, trenutno stanje u kojemu se nalazi stroj, informaciju o tome koliko se slika izmjenjuje u sekundi i tako dalje,
- **Kanali platforme *(eng. Platform channels)*** je komponenta sloja koja omogućuje da aplikacija zaobiđe sloj Flutter stroja tako da se aplikacija direktno spaja na samu platformu. Pomoći kanala se omogućuje korištenje specifičnosti platforme a nisu dostupni unutar Fluttera, na primjer to su biblioteke koje omogućavaju spremanje preferencija korisnika, a to su plist za iOS ili SharedPreferences za Android, ili upravljanje alarmima i tako dalje,
- **Postava za izvođenje izoliranih Dart dijelova kôda *(eng. Dart Isolate setup)*** (*Flutter_isolate | Flutter Package*, 2020) je komponenta sloja koja omogućuje izvođenje pojedinih zadataka asinkrono u odnosu na izvođenje ostatka kôda. Komponenta se koristi prilikom spajanja kanala platforme s platformom. Kako Flutter nije operacijski sustav te nema svoj sustav dretava, Flutter nema dretve koje se koriste za izvršavanje asinkronih zadataka, već je kreirana komponenta koja ima slična svojstva kao i dretva i to je Dart Isolate,
- **Prikazivanje (*eng. Rendering*)** je komponenta sloja koja je zadužena za iscrtavanja zadanih elemenata na zaslon ekrana te za pripremu elemenata koji su sljedeći na redu da se iscrtaju na zaslon uređaja,
- **Sistemski događaji (*eng. System events*)** je komponenta sloja čija je uloga upravljanje sistemskim događajima Flutter aplikacije,

- **Dart VM Management j**e komponenta Flutter stroja koja je zadužena za upravljanje memorijom, kao što su dodjeljivanje i oslobađanje memorije koja je zauzeta a trenutno se više ne koristi,
- **Komponenta raspoređivanja okvira (eng. Frame scheduling)** je komponenta unutar sloja koja je zadužena za upravljanje i pravilno raspoređivanje slika za njihovo pravilno iscrtavanje na zaslon uređaja.
- **Komponenta upravljanja vanjskim sadržajem aplikacije (*eng. Asset resolution*)** je komponenta Flutter stroja koja upravlja vanjskim sadržajem aplikacije odnosno zaslužna je za upravljanje njegovim referencama prema mjestu u memoriji gdje se nalazi prema kôdu.
- **Cjevovod okvira (eng. Frame pipeling)** je komponenta koja je zadužena za upravljanje zahtjevima i brine se da se zahtjev za iscrtavanjem na zaslon uređaja uspješno dopremi do komponente za prikazivanje.
- **Tekstualni tlocrt (eng. Text layout)** je komponenta Flutter stroja koja se koristi za prikaz i manipulaciju teksta unutar same aplikacije. Ona je glavna osnova za sve Widgete koji se oslanjaju na tekst, bilo to kao Widget za prikaz ili kao Widget za unos teksta

3.3.1.2. Dart Framework

Dart razvojni okvir je najviši sloj unutar arhitekture Flutter razvojnog okvira i on se koristi za izradu aplikacija. Sloj se sastoji od komponenata koje grade grafičko sučelje aplikacije. Unutar sloja također se nalaze komponente koje su zadužene za pripremu grafičkih komponenata koje se prosljeđuju Flutter stroju koji ih iscrtava na zaslon uređaja. Elementi Dart Framework sloja su sljedeći:

- **Komponenta osnovnih elemenata (eng. Foundation)** je komponenta sloja unutar koje se nalazi osnova za sve ostale komponente Dart razvojnog okvira. Unutar komponente su sadržani elementarni elementi svakog widgeta,
- **Komponenta animacija (eng. Animation)** je komponenta koja je sadrži osnovne metode koje se koriste za pripremu i reprodukciju animacija svake komponente grafičkog sučelja,
- **Komponenta pripreme teksture (eng. Painting)** komponenta koja je zaslužna za pripremu teksture koja se koristi na widgetu, bilo to jednostavno poput boje ili nešto složenije poput složene teksture,
- **Komponenta gesti (eng. Gestures)** je komponenta koja se koristi za upravljanje raznim gestama koje se kače na widgete kako bi osluškivali korisničke akcije, a to su na primjer pritisak gumba, klizanje po klizaču i slično.

- **Iscrtavanje (*eng. Rendering*)** je komponenta koja priprema same widgete za njihovu daljnju obradu, distribuciju i u konačnici iscrtavanje na zaslon uređaja,
- **Widgets** glavni elementi sučelja jedne Flutter aplikacije. Postoje dvije vrste widgeta, to su Widget sa stanjem (eng. *Statefull Widget*) i widget bez stanja (*eng. Stateless Widget*). Widget sa stanjem je element koji ima stanje koje se koristi kako bi se zapamtilo moglo pratiti promjene nad widgetom. Widget sa stanjem se prilikom ponovnog poziva za osvježavanjem Widget popunjava s podacima koji se nalaze unutar objekta stanja widgeta, dok je widget bez stanja takav widget da prilikom novog iscrtavanja na zaslon ekrana ne pamti prethodno stanje, odnosno se prilikom iscrtavanja generira novi objekt koji predstavlja widget na zaslonu uređaja, te
- **Material i Cupertino** su elementi sloja Dart razvojnog okvira koji sadrže widgete izrađenih prema prirodnim elementima pojedine platforme. Material skup komponenta je skup widgeta koji su napravljeni prema smjernicama Material dizajna i oni se najčešće koriste za izradu Android aplikacija, dok je skup Cupertino, skup iOS widgeta koji su bazirani prema smjernicama Human interface dizajna.

Kao što se vidi Dart razvojni okvir je baziran na widgetima, i sve komponente u slojevima arhitekture Fluttera su kreirane tako da upravljaju widgetima. Widgeti se tijekom izvođenja prosljeđuju Flutter stroju koji ih iscrtava na zaslon uređaja uz pomoć svojih komponenata koje se nalaze unutar sloja Flutter stroja.

3.3.2. Životni ciklus Widgeta

Kao što je prethodno spomenuto, widget je glavna komponente čijom se kombinacijom kreira Flutter aplikacija. Za razliku od Androida i iOSa gdje je se jedna komponenta grafičkog sučelja direktno iscrtava zaslon uređaja i prilikom promjene stanja svi se njegovi elementi tog grafičkog sučelja iznova iscrtavaju na ekran, dok se kod Fluttera (*StatelessWidget class - widgets library - Dart API*, 2020) svaki widget ponaša kao jedna, zasebna komponenta iz koje se generira element objekt koji je zadužen za kreiranje i upravljanje objektom za iscrtavanje (eng. *RenderObject)* koji se iscrtava na zaslon uređaja. Prilikom izmjene stanja widgeta, widget stanje prosljeđuje svojem element objektu koji prema tome koristi isti objekt za iscrtavanje ili kreira novi, radi toga se postiže to da nije potrebno ponovo iscrtavati cijelu hijerarhija grafičkog sučelja, već sam komponentu koja je promijenila svoje stanje.

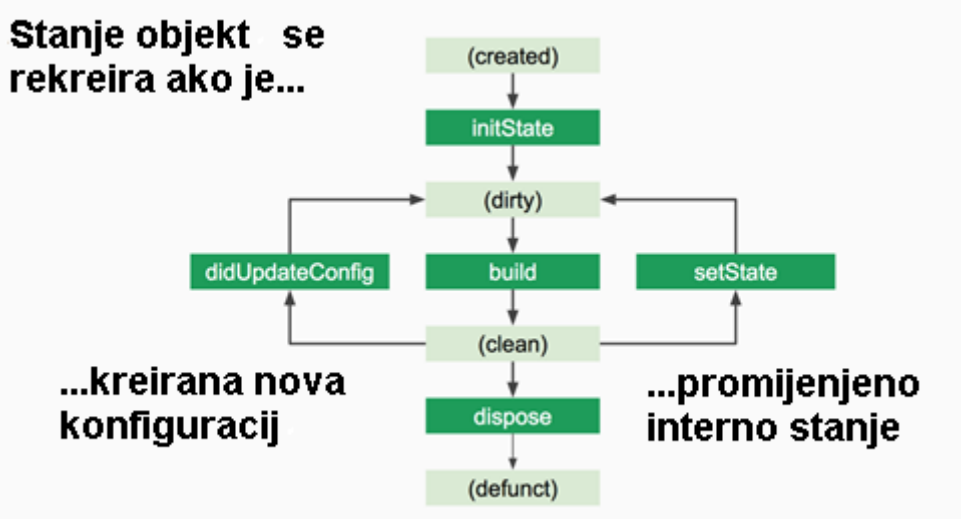
Kao što je prethodno spomenuto, postoje dvije vrste widgeta, a to su widget sa stanjem (eng. Statefull Widget) i Widget bez stanja (eng. Stateless Widget). Widget bez stanja (*[Newbie] Chapter 4. Widget's state - nhancv's blog*, 2019) ima samo jedno stanje i jedan događaj, *build()*. Nakon kreiranja objekta widgeta, okida se događaj *build().* Zbog toga widget

nema promjenu stanja, odnosno takav widget nikada se ne mijenja, što znači da se prilikom promjene podataka koji se nalaze unutar widgeta ne izmjenjuje njegova komponenta element koja je zadužena za kreiranje objekta za iscrtavanje.

Widget sa stanjem je za razliku od widgeta bez stanja, widget kod kojega se može mijenjati unutarnje stanje i promjenom nad podacima te promjene se iznova iscrtavaju na zaslon uređaja s novim podacima. Glavna razlika između widgeta bez stanja i widgeta sa stanjem je ta da se ne okida događaj *build(),* već se okida događaj *createState().* Prilikom okidanja događaja *createState()* stvara se objekt stanja (*eng. State*) za taj widget. Objekta stanja za razliku od widgeta ima životni ciklus. Događaj *build()* kod widgeta sa stanjem se događa unutar njegovog objekta stanja.

Widgeti su elementi koji sami po sebi ne mogu mijenjati svoje stanje, točnije rečeno nemaju svoj životni ciklus. Radi toga postoje dvije vrste widgeta koji se razlikuju po tome dali sadrže stanje ili ne sadrže. Widgetu sa stanjem, kada mu se ukloni objekt stanja ima ista svojstva kao i widget bez stanja.

Objekt stanja (*eng. State*) jednog widgeta ima nekoliko događaja životnog ciklusa, koji su prikazani na slici 8.



Slika 8 Životni ciklus StatefullWidget-a (Izvor: https://nhancv.com/newbie-chapter-4-widgets-state/, 2020.)

Objekt stanja ima pet događaja koji se pozivaju tijekom promjene stanja objekta. Ti događaji su sljedeći:

- **InitState**, je događaj koja se okida neposredno nakon samog kreiranja objekta stanja. Unutar događaja *InitState()* preporuča se inicijalizacija stanja i podataka koji se koriste unutar objekta stanja,

- **Build** je događaj koji signalizira da se kreira element koji će je se koristi za iscrtavanje na zaslon uređaja. Događaj *build()* ima ista svojstva kao i događaj *build()* unutar objekta widgeta, jedina je razlika u tome koliko se puta okida događaj. Unutar objekta stanja događaj se poziva svaki puta kada se stanje označi da je prljav (eng. Dirty), odnosno da se je promijenilo unutarnje stanje objekta stanja i da je potrebno izmijeniti kreirati novi objekt za iscrtavanje. Nakon što se pozove događaj *build(),* unutar objekta stanja, objekt mijenja oznaku iz prljavo u oznaku koja označava da je objekt stanja čist (eng. Clean). To znači da prilikom sljedećeg perioda osvježavanja stanja aplikacije neće okidati događaj *build()* objekta stanja. Objekt stanja može ponovo izmijeniti oznaku svoju oznaku da se je izmijenilo stanje, i to se događa na okidanje jednoga od sljedeća dva događaja, a to su događaji *setState()* i *didUpdateConfig()*.
- **SetState** je događaj koji označava da su se podaci unutar objekta stanja izmijenili te da je potrebno osvježiti objekt za iscrtavanje widgeta.
- **DidUpdateConfig** je događaj koja okida ako dođe do promjena unutar roditeljskog objekta widgeta. Za razliku od događaja *setState()* koja se poziva vanjskim putem, događaj *didUpdateConfig()* se poziva od strane aplikacije kada je potrebno ažurirati widget kada se promjene podaci unutar jednoga od njegovih elementa koji su hijerarhijski iznad widgeta a widget ima vezu prema tim podacima,
- **Dispose** je događaj koji označava da će se objekta stanja obrisati.

Nakon što su opisani arhitekturni slojevi mobilnih platformi Android i iOS i arhitekturni slojevi Flutter razvojnog okvira može se vidjeti kako je glavna razlika između platformi i razvojnog okvira ta na koji način se aplikacije pripremaju za izvođenje i način na koji se izvođenje. Kod platformi se ti procesi izvode unutar operacijskog sustava platforme, dok se Fluttera priprema aplikacije i izvođenje aplikacija izvršava nad strojem za izvođene aplikacija, koji koristi resurse platforme na kojoj se stroj izvršava.

Također se razlikuju i životni ciklusi komponenti koje se koriste prilikom izrade. Kod mobilnih platformi, životni ciklusi se odnose na komponente cijelog grafičkog sučelja jednog ekrana, dok to nije slučaj kod Fluttera, gdje svaka komponenta koja se nalazi unutar hijerarhije grafičkog sustava može ali ne mora ima ti životni ciklus. Ako postoji komponenta sa životnim ciklusom, tada taj životni ciklus nije dijeljen s ostalim komponentama unutar hijerarhije grafičkog sučelja. Time se postiže ponovno iscrtavanje samo onih elemenata grafičkog sučelja koji su izmijenili svoje stanje, a ne iscrtavanja cijelog grafičkog sučelja tog ekrana.

4. Arhitekture aplikacija

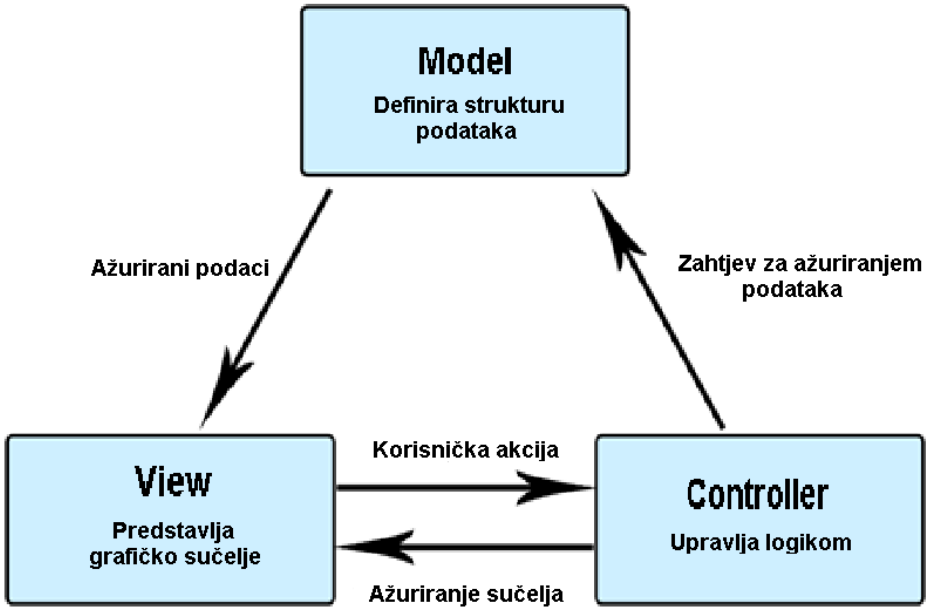
Nakon što su prethodno opisane arhitekture mobilnih platformi i životni ciklusa glavnih elemenata od kojih su izrađene aplikacije, upoznati smo s način kako se izvršavaju aplikacije. U nastavku će se proučiti jedna od osnovnih arhitektura pomoću koje se izrađuju aplikacije, kako bi se aplikacija bila kvalitetnije izrađena, te kako bi se olakšalo daljnje održavanje aplikacije.

Arhitekture aplikacija imaju neke poveznice s arhitekturama zgrada. Glavni zadatak arhitekta je da zgrada koju projektira iskoristi što više prostora, da je funkcionalna te da lijepo izgleda, a da pri tome zgrada može izdržati nepogode koje je mogu zadesiti, te da je sigurna za stanovanje. Tu se može povući paralela s izradom aplikacija. Svaka aplikacija se "izgrađuje" s idejom da bude što jednostavnija i jasnija za čitanje točnije daljnje održavanje, a da pri tome može omogućuje veću protočnost podataka i veći broj akcija koje se odvijaju unutar same aplikacije te koje korisnik pokreće svojim korištenjem.

Od početka izrada aplikacija pa do danas stvorene su razne konvencije, uzorci dizajna i u konačnici arhitekture koje omogućavaju svojom uporabom rješavaju neke zajedničke probleme izrade kostura svake aplikacije. Izrada aplikacije po nekom poznatih uzoraka olakšavaju daljnje održavanje i isto tako olakšavaju problem. upoznavanja novog programera s aplikacijom. S tim razlogom će se u nastavku detaljnije opisati MVC arhitektura.

MVC arhitektura je osmišljena s razlogom da se riješi problem monolitnih struktura aplikacija, tako da se logika podijeli u tri smislene komponente. Prva komponenta je komponenta koja je zadužena za prikaz informacija korisniku, druga komponenta je zadužena za upravljanje i povezivanje ostalih komponenata, dok je treća komponenta zadužena za obradu i pripremu podataka koji će se koristiti unutar komponenata.

Veze između komponenata MVC arhitekture prikazane su slikom 9. Na slici se može vidjeti kako je komponenta pregleda (*eng. View*) zadužena za prikaz podataka, komponenta upravitelja (*eng. Controller*) je komponenta čiji je zadatak obrada korisničkih akcija koji se dohvaćaju preko komponente pregleda, isto tako je zadužena za direktnu komunikaciju s komponentom modela (*eng. Model)*. Veza između komponente pregleda i modela je takva da model informira o pregled tome da su podaci ažurirani.



Slika 9 Veza MVC arhitekture (Prema: https://developer.mozilla.org/en-US/docs/Glossary/MVC, 2020.)

Prema knjizi (Frank Buschmann i ostali, 1996) MVC arhitektura je podijeljena na tri komponente, a to su:

- **Komponenta pregleda (*eng. View*)** je komponenta koja je zadužena samo za prikaz korisničkog sučelja te prikaz podataka iz modela koji se dohvaćaju prilikom obavještavanja Modela da je Model promijenio podatke i da je potrebno ažurirati stanje komponente pregleda. Isto tako komponenta pregleda zadužena je za osluškivanje korisničkih akcija koje prosljeđuje upravitelju kako bi mogao obraditi korisničke akcije,
- **Komponenta upravitelj (*eng. Controller*)** je glavna komponenta unutar MVC arhitekture i ona je zadužena za povezivanje komponenata pregleda i modela, te je zadužen za obradu korisničkih akcija. Ovisno o korisničkoj akciji koja se obradi unutar upravitelja, upravitelj poziva metode iz modela kojima se pokreće zahtjev za obradom ili dohvaćanjem podataka koji se koriste unutar komponenata arhitekture,
- **Model komponenta (eng. Model)** koja je zadužena za obradu i dohvaćanje podataka koji se koriste unutar MVC arhitekture. Model unutar sebe sadrži slušače koji se obavještavaju o promjeni podataka, od metoda koje su izložene za pristup podacima i metoda koje pokreću određenu akciju unutar modela, te od zaštićenih metoda koje se koriste za obradu akcija unutar samog modela.

4.1. Primjer inicijalizacije MVC-a

Kako bi se ostvarila interakcija komponenata MVC arhitekture potrebno je napraviti sljedeće korake:

1. Prvi korak inicijalizacije MVC arhitekture je generiranje modela i sama inicijalizacija unutarnjeg stanja modela.
2. Nakon procesa inicijalizacije modela generira se komponenta pregleda kojemu se prosljeđuje referenca modela koji ga je generirao.
3. Tijekom procesa inicijalizacije komponente pregleda, komponenta se prijavljuje na komponentu modela kako bi se mogla informirati o tome da je došlo do promjene podataka unutar modela i da je potrebno ažurirati pregled.
4. Nakon što se komponenta pregleda registrirala na slušače promjena unutar modela, pregled pokreće generiranje upravitelja kojem prosljeđuje reference komponente pregleda i modela.
5. U inicijalizaciji, upravitelja, upravitelj se može, ali ne mora registrirati za promjene modela. To ovisi o tome da li mijenja li upravitelj unutarnje stanje ili ne.
6. Posljednji korak u inicijalizaciji MVC arhitekture je povezivanje komponente pregleda i upravitelja radi osluškivanja i prosljeđivanja korisničkih akcija.

4.2. Primjer interakcije unutar MVC-a

Uzmimo za primjer jednostavnu aplikaciju koja sadrži listu podataka i gumb koji se koristi za osvježavanje podataka koji se nalaze unutar liste.

Komponenta pregleda se koristi kako bi se korisniku prikazala lista i gumb koji omogućuje osvježavanje liste na kojem je zakačen slušač kako bi se mogla ostvariti interakcija korisnika aplikacija s gumbom. Nakon što korisnik napravi interakciju s gumbom za osvježavanje komponenta pregleda, slušač akcija javlja pregledu akciju, te ju pregled prosljeđuje upravitelju.

U trenutku kada upravitelj zaprimi informaciju o korisničkoj akciji započinje se izvršavati potrebne metoda koje su namijenjene za zaprimljenu akciju. Unutar tih metoda upravitelj poziva jednu od izloženih metoda modela kojima se pokreči zahtijeva za ažuriranjem podataka koji se nalaze unutar modela.

Po zaprimanju zahtjeva od strane upravitelje, model izvršava počinje s izvršavanjem potrebnih unutarnjih metoda kako bi se dohvatili i osvježili potrebni podaci. Po završetku

dohvaćanja novi podataka, model informira sve slušače koji su se prijavili za slušanje izmjena, da je došlo do izmjene podataka.

Komponenta pregleda po završetku obavještavanja modela, preuzima ažurirane podatke iz modela i ažurira listu s novim podacima koji se nalaze unutar modela.

4.3. Primjena MVC-a unutar Android aplikacije

Za implementaciju i primjenu MVC arhitekture unutar prirodne Android aplikacije mogu se iskoristiti već postojeće komponente koje su dostupne bez potrebe dodavanja vanjskih biblioteka.

U poglavlju o arhitekturi mobilne platforme Android su opisane komponente aktivnosti i fragmenta. Jedan od načina na koji se može implementirati MVC arhitektura unutar prirodne Android aplikacije je takva da aktivnost predstavlja komponentu upravitelja, fragment predstavlja komponentu pregleda, dok se model može realizirati uz pomoć obične klase.

Radi načina izvođenja prirodne mobilne aplikacije Android i elemenata koji su prethodno opisani, potrebno je malo prilagoditi način implementacije unutar aplikacije. Prva komponenta koja se kreira je komponenta pregleda, odnosno u ovom slučaju je to aktivnost, koja nakon što se inicijalizira kreira komponente modela i pregleda. Glavna izmjena se odnosi na redoslijed inicijalizacije komponenata MVC arhitekture, dok veza između njih ostaje neizmijenjena. Proces inicijalizacije nije više inicijalizacija modela, komponente pregleda te upravitelja, već će se upravitelj prvi generirati i on će biti zadužen za inicijalizaciju modela i pregleda. Po završetku inicijalizacije pregleda, pregled odnosno fragment se registrirati model kako bi osluškivao izmjene podataka unutar modela.

4.4. Primjena MVC-a unutar iOS aplikacije

Jedan od načina implementacije MVC arhitekture unutar prirodne iOS aplikacija, a da nema potrebe dodavanjem vanjskih biblioteka je taj da ViewController predstavljana komponentu upravitelja, dok je komponenta pregleda predstavljena kao UIView i komponentu modela predstavlja osnovna klasa.

S navedenim komponentama MVC arhitektura, arhitektura se inicijalizira na sljedeći način. Prvi korak je inicijalizacija ViewControllera. Nakon inicijalizacije upravitelja započinje se inicijalizacija komponenata modela i pregleda. Nakon što je završena inicijalizacija komponente pregleda, ostvaruje se potrebna ovisnost u obliku registracije pregleda na model radi informiranja o izmjeni unutarnjeg stanje podataka unutar modela.

4.5. Primjena MVC-a unutar Flutter aplikacije

Za razliku od prethodno opisanih primjena primjene MVC arhitekture unutar prirodnih Android i iOS aplikacija, Flutter aplikacija nema komponenate koje bi se mogle iskoristiti za implementaciju MVC arhitekture. Flutter koristi samo widgete radi čega je potrebno napraviti pripremu kako bi se uspješno implementirala MVC arhitektura unutar Flutter aplikacije.

Za pripremu su kreirane komponente MVCController, MVCView i MVCModel. MVCController je kreiran proširivanjem widgeta sa stanjem koji koristi MVCControllerState za svoj objekt stanja. MVCView je kreiran tako da sadrži referencu na widget koji predstavlja komponentu pregleda koji je tipa StatefulView ili StatelessView. StatefulView koristi objekt MVCState kao svoj objekt stanja. MVCModel je kreiran na tako da se koristi obična klasa unutar koje je su implementirane potrebne metode.

Uz te komponente kreirane su i neke pomoćne komponente kao što su

- MVCModelObserver kojeg je potrebno proširiti kako bi se moglo registrirati na promjene unutar modela, a to su MVCControllerState i MVCState,
- MVCEvent je komponenta koja se koristi za prijenos korisničke akcije od pregleda prema upravitelju i unutar nje se nalaze podaci o widgetu na kojemu se dogodila akcija i vrijednost, ako je potrebna za izvršavanje akcije.

```
abstract class MVCModelObserver {
        void update({int flag});
}
class MVCEvent {
        MVCEvent(this._sender, this._value);


        Widget _sender;
        dynamic _value;
        Widget getSender() {
                return _sender;
        }
        dynamic getValue() {
                return _value;
        }
}
```

Isječak kôda 1 Klasa slušača promjena MVCModela i MVCEventa

Komponenta koja predstavlja model unutar MVC arhitekture je MVCModel koji je kreiran kao apstraktna klasa. Unutar modela nalazi se kolekcija s objektima koji su se

registrirali na promjene unutar modela. Za prijavu i odjavu s modela radi slušanja promjena modela koriste se *subscribe(observer: MVCModelObserver)* i *subscribe(observer: MVCModelObserver).* Metoda *notify(flag: Int)* služi kako bi se obavijestili svi slušači, kojima se prosljeđuje zastavica koja govori koji podaci su se izmijenili.

```
abstract class MVCModel {
        var observers = [];
        void subscribe(MVCModelObserver observer) {
                observers.add(observer);
        }
        void unsubscribe(MVCModelObserver observer) {
                observers.remove(observer);
        }
        void notify({int flag}) {
                 observers.forEach((element) {
                  (element as MVCModelObserver).update(flag: flag);
                });
        }
}
```

Isječak kôda 2 Primjer komponente Modela

Komponenta MVCView predstavlja komponentu pregleda unutar MVC arhitekture. Komponenta je kreirana kao apstraktna klasa unutar koje se nalazi varijabla widgeta koji je predstavljen i StreamController koji se koristi za komunikaciju između komponente pregleda i upravitelja. Tu su i dvije pomoćne metode kojima se ostvaruje komunikacija prema upravitelju. Te metode su *addNotifier(notifier: StreamController)* koja se koristi za registraciju kanala i *notifyController(event: MVCEvent)* koja šalje korisničke akcije kroz kanal.

```
abstract class MVCView {
        Widget viewWidget;
        StreamController controllerNotifier;
        void addNotifier(StreamController notifier) {
                this.controllerNotifier = notifier;
        }
        void notifyController(MVCEvent event) {
                 controllerNotifier.add(event);
        }
}
```

Isječak kôda 3 Primjer komponente pregleda

Kako je MVCStatefulView prošireni widget sa stanjem koji se koristi kao komponenta pregleda unutar MVC arhitekturu. MVCStateufulView kreirana je kao apstraktna klasa i sadrži

referencu na komponentu modela. Tijekom kreiranja MVCStatefulViewa postavlja samog sebe unutar varijable widget koja je naslijeđena od MVCView komponente.

```
abstract class MVCStatefulView<M extends MVCModel> extends
StatefulWidget with MVCView {
  M model;
  MVCStatefulView(M model) {
    this.viewWidget = this;
    this.model = model;
  }
}
```

Isječak kôda 4 Klasa komponente pregleda sa stanjem

MVCState je komponenta koja se koristi za stanje komponente pregleda, pošto svaki widget sa stanjem mora imati vlastiti objekt stanja, a MVCStatefullView je widget sa stanjem. Klasa je kreirana kao apstraktna i unutar sebe sadrži referencu na model, koji se dodjeljuje kroz konstruktor unutar kojega se stanje registrira na promjene unutar modela.

```
abstract class MVCState<V extends MVCStatefulView, M extends
MVCModel> extends  State<V> with MVCModelObserver {
      M model;
      MVCState(M model) {
            model.subscribe(this);
            this.model = model;
       }
}
```

Isječak kôda 5 Primjer klase stanja

Komponenta upravitelj kreirana je tako da se proširuje widget sa stanjem. Prilikom kreiranja upravitelja kreira se novi kanal koji se koristi za osluškivanje korisničkih akcija koji su kreirani preko komponente pregleda. Također je koristi metoda za primanje korisničkih akcija koje je potrebno obraditi.

```
abstract class MVCController<V extends MVCView, M extends MVCModel>
      extends StatefulWidget {
      StreamController<MVCEvent> _streamController;
      V view;
      M model;
      BuildContext context;
      void create(MVCView view, MVCModel model) {
            _streamController = StreamController();
```

```
                view.addNotifier(_streamController);
                _streamController.stream.listen(eventHandler);
        }
         void init() {
                _streamController = StreamController();
                view.addNotifier(_streamController);
                _streamController.stream.listen(eventHandler);
         }
         void eventHandler(MVCEvent event);
}
```

Isječak kôda 6 Klasa komponente upravitelja

MVCControllerState je komponenta koja se koristi za objekt stanja upravitelja i kreirana je kao apstraktna klasa koja proširuje klasu stanja. Unutar metode *build()* se vraćaju widgeti koji grade grafičko sučelje unutar MVCView komponente. Unutar metoda initState(), *didUpdateWidget()* ažurira se kontekst MCVController komponente s kontekstom MVCView komponente. Unutar metode *dispose()* uništava se kontekst MVCControllera i zatvara se kanal koji se koristi za slušanje korisničkih akcija koje se kreiraju unutar MVCView komponente.

```
abstract class MVCControllerState<ExtendedController extends
MVCController>
        extends State<ExtendedController> {
        @override
        Widget build(BuildContext context) {
                return widget.view.viewWidget;
        }
        @override
        void initState() {
                super.initState();
                widget.context = context;
        }
        @override
        void didUpdateWidget(ExtendedController oldWidget) {
                super.didUpdateWidget(oldWidget);
                widget.context = oldWidget.context;
        }
```

```
        @override
        void dispose() {
                widget._streamController.close();
                widget.context = null;
                super.dispose();
        }
}
```

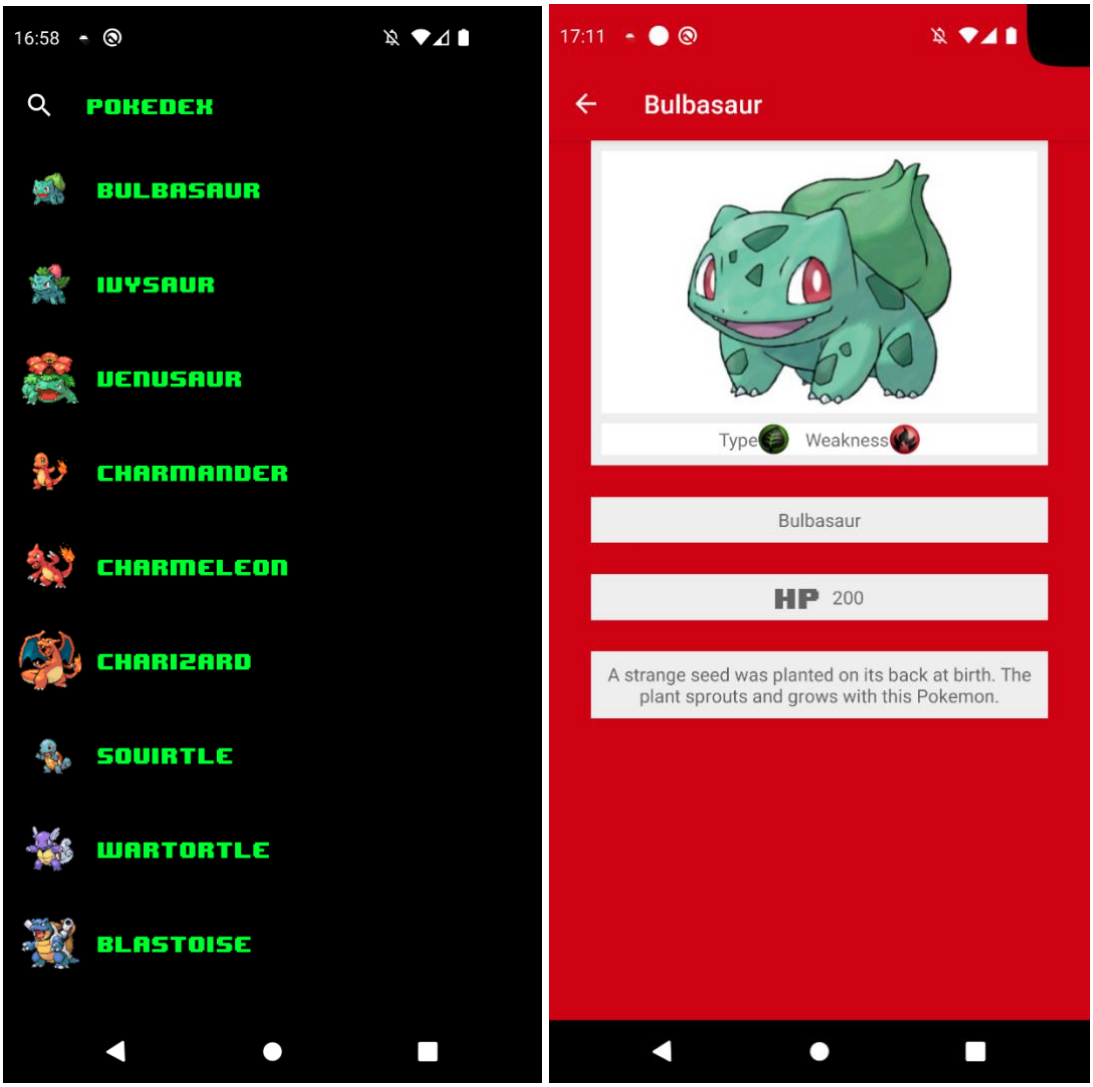
Isječak kôda 7 Klasa stanja upravitelja

Unutar ovog poglavlja su pojašnjene komponente pomoću kojih se ostvaruje jedan od mnogo načina kojima je moguće implementirati MVC arhitekturu unutar aplikacija. U sljedećem poglavlju će se navedeni primjeri implementacije MVC arhitekture primijeniti unutar prirodnih aplikacija Android i iOS, te unutar Flutter aplikacije.

5. Aplikacija

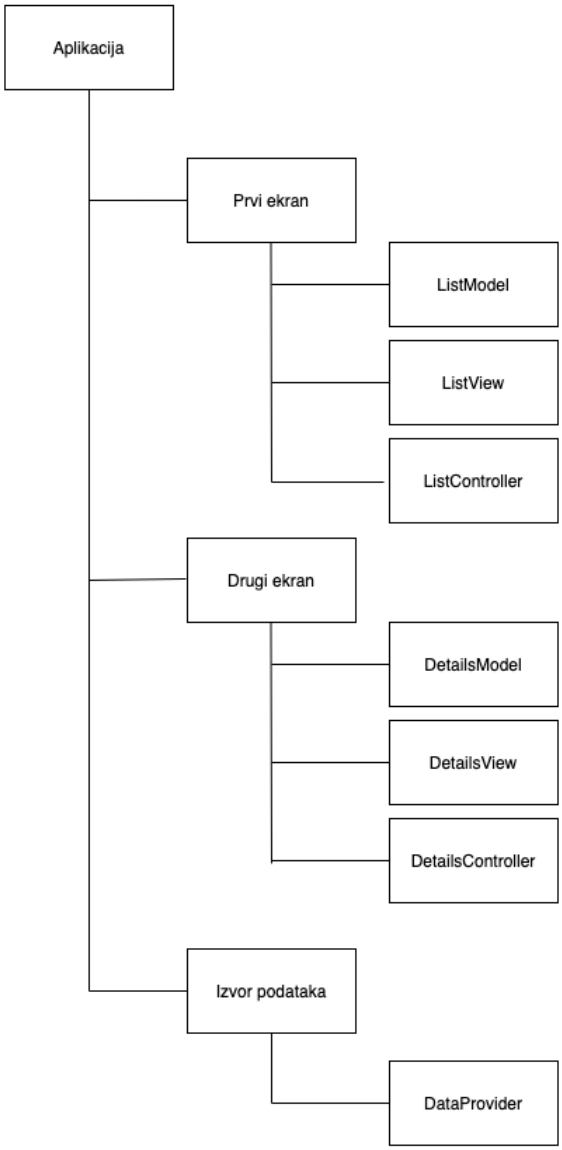
Za usporedbu procesa razvoja aplikacija na platformama koje su prethodno opisane, bit će napravljena mobilna aplikacija. Aplikacija će biti prilagođena na svakoj platformi koja je opisana prethodno u radu, stoga je aplikacija napisana u tri projekta, kao prirodna Android i prirodna iOS aplikacija, te Flutter aplikacija. Projekti se razlikuju po elementima programskog jezika u kojemu je pisana, dok su arhitektura, podaci i ostali elementi zajednički.

Izrađena aplikacija će biti vrlo jednostavna. Funkcionalnosti aplikacije su prikaz liste s podacima, pretraživanje liste, mogućnost osvježavanja liste i prikaz detalja o odabranome podatku iz liste. Funkcionalnosti su podijeljeni na dva ekrana (slika 10), gdje je prvi ekran lista popunjena podacima koji se dohvaćaju preko API poziva i sadrži filter za filtriranje liste. Drugi ekran je ekran s detaljima i služi za statički prikaz podataka o prethodno odabranom elementu unutar liste.



Slika 10 Ekrani aplikacije, prvi ekran lijevo i drugi ekran desno

Za realizaciju aplikacije korištena je MVC arhitektura, koja je implementirana prema koracima koji su opisani u poglavlju o primjeni MVC arhitekture. Kako je aplikacija sačinjena od dva ekrana, aplikacija se može podijeljena u dvije cjeline koje sadrže komponente MVC arhitekture i jedne cjeline unutar koje se nalazi izvor podataka. Struktura aplikacije je prikazna slikom 12. Prva cjelina označava prvi ekran sačinjena od ListModel, ListView i ListController, dok je druga cjelina sačinjena od DetailsModel, DetailsView i DetailsController klasa. Treća cjelina se ne odnosi na komponente arhitekture nego se odnosi na izvor podataka.



Slika 11 Skica strukture aplikacije

Kako arhitektura označava strukturu i interakciju između elemenata, a za njihovo izvođenje potrebni su podaci. Za to je korištena statična klasa DataProvider unutar koje se nalaze metode za pozivanje vanjskih pristupnih točaka. Interakcija s arhitekturom se realizira

tako da komponenta modela poziva metode iz DataProvidera te tako osigurava podatke unutar aplikacije.

Za dohvat podataka korištene su dvije pristupne točke, a to su “*pokeapi.co*” i “*courses.cs.washington.edu*”. Korištene su dvije različite pristupne točke zato što nije bilo moguće pronaći pristupnu točku sa svim željenim podacima.

Pristupna točka “*pokeapi*.co” koristi se za dohvat liste s podacima koji su sortirani po ID-u, dok su na ostalim pristupnim točkama podaci bili poredani po abecedi. Kao odgovor pristupne točke dobiva se sljedeći odgovor:

```
{“name”: ”string”,
”url”: ”string”}
```

Isječak kôda 8 Struktura odgovora pristupne točke pokeapi.co

Radi povezivanja odgovora dviju pristupnih podataka, potrebno je obraditi podatke te iz atributa *“url”* dohvatiti atribut “*id*”. Atribut “*name*” se koristi za polja “*name*” i “*codeName*”. Atribut “*name*” se dalje prilagođava kako bi atribut imao napisano prvo slovo velikim slovite te kako bi se zamijenili znakovi “-” sa znakom razmaka.

Pristupna točka “*courses.cs.washington.edu*” se koristi za dohvat detalja o podatcima i koristi se za dohvat ikona i slika. Odgovor koji se dobije od strane pristupne točke je sljedeći:

```
{"name":"string",
 "shortname":"string",
"hp":int,
"info":{
"id":int,
"type":"string",
"weakness":"string",
"description":"string"},
"images":{
"photo":"string",
"typeIcon":"string",
"weaknessIcon":"string"},
"moves":
```

```
[{
        "name":"string",
        "dp":int,
        "type":"string"}
]}
```

Isječak kôda 9 Struktura odgovora pristupne točke courses.cs.washington.edu

Iz navedenog odgovora se preuzimaju svi podaci osim objekta *moves*. Kako je dobiveni objekt složen bilo je potrebno izraditi pomoćne objekte za atribute *info* i *images*. Podaci koji se nalaze unutar Info objekta nije bilo potrebno dodatno prilagođavati, dok je prilikom obrade objekta *image* bilo potrebno napraviti prilagodbu, tako da se za svaku ikonu i sliku unutar objekta pridodan prefiks kako bi putanja do slika bila potpuna.

Model dohvaća podatke uz pomoć metoda koje se nalaze unutar DataProvider klase. Nakon dobivenog odgovora, model provjerava odgovor te ako odgovor nije dobar, javlja da je potrebno ispisati poruku o grešci. U slučaju da su podaci uredu i da se nije dogodila nikakva greška započinje se s pasiranjem podataka koji se nakon pasiranja spremaju u lokalnu varijablu unutar modela i model okida metodu *notify(flag: int)* uz pomoć koje se obavještavaju svi objekti koji pretplaćeni na model.

Nakon što model okine metodu *notify(flag: int)*, unutar svih pretplaćenih komponenata na taj model poziva se *update(flag: int)*. Argument f*lag* označava koji podaci su se izmijenili unutar modela i pomoću njega pregled odrađuje potrebnu akciju.

Zadnji element koji je važno izdvojiti je komponenta upravitelja. Upravitelj je zadužen za kontrolu interakcije između ostalih komponenata arhitekture. Upravitelj se po potrebi može ali i ne mora registrirati na promjene unutar modela. Veza prema komponenti pregleda ostvaruje se uz pomoć metode *handleEvent()* gdje pregled okida metodu sa zastavicom koja predstavlja korisničku akciju i podatkom koji je potreban kako bi se akcija mogla izvršiti akcija. Upravitelj je isto tako zadužen za pozivanje prema novim upraviteljima koji se kreiraju.

Kako je rečeno, aplikacija se sastoji od dva ekrana. Prvi ekran je lista s podacima i tražilicom koja omogućuje filtriranje liste prema imenu. Prvi ekran je u crnoj boji, te ima tekst koji je pobojan u zeleno, dok je drugi ekran cijeli pobojan crvenom bojom. Podaci koji se koriste unutar tog ekrana detalji prethodno odabranog entiteta.

Unutar prvog ekrana također postoji opcija osvježavanja jednostavnim povlačenjem liste (*eng. Pull-to-refresh*) . Unutar radne trake na vrhu ekrana se nalazi jednostavan naslov ekrana i mala ikona koja označava filter polje. Pritiskom na ikonu za pretraživanje navedeni

naslov ekrana se pretvara u polje za unos teksta. Pretraživanje se odvija tako da se filtrira lista i prikazuju se samo oni podaci koji se podudaraju s unosom unutar polja za unos teksta. Unutar radne trake u desnome dijelu bit će ispisano trenutno stanje baterije kako bi se moglo usporediti dohvaćanje podataka koji su dostupni na razini platforme.

5.1. Android

Aplikacija koja je izrađena za Android napisana je u Kotlin programskom jeziku. Prilikom izrade su korištene osnovne komponente kao što su aktivnost i fragment, također su korištene neke od vanjskih biblioteka.

5.1.1. Priprema aplikacije

Radi lakše izrade aplikacije iskorištene su neke od vanjskih biblioteka. Biblioteke koje su korištene su usko vezane za dohvat podataka i njihovu obradu. Vanjske biblioteke koje su korištenje za izradu Android mobilne aplikacije su:

- OkHttp, kao jednostavni HTTP klijent unutar aplikacije za uspostavu API poziva,
- Retrofit, biblioteka koja olakšava pozive prema API metodama i osigurava pripremu podataka,
- GSON, biblioteka koja se koristi za pripremu JSON-a u POJO objekt koji se dalje koristi unutar same aplikacije, te
- Glide, je biblioteka koja se koristi za prikaz slika koji se nalaze na udaljenoj adresi.

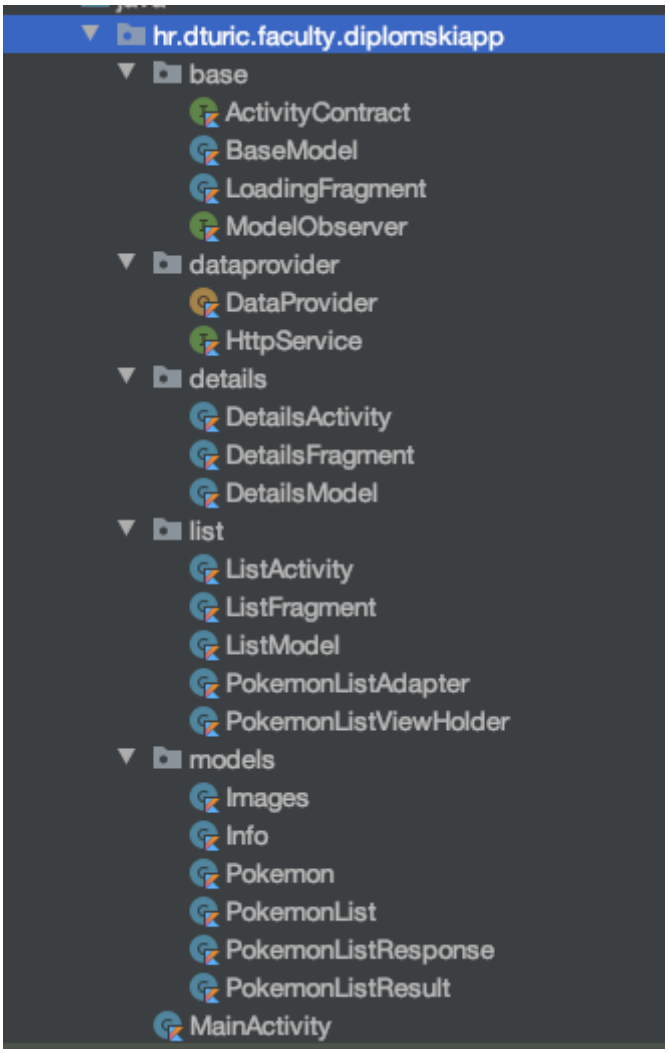
```
// Glide
implementation 'com.github.bumptech.glide:glide:4.11.0'
annotationProcessor 'com.github.bumptech.glide:compiler:4.11.0'

// GSON
implementation 'com.google.code.gson:gson:2.8.6'

// Retrofit & OkHttp
implementation 'com.squareup.retrofit2:retrofit:2.7.2'
implementation 'com.squareup.retrofit2:converter-gson:2.7.2'
implementation 'com.squareup.okhttp3:okhttp:4.4.0'
implementation 'androidx.recyclerview:recyclerview:1.1.0'
```

Slika 12 Biblioteke korištene prikom izrade prirodne Android aplikacije

Za korištenje vanjskih biblioteka koristi se Gradle sustav skripti. Proces korištenja vanjske provodi se na sljedeći način. Prvo je potrebno otvoriti datoteku build.gradle (:app) i u unutar potkategorije dependencies dodati željeni dependency koji označava željenu biblioteku i koju verziju biblioteke koja se želi implementirati unutar projekta. Nakon što se je dodao dependency potrebno je pokrenuti sinkronizaciju projekta. Tijekom sinkronizacije projekta preuzima se željena biblioteka.

5.1.2. Izrada aplikacije

Kao što je prethodno opisano, aplikacija je izrađena po primjeru MVC arhitekture tako da je prilagođena Android komponentama, tako je aktivnost preuzela ulogu komponente upravitelja, Fragment je iskorišten kao komponenta pregleda, dok je komponenta modela izrađena bez nasljeđivanja komponenata platforme. Slika prikazuje strukturu unutar prirodne Android aplikacije.



Slika 13 Struktura prirodne Android aplikacije

Struktura projekta aplikacije je podijeljena u pet cjelina. Prva cjelina sadrži osnovne klase koje se iznova iskorištavaju. Druga cjelina je zadužena za dohvaćanje podataka. DataProvider je objekt koji se ponaša kao statična klasa i osigurava potrebne metode za dohvat podataka, dok je HttpService pomoćno sučelje koje se koristi za rad Retrofit biblioteke. Treća cjelina je cjelina koja je sačinjena od klasa koje se koriste za realizaciju drugog ekrana, dok je četvrta zaslužna za realizaciju prvog ekrana. Unutar nje se također nalaze pomoćne klase koje se koriste za popunjavanje liste (PokemonListAdapter) i klase koja popunjava svaki

element liste (PokemonListViewHolder). Peta cjelina je cjelina koja sadrži modele za podatke koji se koriste unutar aplikacije.

Implementacija komponenti MVC arhitekture se izvodi tako dase prilikom kreiranja aktivnosti, unutar *onCreate()* metode instanciju sve važne komponente, kao što su komponenta modela i Fragment kao komponenta pregleda. Prilikom instanciranja fragmenta prosljeđuje se referenca na objekta modela. Kada je Fragment instanciran aktivnost uz pomoć FragmentManagera povezuje fragment sa svojim životnim ciklusom.

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_list)

        model = ListModel()
        view = ListFragment.newInstance(model!!)
        model?.subscribe(this)
        supportFragmentManager.beginTransaction().also {
            it.replace(android.R.id.content, view!!)
            it.commit()
        }
        prepareToolbar()
    }
```

Isječak kôda 10 Isječak kôda kreiranja upravitelja

Nakon što se inicijalizira fragment, unutar događaja spajanja fragmenta na životni ciklus roditeljske aktivnosti, fragment se registrira na promjene koje se događaju unutar modela.

```
override fun onAttach(context: Context) {
        super.onAttach(context)

        this.model?.subscribe(this)

        if (context is ActivityContract) {
            this.contract = context
        }
}
```

Isječak kôda 11 Isječak kôda kreiranje komponente pregleda

Za prikaz nove aktivnosti koristi se metoda *startActivity(intent: Intent).* Kako bi se kreirao objekt Intent potrebno je proslijediti kontekst (eng. Context) aktivnosti, te isto tako java class objekt aktivnosti koja se želi prikazati. Za prosljeđivanje podataka između dviju aktivnosti koriste se sam Intent objekt.

```
override fun update(flag: Int) {
        if (flag == ListModel.POKEMONS) {
            val pokemons = model?.pokemons
            adapter.changeData(pokemons)
        }
    }
```

Isječak kôda 12 Isječak kôda prikazivanja novog ekrana

Na slici je prikazana metoda *update(flag: int)* koja se koristi kada model javlja da je došlo do izmjene podataka koji se nalaze unutar njega. Parametar *flag* daje informaciju o tome što se izmijenilo unutar modela, uz pomoć koje slušatelj zna koju akciju je potrebno pokrenuti. U ovom slučaju je to akcija kreiranja Intenta kako bi se prezentirao ekran s dobivenim podacima. Prilikom kreiranja Intenta prosljeđuje se aktivnost koja se želi prezentirati, dok se podaci prosljeđuju kroz ugrađenu metodu *putExtra().* Metoda koristi prima jedinstvenu oznaku podatka koja se koristi za izvlačenje podataka i sam podatak koji se prosljeđuje drugoj aktivnosti.

Za dohvat stanja baterije korišten je isječak kôda 13. Prilikom dohvata podatka o trenutnom stanju baterije potrebno je provjeriti koja je trenutna verzija Android platforme na kojoj se izvršava aplikacija. Ako je verzija androida manja od 21 stanje baterije se dohvaća uz pomoć intenta, a ako je veća ili jednaka verziji 21 tada se stanje baterije dohvaća pomoću BatteryManagera.

```
private fun getBatteryLevel(): Int {
        val batteryLevel: Int
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            val batteryManager = requireContext()
.getSystemService(Context.BATTERY_SERVICE) as BatteryManager
            batteryLevel =
batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
        } else {
            val intent = ContextWrapper(requireContext()
.applicationContext)
.registerReceiver(
```

```
                    null, IntentFilter(
                        Intent.ACTION_BATTERY_CHANGED
                    )
                )
                batteryLevel =
                    intent!!.getIntExtra(BatteryManager.EXTRA_LEVEL, -1)
* 100 / intent.getIntExtra(
                        BatteryManager.EXTRA_SCALE, -1
                    )
            }
            return batteryLevel
        }
```

Isječak kôda 13 Isječak kôda dohvaćanja stanja baterije

5.1.3. Grafičko sučelje

Za izradu grafičkog sučelja se koriste XML datoteke koje sadrže sve komponente koje se nalaze unutar grafičkog sučelja i njihov raspored na zaslonu. Uređivanje pojedinog ekrana moguće ostvariti na dva načina. Prvi način je direktno uređivanje XML datoteke, dok je drugi realiziran uz pomoć jednostavnog povlačenja komponenti. Svaki element koji se nalazi unutar grafičkog sučelja kojemu se želi pristupiti unutar programskog kôda aplikacije mora imati svoju jedinstvenu oznaku.

Za izradu liste korišten je RecyclerView komponenta koja se popunjava uz pomoć ListAdaptera čiji je isječak kôda prikazan na slici 14. Za svaki element liste koristi ListViewHolder koji je zadužen za upravljanje grafičkim elementima unutar elementa liste. Prilikom kreiranja adaptera potrebno je prepisati (eng. Override) tri metode, *onCreateViewHolder*() koja se koristi za povezivanje grafičkih elemenata s objektom ViewHolder koji se koristi za upravljanje grafičkim elementima unutar liste, *getItemCount*() se koristi kako bi adapter liste znao koliko je elemenata unutar liste koje je potrebno prikazati i *onBindViewHolder*() je metoda koja se poziva kada je potrebno element liste popuniti s potrebnim podacima.

```
class PokemonListAdapter(
    private var pokemons: ArrayList<PokemonList>,
    private val listener: OnPokemonSelectedListener) :
    RecyclerView.Adapter<PokemonListViewHolder>() {
```

```
        private var filteredData = ArrayList<PokemonList>()

        override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): PokemonListViewHolder {

            val view =

                LayoutInflater.from(parent.context)
.inflate(R.layout.pokemon_list_item, parent, false)

            return PokemonListViewHolder(view)

        }

        override fun getItemCount() = filteredData.size

        override fun onBindViewHolder(holder: PokemonListViewHolder,
position: Int) {

            holder.bind(filteredData[position], listener)

        }

        fun changeData(pokemons: java.util.ArrayList<PokemonList>?) {

            this.filteredData = pokemons ?: ArrayList()

            this.pokemons = pokemons ?: ArrayList()

            notifyDataSetChanged()

        }

        fun filter(filter: String) {

            filteredData = if (filter == "") pokemons

            else ArrayList(pokemons.filter { pokemonList ->

                pokemonList.name.contains(filter, true)

            })

            notifyDataSetChanged()

        }

        interface OnPokemonSelectedListener {

            fun onPokemonSelected(pokemon: PokemonList)

        }

    }
```

Isječak kôda 14 kôd adaptera liste

Isječak kôda 15 prikazuje XML zapis jednog elementa liste. Svaki element liste se nalazi unutar LinearLayouta koji svoje elemente raspoređuje horizontalno, odnosno prikazuje ih s lijeva na desno. Unutar LinearLayouta nalaze se dva elementa, a to su AppCompatImageView i TextView. AppCompatImageView koristi se za prikazivanje slike, dimenzije su mu 48x48dp i ima margine od 12dpa. Drugi element je TextView koji se koristi za ispis teksta i on ima neodređenu visinu koja ovisi o količini teksta koji se nalazi unutar njega. Sam tekst je centriran vertikalno unutar roditelja, zelene je boje i ima dodijeljen font.

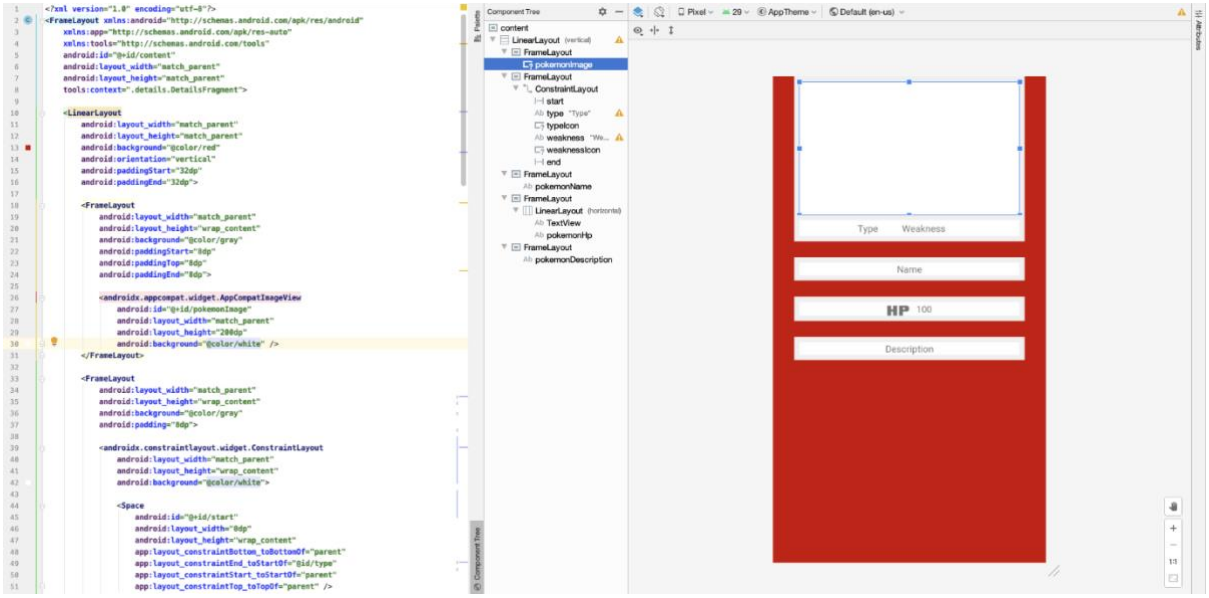
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <androidx.appcompat.widget.AppCompatImageView
        android:id="@+id/pokemonIcon"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:layout_marginTop="24dp"
        android:layout_marginStart="24dp"
        android:layout_marginBottom="24dp" />
    <TextView
        android:id="@+id/pokemonName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_marginEnd="12dp"
        android:textColor="@color/green"
        android:fontFamily="@font/bit_8_hud"
        tools:text="Pokemon name" />
</LinearLayout>
```

Isječak kôda 15 XML opis elementa liste

Dodatni elementi koji su korišteni za izradu prvog ekrana su Toolbar i SwipeRefreshLayout. Toolbar element se koristi za dodavanje polja za pretraživanje i unutar njega se nalazi naslov ekrana, te pomoćni gumbi za navigaciju kroz aplikaciju. SwipeRefreshLayout je element koji omogućuje implementiranje opcije osvježavanje liste pomoću jednostavnog povlačenja njegovoga sadržaja.

Drugi ekran je realiziran uz pomoć DetailsActivity, DetailsFragment i DetailsModel klasa. Za izradu sučelja korištene su komponente kao što su AppCompatImageView, TextEdit i FrameLayout. Na slici 14 se može vidjeti uređivač grafičkog sučelja za Android prirodnu aplikaciju. S lijeve strane se nalazi XML opis sučelja, dok je s desne strane prikazano grafičko sučelje koje je opisano XML zapisom. Na sredini slike se može vidjeti stablo koje reprezentira hijerarhiju komponenti unutar grafičkog sučelja ekrana. Elementi su sadržani unutar

LinearLayout koji svoje elemente iscrtava vertikalno, odnosno od gore prema dolje. Svaka od sekcija na ekranu je smještena unutar FrameLayouta, osim sekcije koja se nalazi ispod sekcije sa slikom, ta sekcija je napravljena uz pomoć ConstraintLayouta. Za ispis teksta korišten je TextView element.



Slika 14 Kreiranje drugog ekrana aplikacije

Unutar ovog poglavlja su opisani koraci i struktura prirodne Android aplikacije, dok će u nastavku biti opisani koraci izrade i struktura prirodne iOS aplikacije i nakon toga Flutter aplikacije.

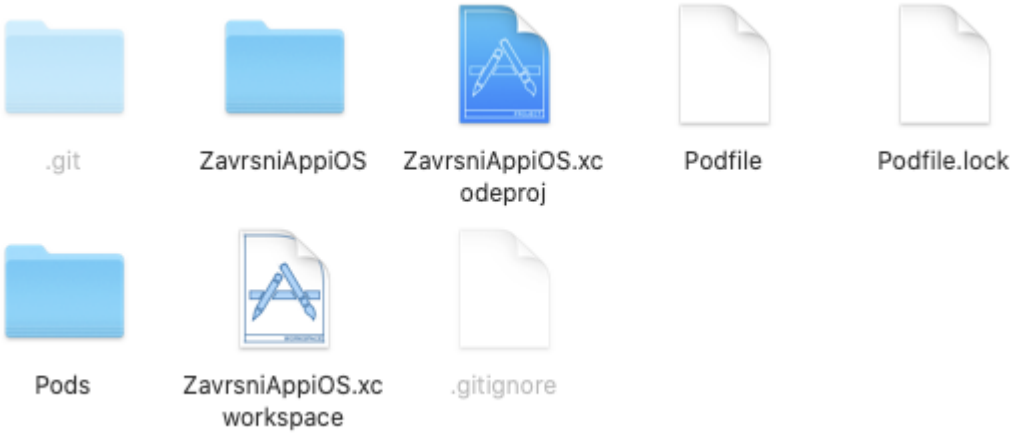
5.2. IOS

Aplikacija pisana za iOS platformu je napisana u Swift programskome jeziku. Prilikom izrade korištene su standardne komponente kao što su Storyboard, ViewControlleri i ostale komponente navigacije, i vanjske biblioteke.

5.2.1. Priprema aplikacije

Prilikom izrade aplikacije korištene su vanjske biblioteke koje olakšavaju dohvaćanje i obradu podataka dohvaćenih s vanjskih izvora i korištena je biblioteka za ispis poruke o grešci. Vanjske biblioteke koje su korištene u izradi aplikacije su:

- Alamofire, biblioteka koja olakšava upravljanje API pozivima,
- SwiftyJSON, je biblioteka koja služi za jednostavnije pasiranje JSON objekta,
- Kingfisher, koji se koristi za asinkrono učitavanje slika dohvaćenih preko URL-a, te
- TTGSnackbar, koji se koristi za prikaz poruka o greški.



Slika 15 Primjer projekta nakon dodavanja vanjskih biblioteka

```
target 'ZavrsniAppiOS' do
  # Comment the next line if you d
  use_frameworks!

  # Pods for ZavrsniAppiOS

pod 'Alamofire', '~> 5.2'
pod 'SwiftyJSON', '~> 4.0'
pod 'Kingfisher', '~> 5.0'
pod 'TTGSnackbar', '~> 1.10.3'

end
```

Slika 16 Vanjske biblioteke korištene prilikom izrade iOS aplikacije

```
Danijels-MBP:ZavrsniAppiOS danijelt$ pod install
Analyzing dependencies
Downloading dependencies
Generating Pods project
Integrating client project
Pod installation complete! There are 4 dependencies from the Podfile and 4 total pods installed.

[!] Automatically assigning platform `iOS` with version `13.2` on target `ZavrsniAppiOS` because no platform was specified. Pl
ease specify a platform for this target in your Podfile. See `https://guides.cocoapods.org/syntax/podfile.html#platform`.
Danijels-MBP:ZavrsniAppiOS danijelt$
```

Slika 17 Primjer pod install komande za instalaciju vanjskih biblioteka

Dodavanje vanjskih biblioteka unutar iOS projekta je malo složenije. Sustav koji se koristi za upravljanje vanjskim bibliotekama se zove CocoaPod. Kada želimo dodati biblioteku unutar projekta potrebno je unutar korijenskog direktorija otvoriti komandnu liniju i pozvati sljedeću komandu *pod init* koji za tu akciju priprema projekt iz .xcodeproject u .xcworkspace koji se u pozadini ponaša kao novi projekt i unutar sebe sadrži .xcodeproject kao biblioteku (Slika 15). Nakon što se kreirala *pod* datoteka potrebno ju je urediti tako da se dodaju unutar pod datoteke željene biblioteke, i to se dodaje unutar Target “ime_projekta” do (Slika 16). Nakon dodavanja *pod* referenci na biblioteku potrebno je spremiti izmjene i ponovo pokrenuti komandnu liniju i potrebno je unijeti sljedeću komandu *pod install* (Slika 17) koja služi za dodavanja biblioteke unutar .xcworkspace projekta.

5.2.2. Izrada aplikacije

Kako su sve tri aplikacije zamišljene da imaju što sličniju arhitekturu kako bi se lakše usporedile njihove izvedbe. Izvedba iOS aplikacija je prilagođena tako da se ViewController koristi kao komponenta upravitelja, UIView je komponenta pregleda, dok se Model izradio uz pomoć obične klase, unutar koje su implementirane pomoćne metode koje se koriste za obavještavanje komponenata arhitekture koje su se prethodno registrirale na promjene unutar Modela.

Implementacija MVC arhitekture se odvija na sljedeći način. Nakon okidanja događaja *viewDidLoad()* unutar ViewControllera (Isječak kôda 16), započinje povezivanje komponenta upravitelja s ostalim komponentama koje tvore MVC arhitekturu. Upravitelj je zadužen za kreiranje modela, dok je komponenta pregleda kreirana direktnom vezom unutar storyboarda. Nakon što je kreiran model, nad komponentom pregleda se poziva metoda *prepare()* unutar koje se prosljeđuju model i veza koja se koristi za komunikaciju između komponente upravitelja i pregleda. Pošto upravitelj ima potrebu za slušanje promjena unutar modela, upravitelj se također prijavljuje na promjene unutar modela.

```
override func viewDidLoad() {
        super.viewDidLoad()
        localModel = ListModel()
        localView.prepare(model: localModel!, contract: self)
        localModel?.subscribe(modelProtocol: self)
        localModel?.fetchPokemons()
        prepareSearch()
}
```

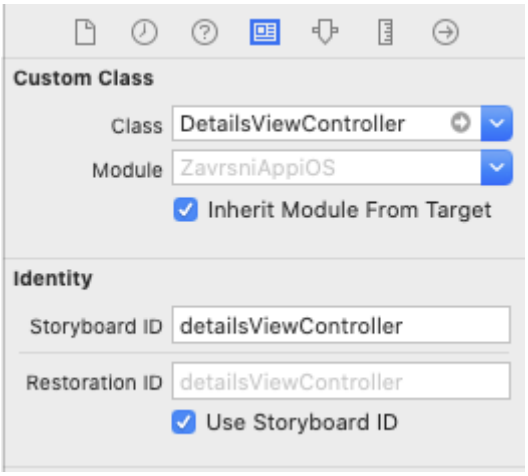
Isječak kôda 16 Isječak kôda kreiranja upravitelja

Metoda *prepare*(), unutar komponente pregleda je prikazana isječkom kôda 17. Unutar metode se spremaju reference na model i na upravitelja kako bi se mogla uspostaviti komunikacija prema njima. Na kraju pripreme, komponenta pregleda se registrira na promjene unutar modela, kako bi mogla dobiti informaciju da je došlo do izmjene stanja podataka unutar modela.

```
func prepare(model: ListModel, contract: ListContract) {
        self.model = model
        self.contract = contract
        model.subscribe(modelProtocol: self)
}
```

Isječak kôda 17 Isječak kôda kreiranja komponente pregleda

Proces prezentiranja novog ekrana odvija se na sljedeći način. Unutar uređivača grafičkog sučelja odabirom na željeni ViewController koji se želi prezentirati, dodjeljuje mu se klasa koja predstavlja taj ViewController (slika 18). Uz klasu ViewControllera potrebno je dodijeliti i jedinstvenu oznaku uz pomoć koje će se ViewController kreirati unutar kôda.



Slika 15 Opcije ViewControllera unutar uređivača grafičkog sučelja

Kada su napravljene sve potrebne pripreme unutar uređivača grafičkog sučelja potrebno je napraviti sljedeće korake (slika 18). Prvi korak je dohvaćanje instance storyboarda, zatim uz pomoć metode *instantiateViewController()* i prosljeđivanjem jedinstven oznake željenog ViewControllera kreira se njegova instanca. Nakon što se je ViewController kreirao sljedeći korak je prezentiranje istoga. Prezentiranje se odvija tako da se pozove upravitelj navigacijom (*eng. Navigation Controller*) koji sadrži metodu *show()* kojoj se prosljeđuje ViewController koji se želi prezentirati i referenca objekta ViewControllera koji se pozvao akciju.

```
func update(flag: Int) {

        if (flag == ListModel.POKEMON) {

            let pok = localModel?._pokemon

            let vc =
self.storyboard?.instantiateViewController(withIdentifier:
"detailsViewController") as! DetailsViewController

            self.navigationController?.show(vc, sender: nil)

            vc.prepare(pokemon: pok!)

            showLoader()

        }

}
```

Isječak kôda 18 Isječak kôda prikazivanja novog ekrana

Isječak kôda 18 prikazuje metodu *update(flag: int)* koja se poziva u trenutku kada model obavijesti da je došlo do izmjene podataka. Isječak koda prikazuje slučaj kada je potrebno prikazati ekran s detaljima odabranog entiteta s liste.

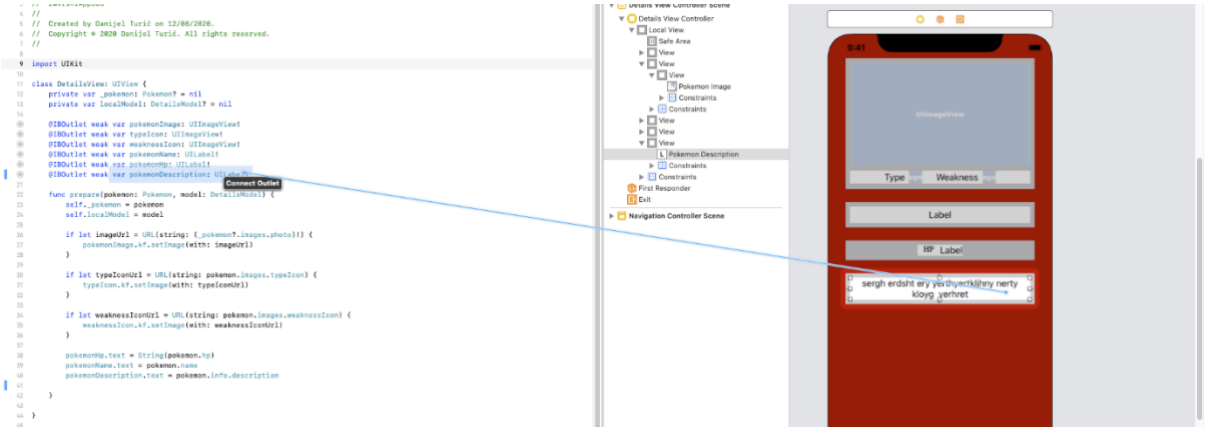
Dio kôda pomoću kojeg se dohvaća trenutno stanje baterije uređaja unutar iOS aplikacije prikazan je isječkom kôda 19. Za dohvaćanje trenutnog stanja baterije uređaja potrebno je dozvoliti praćenje stanja baterije. Nakon što je dozvoljeno čitanje stanja baterije, provjerava se može li se očitati stanje ili ne može. Ako je nemoguće očitati stanje baterije, vraća se vrijednost –1 a u suprotnom se vraća trenutno stanje baterije.

```
private func receiveBatteryLevel() -> Int {
      let device = UIDevice.current
      device.isBatteryMonitoringEnabled = true
      if device.batteryState == UIDevice.BatteryState.unknown {
        return -1
      } else {
        return Int(device.batteryLevel * 100)
      }
}
```

Isječak kôda 19 Isječak kôda za dohvaćanje stanja baterije

5.2.3. Grafičko sučelje

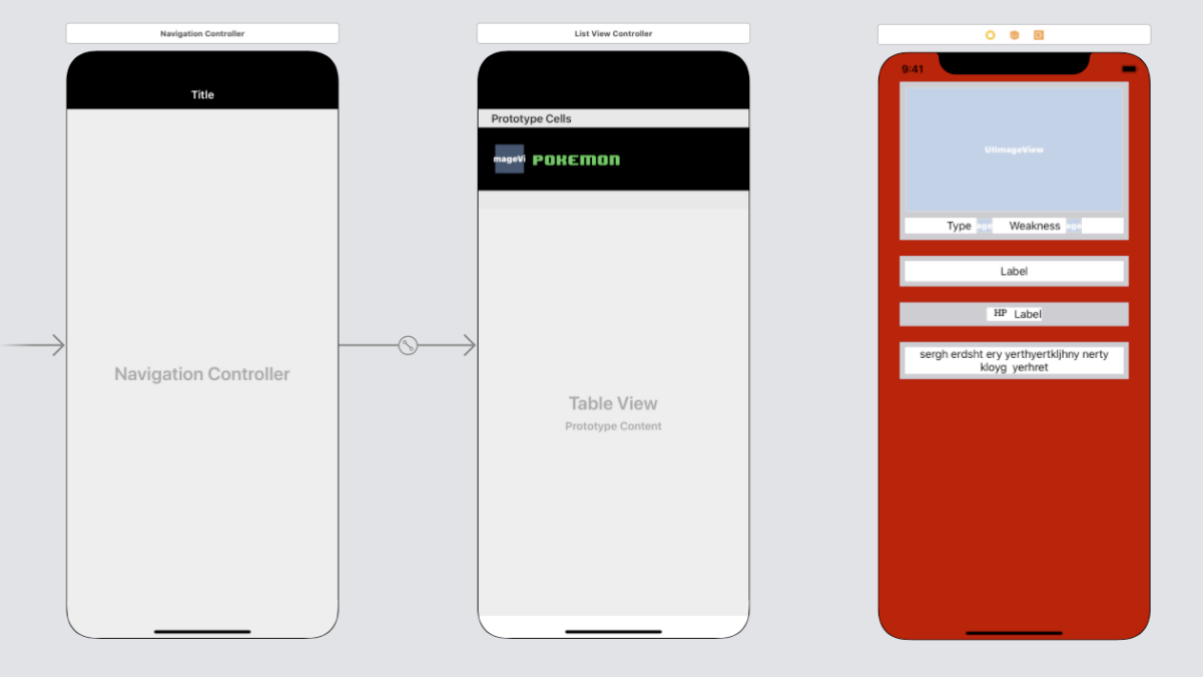
Aplikacija je izrađena uz pomoć Storyborda. Storyboard je komponenta koja omogućava vizualno slaganje grafičkih komponenata aplikacije, tako da se jednostavnim akcijama povlačenja elemenata kreira korisničko sučelje za određeni ViewController. Kada su komponente posložene u željenom rasporedu, koriste se Constraint komponente koje služe kako bi se posložene komponente što sličnije prikazivale na raznim uređajima i ekranima. Unutar sučelja kojim se uređuju Stroyboard komponente, postoje opcije kojima se raznim komponentama korisničkog sučelja dodjeljuju razni atributi kao što su ID, korespondirajuća klasa, boja i ostale opcije kojima se također može pristupiti i preko programskog načina.



Slika 16 Primjer povezivanja elemenata grafičkog sučelja s kôdom

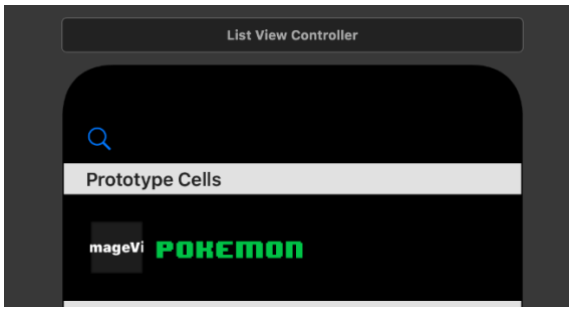
Kreiranje veze među elementima ekrana i programskog kôda se ostvaruje jednostavnim povlačenjem željene komponente prema dijelu programskog kôda koji je povezan s trenutnim ekranom (slika 19).

Na slici 20 su prikazana tri ViewControllera od kojih je izrađena prirodna iOS aplikacija. Prvi ViewController se koristi radi navigacije između preostala dva ViewControllera. ListViewController predstavlja prvi ekran i on je ugrađen unutar upravitelja navigacije, dok se drugi ekran, odnosno DetailsViewController naknadno dodaje unutar navigacijskog upravitelja te se tako ostvaruje navigacija između ekrana.



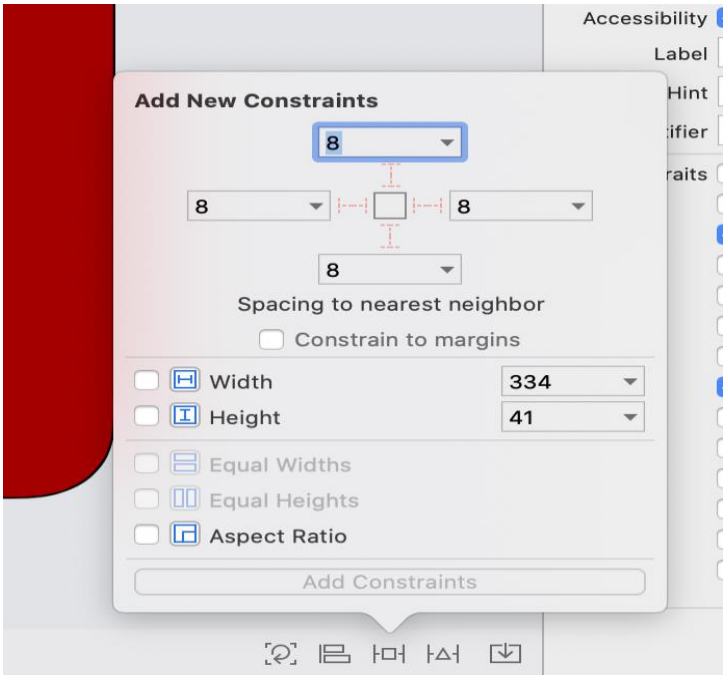
Slika 17 Pregled ViewControllera unutar uređivača grafičkog sučelja

Za realizaciju prvog ekrana korištene su komponente UITableView koji koristi UITableCell kao svaki element unutar tablice. Za pretraživanje se koristi UISearchController koji se ugrađuje unutar NavigationBara, te UIRefreshController čija je uloga slušanje akcije povlačenja za ažuriranjem liste.



Slika 21 Element liste unutar uređivača storyboarda

Slika 21 prikazuje jedan element tablice unutar uređivača grafičkog sučelja, koji će se koristiti za popunjavanje tablice podacima. Nakon kreiranja sučelja sve što je potrebno je kreirati Constraint elemente (slika 22) opcijom koja se nalazi unutar uređivača.



Slika 22Prikaz opcija za uređivanje rasporeda između grafičkih elemenata

Punjenje UITableView komponente podacima je prikazano isječkom kôda 20 i odvija se tako da se je komponenta pregleda proširila protokolima UITableViewDelegate i UITableViewDataSource koji osiguravaju sljedeće metode:

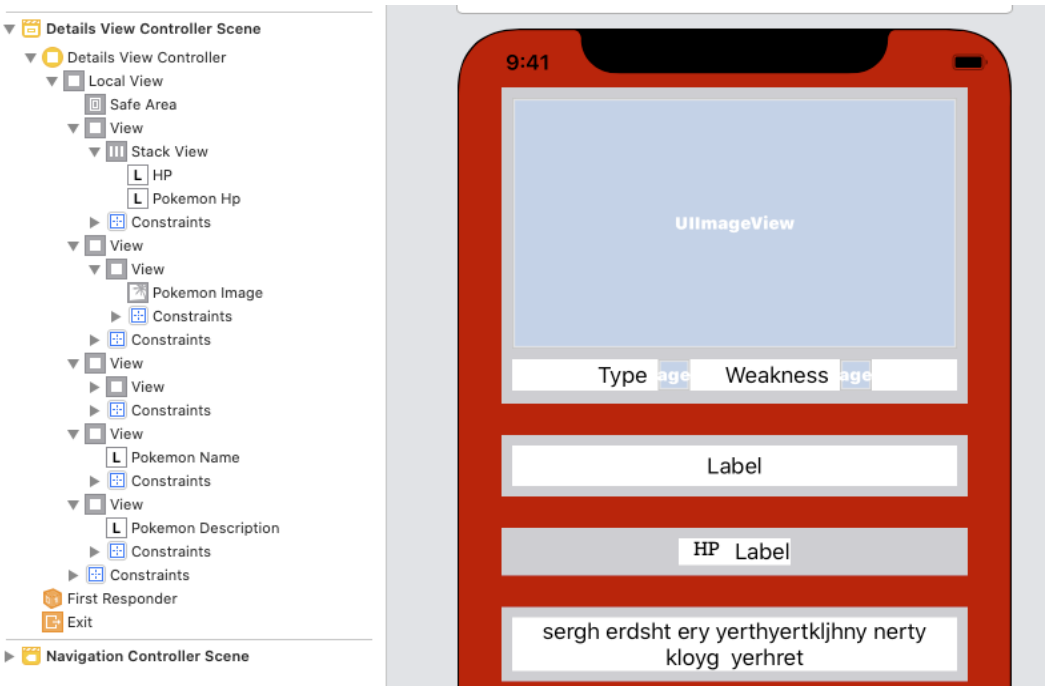
- tableView:cellForRowAtIndexPath, služi za povezivanje ćelije s podacima, te

- tableView:numberOfRowsInSection, koja služi listi da zna koliko ćelija je potrebno prikazati.

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

        return _filteredPokemonData.count

    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier:
"pokemonTableCell", for: indexPath) as! PokemonTableCell


        cell.prepare(pokemonList:
_filteredPokemonData[indexPath.row], contract: contract!)

        return cell

    }
```

Isječak kôda 20 Isječak kôda kojim se upravlja punjenjem tablice

Drugi ekran je realiziran uz pomoć jednostavnih komponenata, kao što su UIImage, Label i View komponente. Na slici 23 je prikazan raspored elemenata unutar grafičkog sučelja drugog ekrana aplikacije. Korijenska komponenta sučelja koja je podijeljena u nekoliko sekcija koje su raspoređene unutar sučelja uz pomoć constraints elemenata. Sve sekcije osim sekcije koja se nalazi ispod slike je popunjena s Label elementima, dok se složena komponenta popunjan sa slikama i s labelama,



Slika 18 Prikaz drugog ekrana unutar uređivača grafičkog sučelja

5.3. Flutter

Flutter aplikacija je napisana u Dart programskome jeziku. Za izradu Flutter aplikacije korištene su osnovne komponente kao widget bez stanja i widget sa stanjem, te neke od vanjskih biblioteka.

5.3.1. Priprema aplikacije

Prilikom izrade Flutter aplikacije korištene su vanjske biblioteke koje omogućuju jednostavniji dohvat i obradu podataka, te biblioteka unutar koje se nalazi priprema za MVC arhitekturu koja je prethodno detaljnije opisana. Za realizaciju aplikacije korištene su sljedeće biblioteke.

- mvc, biblioteka koja sadrži sve elemente koji se koriste za MVC arhitekturu,
- http je biblioteka koja olakšava kreiranja poziva prema API pristupnim točkama, te
- recase, biblioteka koja se koristi za uređivanje String-ova.

```
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2
  mvc: ^0.0.5
  http: ^0.12.2
  recase: ^3.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter
```

Slika 19 Prikaz biblioteka koje su korištene prilikom izrade Flutter aplikacije

Dodavanje vanjskih biblioteka unutar Flutter projekta je relativno jednostavna operacija. Za upravljanje vanjskim bibliotekama koristi se YAML sustav skripti. Proces dodavanja vanjske biblioteke unutar Flutter projekta provodi se na sljedeći način. Prvi korak je taj da se unutar dependencies sekcije dodaje ime i dodati broj verzije željene biblioteke. Nakon što se naprave izmjene potrebno je pokrenuti komandu *Pug get* kojom se pokreće preuzimanje i implementacija nove biblioteke unutar projekta.

5.3.2. Izrada aplikacije

Za implementaciju MVC arhitekture unutar Flutter projekta bilo je potrebno napraviti pripremu. Sama priprema je opisana unutar poglavlja 4.5. Isječci kôda 21 i 22 prikazuju prikazuje povezivanje komponenata MVC arhitekture. Unutar komponente upravitelja prvo se kreira komponenta modela, zatim se kreira komponenta pregleda kojoj se prosljeđuje objekt komponente modela, gdje se objekt modela prosljeđuje objektu stanja. Kada je objekt stanja kreiran, stanje se prijavljuje na model radi slušanja promjene stanja podataka unutar modela. Metoda *init*() se koristi za uspostavu interakcija između komponente pregleda i upravitelja (Isječak kôda 23).

```
ListController() {
    model = ListModel();
    view = MainListView(model);
    init();
}
```

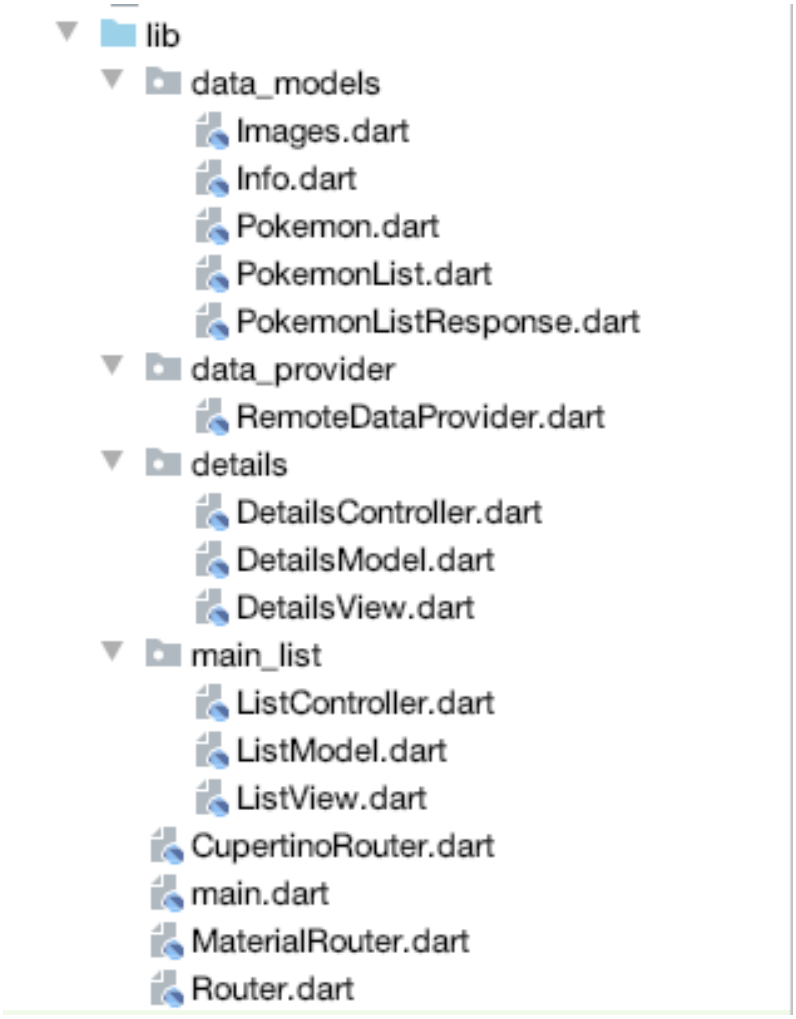
Isječak kôda 21 Isječak kôda kreiranja komponente upravitelja

```
ListViewState(model) : super(model) {
    model.subscribe(this);
}
```

Isječak koda 22 Isječak kôda kreiranja komponente pregleda

```
void init() {
    _streamController = StreamController();
    view.addNotifier(_streamController);
    _streamController.stream.listen(eventHandler);
}
```

Isječak koda 23 Isječak kôda metode init()



Slika 20 Struktura Flutter aplikacije

Slika 25 prikazuje strukturu Flutter aplikacije. Projekt je podijeljen unutar četiri sekcije. Prva sekcija sadrži modele za podatke koji se koriste unutar aplikacije. Druga sekcija sadrži statičnu klasu koja se koristi za dohvat vanjskih podataka, treća sekcija sadrži komponente od kojih je građen drugi ekran, dok četvrta sekcija sadrži komponente prvog ekrana. Tu su još pomoćne klase Router koje se koriste za navigaciju između ekrana.

Isječak kôda 24 prikazuje elemente kojima se omogućuje za kretanje među ekranima zaslužan je Navigator, dok je Router kao pomoćna klasa za Navigator koja generira omotač za željenu rutu. Prilikom kreiranja omotača za prezentaciju novog ekrana šalju se jedinstvena oznaka rute i podatke koji su prosljeđuju prema drugome ekranu. Omotači se također podijeliti na Material i Cupertino.

```
@override
  void update({flag}) {
    if (flag == ListModel.POKEMON) {
      var pokemon = model.getPokemon();
```

```
            Navigator.of(context).pushNamed(Router.DETAILS_ROUTE,
arguments: pokemon);
          }
      }
      abstract class Router {
        static const String INITIAL_ROUTE = "initial_route";
        static const String ROUTE_SPLASH = "splash";
        static const String DETAILS_ROUTE = "details_route";

        Route<dynamic> generateRoute(RouteSettings settings);
      }
      class CupertinoRouter extends Router {
        @override
        Route generateRoute(RouteSettings settings) {
          if (settings.name == Router.DETAILS_ROUTE) {
            return CupertinoPageRoute(
                builder: (_) => DetailsController(pokemon:
      settings.arguments));
          }
          return CupertinoPageRoute(builder: (_) => ListController());
        }
      }
      class MaterialRouter extends Router {
        @override
        Route generateRoute(RouteSettings settings) {
          if (settings.name == Router.DETAILS_ROUTE) {
            return MaterialPageRoute(
                builder: (_) => DetailsController(pokemon:
      settings.arguments));
          }
          return MaterialPageRoute(builder: (_) => ListController());
        }
      }
```

Isječak koda 24 Isječci kôda kojima se realizira prikazivanje novog ekrana

Za dohvat stanja baterije uređaja potrebno je napraviti kanal prema platformi na kojoj se izvršava aplikacija. Prvi korak je kreiranje kanala prema platformama, to se izvodi tako da se unutar Android i iOS module koji se nalaze u projektu Flutter aplikacije registriraju kanali

(Isječak kôda 25 i 26). Unutar Android registracija na kanal se odvija tako da se kreira prvo kreira MethodChannel s parametrima koji su komponenta binary messanger koja se nalazi unutar Flutter stroja i ime kanala na koji se spaja. Nakon što je kreiran kanal postavlja se slušač čijim se okidanjem se u kanal šalju podaci o trenutnom stanju baterije.

```
      const val CHANNEL_NAME = "battert_state_channel"

      MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
CHANNEL_NAME).setMethodCallHandler { call, result ->

                  if (call.method == "getBatteryLevel") {

                      val batteryLevel = getBatteryLevel()


                      if (batteryLevel != -1) {

                          result.success(batteryLevel)

                      } else {

                          result.error("UNAVAILABLE", "-1", null)

                      }

                  } else {Wresult.notImplemented()}

      }
```

Isječak koda 25 Isječak kôda spajanja kanala sa Android platformom

Za dohvaćanje stanja baterije za iOS uređaja potrebno je napraviti sljedeću pripremu. Prvi korak je dohvaćanje trenutnog ViewController na kojemu se izvršava Flutter stroj. Nakon dohvaćanja ViewControllera potrebno je uspostaviti vezu prema kanalu, taj proces je isti kao i na Android, jedina razlika je naziv klase koja se kreira. Nakon kreiranja objekta kanala, na njega se priključuje slušač čijim se okidanjem vraća trenutno stanje baterije na iOS uređaju.

```
      let controller : FlutterViewController = window?.rootViewController
as! FlutterViewController

          let batteryChannel = FlutterMethodChannel(name:
"battert_state_channel", binaryMessenger: controller.binaryMessenger)

          batteryChannel.setMethodCallHandler({

            (call: FlutterMethodCall, result: @escaping FlutterResult) ->
Void in

              guard call.method == "getBatteryLevel" else {

                result(FlutterMethodNotImplemented)

                return

              }

              self.receiveBatteryLevel(result: result)

        })
```

Isječak koda 26 Isječak kôda spajanja kanala sa iOS platformom

Za povezivanje kanala unutar Flutter aplikacije potrebno je kreirati objekt MethodChannel klase i unutar konstruktora proslijediti naziv kanala na koje se želi spojiti. Isječak kôda 27 prikazuje funkciju _getBatteryLevel() koja ima povratnu vrijednost tipa Future što znači da se povratna vrijednost funkcije vraća asinkrono u odnosu na poziv funkcije. Unutar funkcije se uz pomoć kreiranog objekta kanala se okidaju slušači koji su registrirani na strani platforme uz pomoć metode *invokeMethod(methodName: String)* gdje joj se prosljeđuje ima metode koja se želi izvršiti. Ključna riječ await označava da se rezultat metode ne očekuje odmah, već da će biti vraćen asinkrono. Radi toga se koristi povratna vrijednost Future i ključna riječ async.

```
Future<void> _getBatteryLevel() async {
    String batteryLevel;
    try {
      final int result = await
platformChannel.invokeMethod('getBatteryLevel');
      batteryLevel = result.toString();
    } on PlatformException catch (e) {
      batteryLevel = e.message;
    }
     setState(() {
      _batteryLevel = batteryLevel;
    });
  }
```

Isječak koda 27 Isječak kôda preuzimanja informacije o stanju baterije preko kanala

5.3.3. Grafičko sučelje

Za kreiranje korisničkog sučelja koriste se widgeti. Grafičko sučelje se gradi tako da se ugnježđuju željeni widgeti te se tako stvara stablo widgeta koje definira grafičko sučelje. Widgeti se ugnježđuju tako da se prosljeđuju unutar konstruktora svojeg roditelja. Također se svi atributi koje widget treba posjedovati prosljeđuju unutar konstruktora.

Flutter aplikacije su više-platformske aplikacije i moguće ih je izvoditi na mobilnim platformama na Android i iOS, iz tog razloga postoje komponente koje izrađene po uzoru na prirodne komponente svake od platformi kako bi aplikacija izgledala što sličnije prirodnoj aplikaciji. Radi toga je potrebno provjeravati na kojoj se platformi izvršava aplikacija, kako bi se mogao iscrtati widget koji je oponaša izgled slične komponente unutar prirodne aplikacije. Kako su sve komponente grafičkog sučelja widgeti, ta se operacija postiže jednostavnim

grananjem, gdje se unutar svake od grane kreira svoj widget s određenim atributima specifičnim za željenu platformu.

Isječak kôda 28 prikazuje korijenske widgete od kojih je građena Flutter aplikacija. Korijenski widget se određuje jednostavnim grananjem ovisno tome na kojoj se platformi izvršava Flutter aplikacija. MaterialApp je korijenski widget za Android aplikaciju, dok je CupertinoApp korijenski widget za iOS aplikaciju. Unutar konstruktora se prosljeđuju elementi kao što su naslov aplikacije, koji je početni ekran aplikacije i inicijalna ruta aplikacije. Za generiranje rute Navigatora postavlja se metoda iz jedne od realizacija usmjerivača koje služe za kreiranje omotača koji je specifičan za platformu nad kojoj se trenutno izvodi aplikacija.

```
@override
Widget build(BuildContext context) {
  return Platform.isAndroid
      ? MaterialApp(
          title: "",
          home: ListController(),
          initialRoute: Router.ROUTE_SPLASH,
          onGenerateRoute: MaterialRouter().generateRoute)
      : CupertinoApp(
          title: "",
          home: ListController(),
          initialRoute: Router.ROUTE_SPLASH,
          onGenerateRoute: CupertinoRouter().generateRoute,
        );
}
```

Isječak koda 28 Isječak kôda korijenskog Widgeta

Prvi ekran Flutter realiziran je pomoću Scaffold i CupertinoPageScaffold widgeta koji se koriste kao korijenski widgeti prvog ekrana. Isječak kôda 28 prikazuje isječak kôda navigacijske elemente prvog ekrana unutar kojih su sadržani elementi kao što su navigacija i slično. Scaffold se koristi za Android komponente, dok se CupertinoPageScaffold se koristi za komponente iOS platforme. Ostali elementi koji su korišteni prilikom realizacije ekrana su ListView za Android i SliverList za iOS platformu.

```
      @override
        Widget build(BuildContext context) {
          return Platform.isAndroid
              ? Scaffold(
                  appBar: AppBar(
                    backgroundColor: Color.fromARGB(255, 12, 12, 12),
                    actions: <Widget>[
                      Padding(
                        padding: EdgeInsets.only(right: 24, top: 24),
                      child: Text(
                        _batteryLevel,
                        style: TextStyle(color: Color.fromARGB(255, 255,
255, 255), fontSize: 18),
                      )),
                    ],
                    title: typing
                        ? searchBox()
                        : Text(search == "" ? "Pokedex" : search,
                            style: titleStyle()),
                    leading: IconButton(
                      icon: Icon(typing ? Icons.done : Icons.search),
                      onPressed: () {
                        setState(() {
                          typing = !typing;
                        });
                      },
                    ),
                  ),
                  body: body())
              : CupertinoPageScaffold(
                  navigationBar: CupertinoNavigationBar(
                      backgroundColor: Color.fromARGB(255, 12, 12, 12),
                      trailing: Text(
                        _batteryLevel,
                        style: TextStyle(color: Color.fromARGB(255, 255,
255, 255)),
                      ),
```

```
            leading: CupertinoButton(
              child: Icon(typing ? Icons.done : Icons.search),
              onPressed: () {
                setState(() {
                  typing = !typing;
                });
              },
            ),
            middle: typing
                ? searchBox()
                : Text(search == "" ? "Pokedex" : search,
                    style: titleStyle())),
        child: Scaffold(
            body: SafeArea(
          child: body(),
        )));
}
```

Isječak koda 29 Isječak kôda navigacijske trake prvog ekrana

```
        Widget body() {
          return Container(
              color: Color.fromARGB(255, 240, 240, 240),
              child: Stack(children: [
                Container(
                    color: Color.fromARGB(255, 12, 12, 12),
                    child: Platform.isAndroid
                        ? RefreshIndicator(
                            child: materialList(),
                            onRefresh: onRefresh,
                          )
                        : CustomScrollView(
                            slivers: <Widget>[
                              CupertinoSliverRefreshControl(onRefresh:
onRefresh),
                              cupertinoList()
                            ],
                          )),
                Visibility(
                    visible: isLoading,
                    child: Container(
                        color: Color.fromARGB(140, 0, 0, 0),
                        alignment: Alignment.center,
                        child: CircularProgressIndicator())),
                Visibility(
                    visible: error != null,
                    child: Container(
                        color: Color.fromARGB(140, 0, 0, 0),
                        alignment: Alignment.center,
                        child: Text(
                          error ?? "",
                          style: TextStyle(color: Color.fromARGB(255, 0, 0,
0)),
                        ))),
              ]));
        }
```

Isječak koda 30 Isječak kôda tijela prvog ekrana

Isječak koda 31 prikazuje način kreiranja liste s podacima. Prva metoda se koristi za kreiranje Cupertino liste točnije za kreiranje liste koja ima atribute kao i unutar prirodne iOS aplikacije, dok se druga metoda koristi za kreiranje Material liste koja se koristi unutar Android aplikacije. Za kreiranje Cupertino liste koristi se widget SilverList koji koristi widget graditelja (*eng. Builder*) liste SilverChildBuilderDelegate. Unutar graditelja se prosljeđuje element liste koji je omotan slušačem klika. Kao dodatni parametar za graditelja se šalje veličine liste koje ju potrebno izlistani. Za kreiranje Material liste koristi se ListView.Builder, koji je jako sličan graditelju liste koji se koristi za Cupertino listu. Parametri koji se prosljeđuju tijekom kreiranja graditelja su broj elemenata liste i widget koji će se koristiti kao element unutar liste. Iz koda se može vidjeti da je i taj widget koji predstavlja element liste omotan slušačem, widget je omotan slušačem kako bi se mogao registrirati odabir elementa liste.

```
Widget cupertinoList() {
    return SliverList(
      delegate: SliverChildBuilderDelegate((context, index) {
        return GestureDetector(
            child: listItem(widget.filteredPokemons[index]),
            onTap: () {
              widget.notifyController(
                  MVCEvent(null,
widget.filteredPokemons[index].codeName));
            });
      }, childCount: widget.filteredPokemons.length),
    );
  }
  Widget materialList() {
    return ListView.builder(
        itemCount: widget.filteredPokemons.length,
        itemBuilder: (context, index) {
          return GestureDetector(
              child: listItem(widget.filteredPokemons[index]),
              onTap: () {
                widget.notifyController(
                    MVCEvent(null,
widget.filteredPokemons[index].codeName));
              });
        });
```

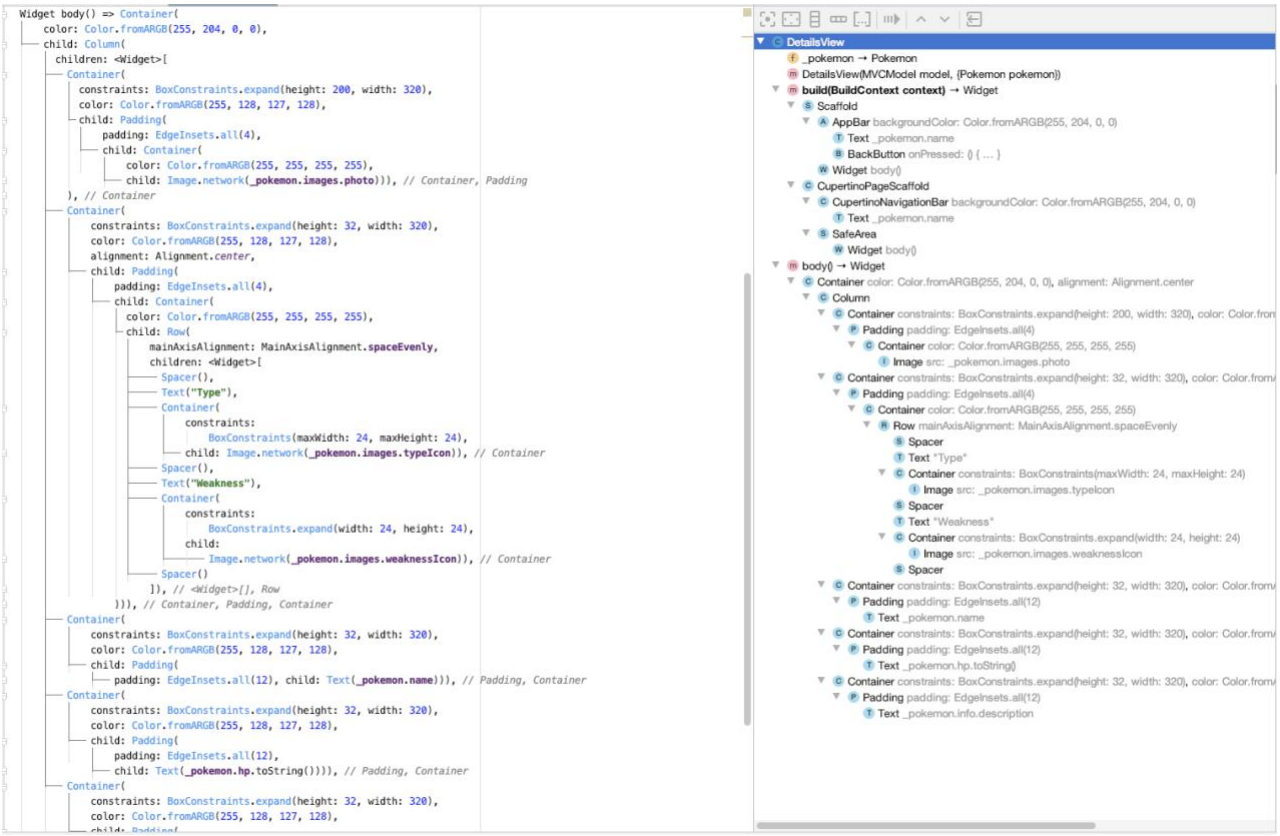
Isječak koda 31 Kreiranje lista s podacima unutar Dart kôda

Jedan element liste je prikazan na isječkom kôda 32. Svaki element liste se kreira pomoću metode koja se zove listItem i koja poprima objekt Pokemon za kojeg će prikazivati podatke. Prvi Widget je Container, on se koristi kako bi se mogla pobojati pozadina jednog elementa liste u određenu boju. Kao parametar unutar konstruktora Container-a prosljeđuje se element Padding koji se tako ugnježđuje unutar Container elementa. Padding element se koristi kako bi se nad njegovim ugniježđenim elementom primijenili padding-i. Unutar elementa Padding dodan je element Row koji ima ulogu slaganja svoje djece vodoravno, odnosno s lijeva na desno. Kao djeca Row elementa su Image, koji preuzima sliku preko interneta, uz pomoć url-a i Text element koji ispisuje tekst na ekranu. Unutar elementa Text postoji atributi fontFamily koji označava kojim fontom će biti ispisana tekst i color, odnosno kojom bojom će biti pobojan ispisani tekst.

```
Widget listItem(PokemonList pokemon) {
    return Container(
        color: Color.fromARGB(255, 12, 12, 12),
        child: Padding(
            padding: EdgeInsets.all(24),
            child: Row(
              children: <Widget>[
                SizedBox(
                  width: 48,
                  height: 48,
                  child: Image.network(pokemon.url),
                ),
                Text(pokemon.name,
                    style: TextStyle(
                        fontFamily: "8-BIT",
                        color: Color.fromARGB(255, 0, 255, 0)))
              ],
            )));
  }
```

Isječak koda 32 Element liste unutar Dart kôda

Za komponentu koja omogućuje osvježavanje sadržaja jednostavnim povlačenjem se koristi RefreshIndicator za Android i CupertinoSliverRefreshControl za iOS. Svaka od navedenih komponenata je kreirana kao roditelj svakoj od listi. Polje za pretraživanje se realizirano s osnovnim komponentama, te nije bilo potrebe za korištenjem Widget-a specifičnih za svaku od platformi.



Slika 21 Primjer uređivanja grafičkog sučelja drugog ekrana

Drugi ekran (Slika 26) realiziran je jednostavnim widgetima, kao što su Text, Image i ostali pomoćni elementi kao što su Padding, Column i slično. Korijenski komponenta ekrana je Column čije svojstvo da se elementi koji se nalaze unutar nje iscrtavaju vertikalno, odnosno od gore prema dolje. Kako je ekran podijeljen u nekoliko sekcija, svaka sekcija je odvojena unutar Container komponente koja služi za primjenu rasporeda slika i tekstualnih elemenata koji se nalaze unutar nje. Sekcija koja se nalazi ispod sekcija sa slikom ima malo složeniju strukturu i ona je realizirana uz pomoć Row komponente. Row komponenta je slična Column komponenti samo je razlika u tome što Row svoje elemente iscrtava vodoravno, odnosno s lijeva na desno. Za dodavanje prostora između elemenata korištena je komponenta Padding, dok je komponenta Space korištena za popunjavanje slobodnog mjesta unutar komponente kako bi se održao željeni raspored unutar komponente.

Kako je unutar ovog poglavlja detaljnije pojašnjen proces izrade prirodnih aplikacija za Android i iOS, i aplikacije za Flutter, unutar sljedećeg poglavlja će se usporediti procesi izrade aplikacija i dojam koji ostavlja Flutter aplikacija u odnosu na prirodne aplikacije.

6. Usporedba aplikacija

U prethodnom poglavlju su detaljnije opisani procesi izrade aplikacija. Kao rezultat opisanih procesa su tri aplikacije, gdje su dvije aplikacije koje su pisane prirodno za mobilnu platformu, dok je treća Flutter aplikacija. Unutar ovog poglavlja će se usporediti procesi izrade i aplikacije kao finalni proizvod procesa izrade.

Kriteriji po kojima će se vršiti usporedba su:

- Složenost pripreme koja je potrebna kako bi se izradila aplikacija,
- Proces izrade aplikacije,
- Spajanje na specifičnosti platforme,
- Izrada i manipulacija s elementima grafičkog sučelja,
- Aplikacija kao finalni proizvod, izvođenje Flutter aplikacije na platformi u odnos na prirodnu aplikaciju za tu platformu.

6.1. Priprema

Pod pripremu aplikacije smatra se svi koraci koji su potrebni kako bi se moglo neometano započeti s izradom aplikacije s pretpostavkom da su već ranije pripremljeni svi potrebni alati za izradu aplikacije. Tako rečeno priprema za izradu aplikacija se odnosi na to koje je sve korake potrebno napraviti nakon kreiranja projekta kako bi se započelo s izradom aplikacije.

Za izradu prirodnih aplikacija za platforme Android i iOS koriste se elementi kao što su aktivnost koja sadrži kôd aplikacije i XML datoteke koja predstavlja grafičko sučelje prirodne aplikacije Androida, dok su kôd iOSa to ViewController koji sadrži programski kôd i StoryBoard koji sadrži grafičke elemente aplikacije. Za izradu Flutter aplikacija koriste se samo widgeti i stanja, pod uvjetom da se koristi widget sa stanjem inače se ne koriste stanja. Radi toga glavna razlika između Flutter aplikacija i prirodnih aplikacija je to što se kod prirodnih aplikacija omogućeno odvajanje grafičkog sučelja aplikacije od kôda aplikacije, dok kod Fluttera to nije slučaj, kod njega su te dvije komponente povezane, pa se grafičko sučelje kreira unutar samog kôda. Što se tiče razvoja aplikacija, ta razlika se najviše razlikuje u primjeni arhitekture. Arhitekture koje su danas raširene i koje se često koriste su lakše za implementirati na prirodnim platformama, nego na Flutteru, gdje je potrebno napraviti pripremu ili je potrebno proučiti novije arhitekture koje se mogu lakše implementirati unutar Flutter aplikacije.

Što se tiče proširivanje projekata vanjskim bibliotekama, za prirodne aplikacije je bilo veće potrebe za vanjskim bibliotekama nego li je to slučaj kod Fluttera, gdje je izuzev biblioteke koja je korištena kao priprema za MVC arhitekturu i biblioteke za uređivanje podataka tipa String, korištena samo jedan biblioteka i to je biblioteka koja se koristi za povezivanje na vanjske pristupne točke. Za prirodne aplikacije su se koristile biblioteke za povezivanje na vanjske pristupne točke, za pasiranje dobivanih podataka te biblioteka za preuzimanje i prikazivanje slika koje se nalaze na udaljenoj lokaciji. Što se tiče izrade jednostavnih aplikacija tu je prednost za Flutter aplikacije, jer nije potrebno implementirati veliki broj vanjskih biblioteka, te samim time se olakšava održavanje aplikacija jer nema potrebe za ažuriranjem biblioteka i kôda radi promjena unutar samih biblioteka.

Prilikom izrade Android i iOS aplikacija korišten je veći broj vanjskih biblioteka nego li je to slučaj kod Flutter aplikacije. Najviše biblioteka se odnosilo na povezivanje na API pristupne točke i na pasiranje podataka. Za Android i iOS su također korištene biblioteke za preuzimanje i prikazivanje slikama koje nisu koje se nalaze na udaljenoj lokaciji i koje je potrebno.

Što se tiče izrade složenih aplikacija koje su ovisne o pozadinskim servisima i slično, bolja odluka je izrada prirodnih aplikacija. Izrada takvih aplikacija za Flutter nije ne moguća već je potrebna veća priprema. Unutar pripreme spada pisanje potrebnih elemenata unutar prirodne platforme, te je potrebno povezati takve elemente sa Flutter aplikacijom preko kanala. Unutar repozitorija koji sadrži biblioteke za Flutter se iz svakog dana nalazi sve veći broj biblioteka koji olakšavaju taj proces spajanja na specifičnosti svake platforme, ali to postavlja pitanje koliko često se ažuriraju takve biblioteke i koliko često je potrebno ažurirati aplikacije radi izmjena. Radi toga su prirodne aplikacije bolji izbor kada su u pitanju složenije aplikacije.

6.2. Izrada

Proces izrade aplikacija se nije previše razlikovao. Glavna razlika je bila programski kôd pomoću kojega je pisana aplikacija, uz programski jezik razlika se pojavila kada je bilo potrebno preuzeti podatke specifične za pojedinu platformu. Kod prirodnih aplikacija je bilo dovoljno napisati dio kôda za dohvaćanje podataka, dok je prilikom izrade Flutter aplikacije bilo potrebno napisati sličan dio kôda na svakoj platformi, te je bilo potrebno izraditi kanale kojima bi se ti podaci dopremili unutar Flutter aplikacije.

Što se tiče primjene arhitekture unutar aplikacija nije bilo velikih razlika, to je slučaj jer se napravila priprema kojim se ista arhitektura implementirala unutar svih aplikacija. Da nije bilo takve pripreme izrada aplikacija bi se uvelike razlikovala. Što se tiče prirodnih aplikacija i

arhitektura koje se koriste za njihovu izradu, tu je prednost u odnosu na Flutter jer se za Flutter kreiraju nove vrste arhitektura koje još nisu toliko raširene kao arhitekture koje se koriste za izradu prirodnih aplikacije.

6.3. Grafičko sučelje

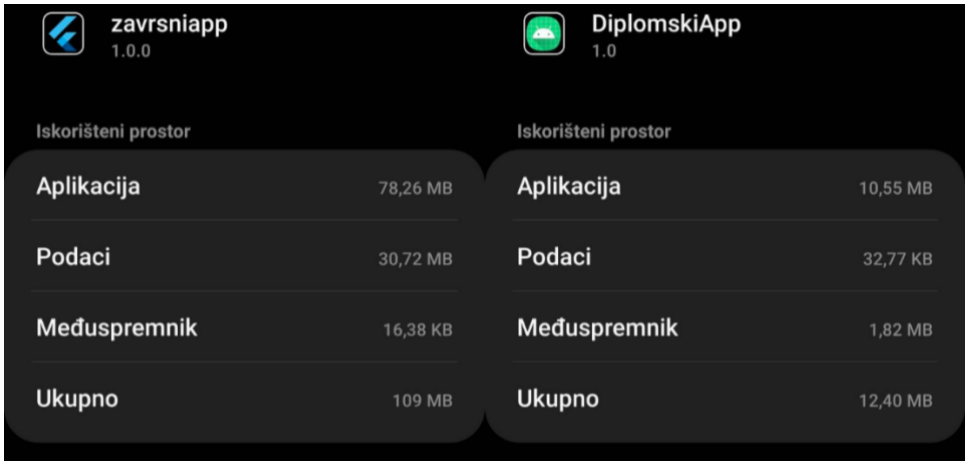
Kao što je prethodno navedeno, glavna razlika između prirodnih aplikacija i Flutter aplikacije, što se tiče izrade grafičkog sučelja je ta da kod prirodnih aplikacija postoji podjela između grafičkog sučelja i kôda, što nije slučaj kod Fluttera. Kod Fluttera se komponente grafičkog sučelja generiraju direktno unutar samog kôda. Prednost takvog pristupa je ta da nema potrebe za spajanjem elemenata grafičkog sučelja s kôdom. Nedostatak je taj da je takav kôd složeniji i samim time teži za daljnje održavanje.

Druga razlika koja se pojavljuje prilikom izrade prirodnih aplikacija i Flutter aplikacija su prirodne komponente grafičkog sučelja. Kako je Flutter razvojni okvir više-platformski razvoj aplikacija sadrži mogućnost korištenja widgeta koji su slični onima unutar prirodnih aplikacija kako bi se finalna aplikacija što manje razlikovala on onih koje su napisane prirodno za platformu. S tom mogućnošću se javlja problem izrade takvog sučelja koje će se svakoj platformi zato što je potrebno koristiti grananja prilikom izrade sučelja, gdje svaka grana sadrži widget koji ima svojstva te platforme.

6.4. Završni dojam

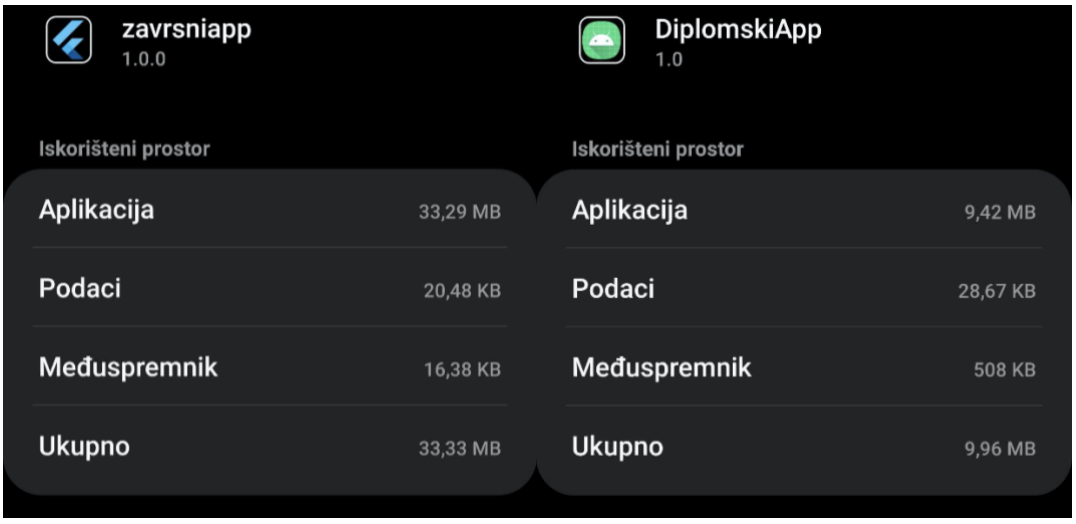
Kao završni dojam uspoređivat će se izgled i ponašanje aplikacija između prirodno napisanih aplikacija i Flutter aplikacije. Uz to će se i usporediti prostor kojeg zauzimaju instalirane aplikacije.

Na slici 27 su prikazane veličine aplikacija instaliranih na mobilnoj platformi Android. Lijevo je prikazana veličina Flutter aplikacije koja je izgrađena u verziji za otklanjanje pogrešaka, dok je desno veličina prirodne Android aplikacije. Flutter aplikacija je velika 109 MB, dok je prirodna aplikacije velika 12.4 MB. Ako pogledamo strukturu arhitekture Flutter razvojnog okvira, možemo pretpostaviti da najveći dio memorijskog prostora zauzima Flutter stroj za izvođenje Flutter aplikacija.



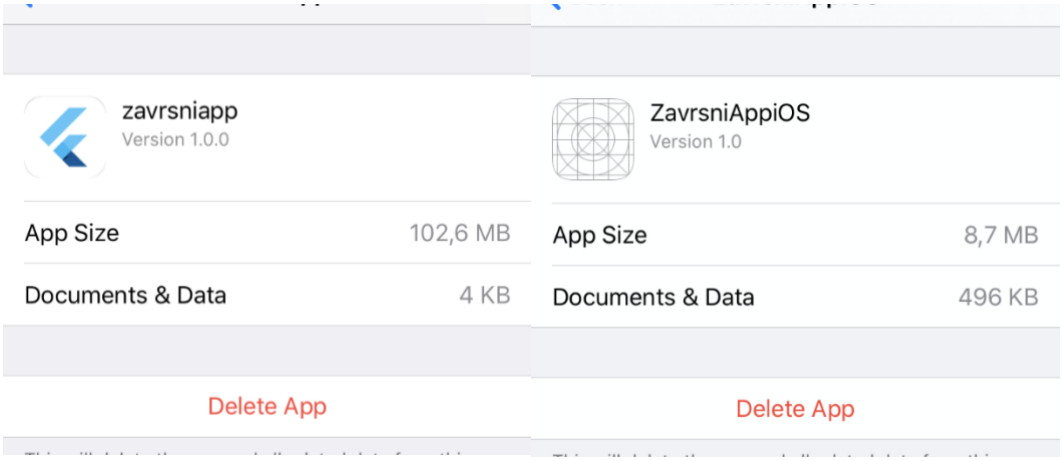
Slika 22 Veličina verzije za otklanjanje pogrešaka Flutter i prirodne Android aplikacije

Veličine aplikacije produkcijskih verzija Flutter i prirodne Android aplikacije prikazane su slikom 28, slika lijevo prikazuje veličinu Flutter aplikacije, dok je desno prirodna Android aplikacija. Iz slike se odmah da vidjeti kako je produkcijska verzija aplikacije manja od verzije za otklanjanje pogrešaka. Flutter aplikacije je u produkcijskog verziji manja za 76 MB, odnosno produkcijska verzija aplikacije zauzima 30% od verzije za otklanjanje pogrešaka, dok je prirodna aplikacija manje za 2.5 MB.



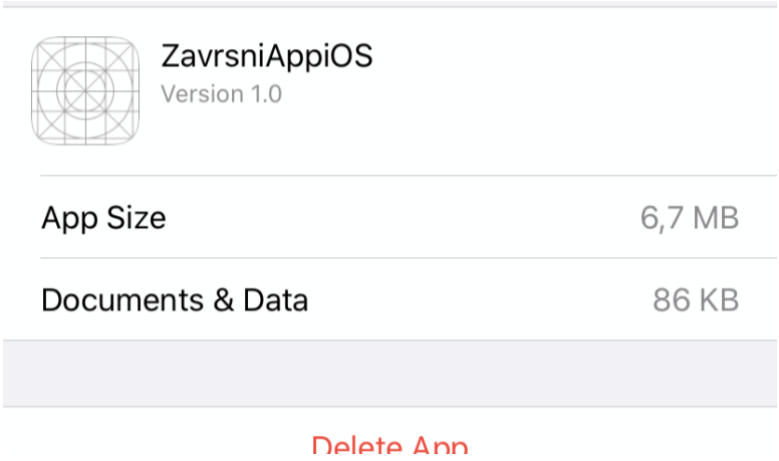
Slika 23 Veličina produkcijskih verzija Flutter i prirodne Android aplikcije

Slika 29 prikazuje usporedbu veličina Flutter aplikacije (Lijevo) i iOS aplikacije (Desno). Sa slika se može vidjeti da prirodna aplikacija zauzima mnogo manje memorijskog prostora od Flutter aplikacije, točnije prirodna iOS aplikacija zauzima 8.7 MB, dok Flutter aplikacija zauzima 102 MB. Ako se usporede veličine Flutter aplikacija na prirodnim platformama i ako se pogleda struktura arhitekture Flutter razvojnog okvira može se zaključiti da je glavni razlog veličine Flutter aplikacije, Flutter stroj za izvršavanje Flutter aplikacija.



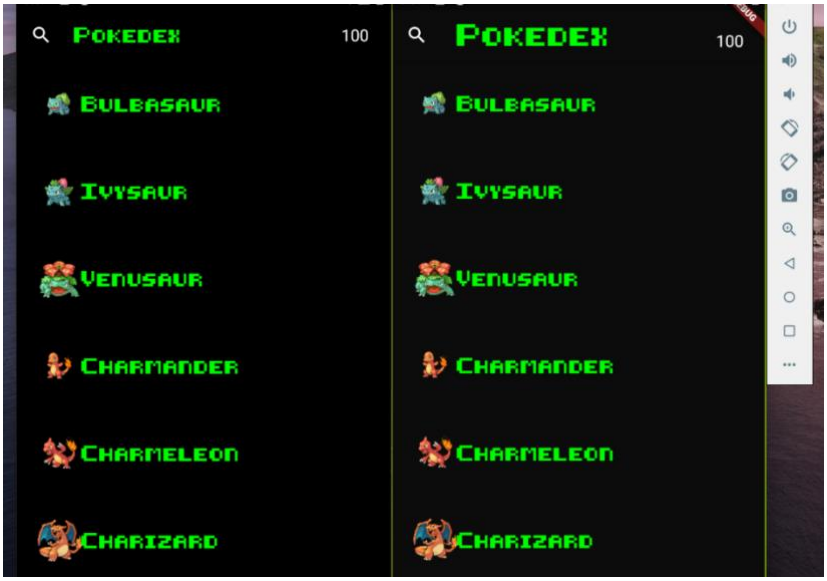
Slika 24 Veličina verzije za otklanjanje pogrešaka Flutter i iOS aplikacije

Slika 30 prikazuje veličinu prirodne iOS aplikacija. Flutter aplikacija je izostavljena iz razloga nemogućnosti kreiranja produkcijske verzije zbog restrikcija besplatnog certifikata razvojnog programera. Zbog poteškoća izrade produkcijske verzije Flutter aplikacije, za veličinu produkcijske verzije aplikacije pretpostavit će se da je aplikacija slična onoj na Android platformi. To se pretpostavilo na osnovi usporedbe verzija za otklanjanje pogrešaka Flutter aplikacija instaliranih na prirodnim mobilnim platformama. Zbog toga se pretpostavlja da produkcijska verzija Flutter aplikacije instalirane na iOS platformi zauzima oko 30 MB memorijskog prostora. Prirodna iOS aplikacija zauzima 6.7 MB, što je za 2 MB manje od verzije za otklanjanje pogrešaka.



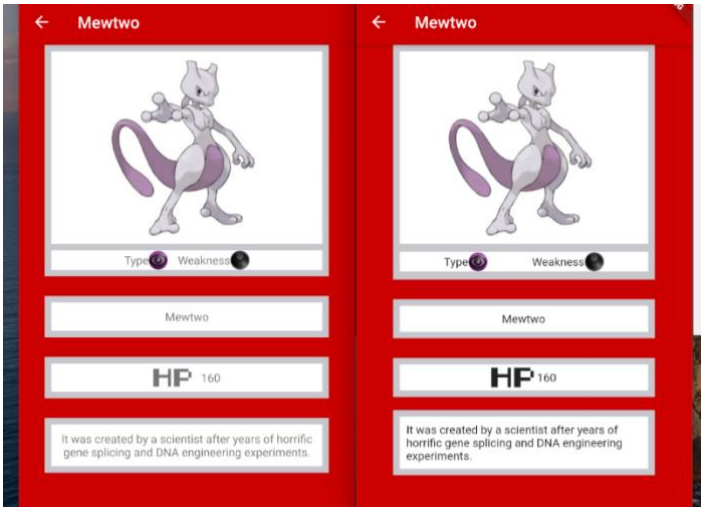
Slika 25 Veličina produkcijske verzije iOS aplikacije

Usporedbom veličina prirodnih aplikacija i Flutter aplikacije instaliranih na mobilnim platformama Android i iOS može se vidjeti da je Flutter aplikacije veća od prirodnih aplikacija, tome se može pridodati tome da je za izvođenje Flutter aplikacija potreban Flutter stroj za izvođenje Flutter aplikacija. Ako uzmemo u obzir da je danas na mobilnim uređajima dostupno dosta memorijskog prostora, razlika između prirodnih i Flutter aplikacija je nije toliko velika da bi bila presudna prilikom odabira izrade prirodnih aplikacija ili Flutter aplikacije.



Slika 26 Usporedba prvog ekrana prirodne Android aplikacije i Flutter aplikacije

Slika 31 prikazuje usporedbu prirodne Android aplikacije (Lijevo) i Flutter aplikacije (Desno). Prvi ekran aplikacije se po ničemu ne razlikuju jedan od drugoga, što znači da se unutar Fluttera može rekreirati prirodna Android aplikacija.

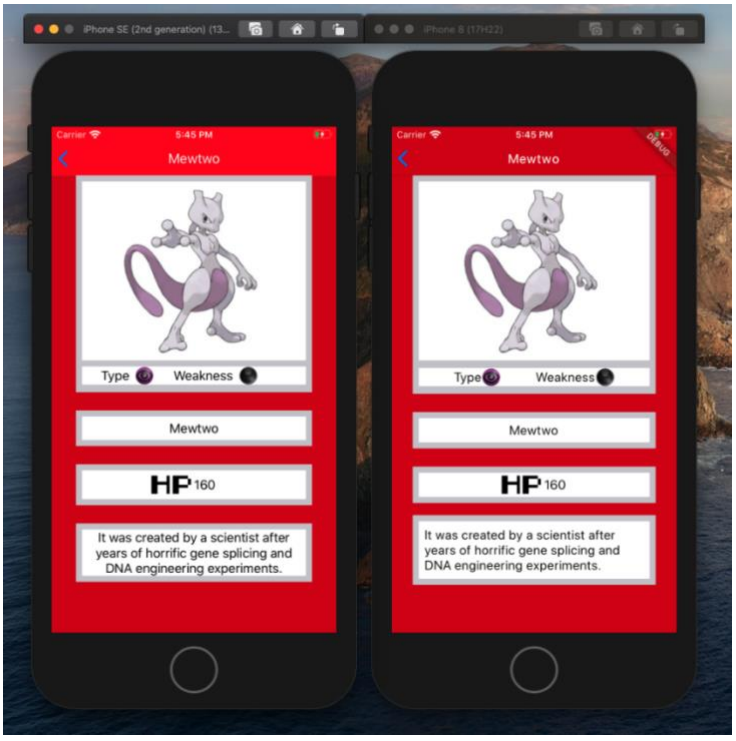


Slika 27 Usporedba drugog ekrana prirodne Android aplikacije i Flutter aplikacije

Slika 32 prikazuje drugi ekran aplikacije izrađen unutar prirodne Android aplikacije (Slika lijevo) i Flutter aplikacije (Slika desno). U odnosu na usporedbu prvog ekrana, na ovoj usporedbi se mogu vidjeti neke razlike unutar aplikacije. Prva razlika koja se primijeti je standardna boja za ispis teksta. Druga razlika koja je uočljiva je način lomljenja teksta. To se može vidjeti unutar sekcije gdje je ispisan opis odabranog entiteta.

Slika 28 Usporedba prvog ekrana prirodne iOS i Flutter aplikacije

Nas slici 33 je prikazana usporedba prvoga ekrana prirodne iOS aplikacije (Lijevo) i Flutter aplikacije (Desno). Iz usporedbe se može vidjeti razlika u polju za pretraživanje. Razlog tome je to što unutar Fluttera ne postoji mogućnost da se iscrta prirodna komponenta koja se koristi za izradu prirodnih iOS aplikacija. Drugih razlika nema.



Slika 29 Usporedba drugog ekrana prirodne iOS aplikacije i Flutter aplikacije

Slika 34 prikazuje usporedbu drugog ekrana izrađenih prirodnom iOS aplikacijom (Lijevi) i Flutter aplikacijom (Desno). Usporedbom ova dva ekrana ne mogu se učiti velike razlike.

Što se tiče usporedbe animacija i ponašanja elemenata aplikacije, razlika između prirodnih aplikacija i Fluttera se gotovo ni ne osjeti. Animacije prezentiranja novog ekrana su slične kao i kod prirodnih aplikacija, jedina razlika koja se može uočiti je izmjena boje prilikom izmjene ekrana.

Što se tiče sveukupne usporedbe doživljaja aplikacija, može se reći da aplikacije koje su napisane uz pomoć Flutter razvojnog okvira stvaraju isti dojam prilikom korištenja kao i aplikacija koja je prirodno napisana za tu platformu. Jedina veća razlika je razlika u komponentama za iOS, ali kako se Flutter još razvija može se očekivati da će se te razlike uskladiti, te da više neće biti elemenata koji se nalaze unutar prirodne aplikacije a da ih unutar Fluttera nema.

	Prirodne aplikacije	Flutter aplikacija
Primjena MVC arhitekture	Primjena MVC arhitekture unutar prirodnih aplikacija se postiže uz jednostavnu primjenu dostupnih komponenti.	Za implementaciju MVC arhitekture unutar Flutter aplikacije bila je potrebna priprema kojom su se generirale komponente arhitekture.
Dodavanje vanjske biblioteke	Dodavanje vanjskih biblioteka unutar Android aplikacije je jednostavno, dok je proces dodavanja biblioteka unutar iOSa dosta složen proces.	Dodavanje vanjskih biblioteka unutar Flutter projekta je jednako zahtijevan kao što je to slučaj kod Androida.
Potreba za vanjskim bibliotekama	Potreba za vanjskim bibliotekama se najviše koriste za rješavanje dohvata i obrade vanjskih podataka.	Za izradu Flutter aplikacije bila je potrebna biblioteka koja se koristi za dohvat vanjskih podataka.
Korištenje sustavnih podataka	Za dohvaćanje sustavnih podataka nije bilo prevelike pripreme, moglo se direktno pristupiti sustavnim podacima	Za dohvaćanje sustavnih podataka bilo je potrebno napisati kôd unutar modula koji predstavlja platformu, te je bilo potrebno izraditi kanale koji se spajaju na

		modul kako bi se pristupilo podacima
Izrada aplikacija po smjernicama za grafičke elemente platforme	Prilikom izrade aplikacija elementi grafičkog sučelja su već izrađeni prema smjernicama grafičkog sučelja	Prilikom izrade aplikacija postoji mogućnost korištenja grafičkih elemenata koji su izrađeni prema smjernicama, ali radi primjene više različitih smjernica izrada grafičkog sučelja postaje složenija
Izrada i uređivanje elemenata grafičkog sučelja	Prilikom izrade prirodnih aplikacija elementi grafičkog sučelja su odvojeni od samog kôda, radi toga je lakše manipulirati grafičkim sučelje, ali se stvara potreba za povezivanjem grafičkog sučelja sa kôdom	Grafičko sučelje i kôd aplikacije su povezani, točnije grafičko sučelje se kreira unutar samog kôda. Prednost toga je što nema potrebe za povezivanjem elemenata sa kôdom, dok je uređivanje otežano radi nedostatka prikaza.
Veličina aplikacije	Jednostavne prirodne aplikacije zauzimaju desetak megabajta memorijskog prostora	Jednostavne Flutter aplikacije zauzimaju stotinjak megabajta memoriskog prostora
Doživljaj korištenja	Aplikacije su napravljene pomoću prirodnih elemenat platforme	Izrađena Flutter aplikacija se ponaša slično kao i ista aplikacija prirodno napisana za platformu, ne osjeti se razlika prilikom korištenja aplikacije.

Tablica 1 Tablica razlika između razvoja prirodnih i Flutter aplikacija

7. Zaključak

Za razliku od mobilnih platformi Android i iOS koje su dostupne više od desetljeća, Flutter je relativno nova tehnologija, koja je dostupna sve par godina što znači da je još nije postigla svoj potpuni potencijal. Proces izrade prirodnih mobilnih aplikacija je usko vezan uz razvoj platformi, dok je Flutter takva tehnologija da se mora prilagođavati drugom platformama zato što se Flutter izvodi uz pomoć stroja za izvođenje Flutter aplikacija, koji se ponaša kao dodatni sloj nad platformom na kojoj se izvodi. U ovome radu je pokazano da se Flutter može uspoređivati s prirodnim aplikacijama za platformu na kojoj se izvodi.

Sam proces razvoja prirodnih aplikacija i Flutter aplikacija se razlikuje u odabiru arhitekture, te prilikom razvoja grafičkog sučelja za aplikaciju. Prirodne aplikacije već sadrže elemente koji se mogu primijeniti prilikom izrade standardnih arhitektura, kao što je MVC arhitektura, dok se kod Fluttera mora napraviti priprema za implementaciju standardnih arhitektura ili se mora odabrati neka nova arhitektura koju je lakšte primijeniti unutar Flutter aplikacije. Glavni razlog je u tome što je Flutter sačinjen od Widget elemenata koji su ujedno i elementi grafičkog sučelje, te je teže odvojiti aplikaciju u komponente od kojih se grade standardne arhitekture kao što su MVC.

Kako su svi elementi grafičkog sučelja nastali iz widgeta koji su prilagođeni grafičkim smjernicama za svaku platformu za koju je dostupan Flutter stroj, moguće je izraditi aplikaciju koja je izrađena prema grafičkim smjernicama koje se primjenjuju za izradu prirodnih aplikacija platforme. Proces prilagodbe aplikacije za svaku platformu zna biti kompleksan i zbunjujući jer je potrebno raditi grafičko sučelje koje prati smjernice za više platformi, ali kako se Flutter komponente prilagođavaju svakoj od platformi otvara se prostor za olakšano izrađivanje aplikacija koje ne prate grafičke smjernice, već su izrađene van smjernica.

Flutter kao razvojni okvir ima veliki potencijal za razvoj više-platformskih aplikacija. Pošto je razvojni okvir relativno nov, pokazao se kao dobra konkurencija i zamjena za razvoj prirodnih platformskih aplikacija. Sada ostaje pitanje daljnjeg razvoja Fluttera, te hoće li se u budućnosti pojaviti platforma koja će već imati ugrađeni Flutter stroj kojim će se olakšati izrada i korištenje Flutter aplikacija.

Literatura

A brief Introduction to Flutter | Dream Squad—We Develop. (2019, travanj 11). https://www.dreamsquadgroup.com/whats-flutter-brief-introduction/

Android (operacijski sustav) – Wikipedija. (2020). https://hr.wikipedia.org/wiki/Android_(operacijski_sustav)

Apple iOS Architecture. (2018, rujan 3). https://www.tutorialspoint.com/apple-ios-architecture

Compiling and running—Kotlin Programming Language. (2020). Kotlin. https://kotlinlang.org/docs/tutorials/kotlin-for-py/compiling-and-running.html

External Accessory | Apple Developer Documentation. (2020). https://developer.apple.com/documentation/externalaccessory

Flutter architectural overview. (2020). https://flutter.dev/docs/resources/architectural-overview

Flutter_isolate | Flutter Package. (2020). Dart Packages. https://pub.dev/packages/flutter_isolate

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, & Michael Stal. (1996). *Pattern-oriented software Arhitectured*.

Introduction to the Accelerate Framework in Swift | AppCoda. (2017, svibanj 26). https://www.appcoda.com/accelerate-framework/

IOS. (2020). U *Wikipedia*. https://en.wikipedia.org/w/index.php?title=IOS&oldid=978148062

IOS Application Life Cycle Example. (2018, svibanj 24). https://www.dev2qa.com/ios-application-life-cycle-example/

Koneva, A., January 21, & 2014. (2014, siječanj 21). *Interoperating Between C++ and Objective-C*. Dr. Dobb's. http://www.drdobbs.com/cpp/interoperating-between-c-and-objective-c/240165502

Making the Most of Flutter: From Basics to Customization | by Alibaba Tech | HackerNoon.com | Medium. (2018, kolovoz 1).

https://medium.com/hackernoon/making-the-most-of-flutter-from-basics-to-customization-433171581d01

Mobile Operating System Market Share Worldwide | StatCounter Global Stats. (2020, kolovoz 1). https://gs.statcounter.com/os-market-share/mobile/worldwide

[Newbie] Chapter 4. Widget's state—Nhancv's blog. (2019, studeni 12). https://nhancv.com/newbie-chapter-4-widgets-state/

Platform Architecture | Android Developers. (2020). https://developer.android.com/guide/platform

Start Developing iOS Apps (Swift): Work with View Controllers. (2020). https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/WorkWithViewControllers.html

StatelessWidget class—Widgets library—Dart API. (2020). https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html

The Engine architecture · flutter/flutter Wiki · GitHub. (2020, srpanj 10). https://github.com/flutter/flutter/wiki/The-Engine-architecture

Tung, L. (2019, srpanj 29). *Here are the real reasons Windows Phone failed, reveals ex-Nokia engineer*. ZDNet. https://www.zdnet.com/article/here-are-the-real-reasons-windows-phone-failed-reveals-ex-nokia-engineer/

Understand the Activity Lifecycle | Android Developers. (2020). https://developer.android.com/guide/components/activities/activity-lifecycle

Popis slika

Popis isječaka kôda

Popis tablica

Prilozi

Izvorni kôd prirodne Android aplikacije: https://gitlab.com/Im_Pew/diplomskiappandroid

Izvorni kôd prirodne iOS aplikacije: https://gitlab.com/Im_Pew/diplomskiappios

Izvorni kôd Flutter aplikacije: https://gitlab.com/Im_Pew/diplomskiappflutter

Izvorni kôd Flutter MVC arhitekture: https://gitlab.com/Im_Pew/flutter-mvc