

UNIVERZITET SARAJEVO

ELEKTROTEHNIČKI FAKULTET

RAČUNARSTVO I INFORMATIKA



UBER PLATFORMA

Mentor:
Prof. dr. Vensada Okanović

Student:
Semir Suljević

Sarajevo, septembar 2021.

Univerzitet u Sarajevu

Naziv fakulteta/akademije: Elektrotehnički fakultet

Naziv odsjeka i/ili katedre: Računarstvo i informatika

Izjava o autentičnosti radova

Seminarski rad, završni (diplomski odnosno magistarski) rad za I i II ciklus studija i integrirani studijski program I i II ciklusa studija, magistarski znanstveni rad i doktorska disertacija¹

Ime i prezime: Semir Suljević

Naslov rada: Uber platforma

Vrsta rada: Završni rad prvog ciklusa studija

Broj stranica: 70

Potvrđujem:

- da sam pročitao/la dokumente koji se odnose na plagijarizam, kako je to definirano Statutom Univerziteta u Sarajevu, Etičkim kodeksom Univerziteta u Sarajevu i pravilima studiranja koja se odnose na I i II ciklus studija, integrirani studijski program I i II ciklusa i III ciklus studija na Univerzitetu u Sarajevu, kao i uputama o plagijarizmu navedenim na web stranici Univerziteta u Sarajevu;
- da sam svjestan/na univerzitetskih disciplinskih pravila koja se tiču plagijarizma;
- da je rad koji predajem potpuno moj, samostalni rad, osim u dijelovima gdje je to naznačeno;
- da rad nije predat, u cjelini ili djelimično, za stjecanje zvanja na Univerzitetu u Sarajevu ili nekoj drugoj visokoškolskoj ustanovi;
- da sam jasno naznačio/la prisustvo citiranog ili parafraziranog materijala i da sam se referirao/la na sve izvore;
- da sam dosljedno naveo/la korištene i citirane izvore ili bibliografiju po nekom od preporučenih stilova citiranja, sa navođenjem potpune reference koja obuhvata potpuni bibliografski opis korištenog i citiranog izvora;
- da sam odgovarajuće naznačio/la svaku pomoć koju sam dobio/la pored pomoći mentora/ice i akademskih tutora/ica.

Mjesto, datum: Sarajevo, 2021.

Potpis: Semir Suljević

¹ U radu su korišteni slijedeći dokumenti: Izjava autora koju koristi Elektrotehnički fakultet u Sarajevu; Izjava o autentičnosti završnog rada Centra za interdisciplinarne studije – master studij „Evropske studije”, Izjava o plagijarizmu koju koristi Fakultet političkih nauka u Sarajevu.

Postavka zadatka

U završnom radu potrebno je napraviti softver koji demonstrira u suštini rad Uber kompanije za naručivanje vožnji. Opisati korištene tehnologije pri radu i objasniti njihove osobine. Izlaganje u radu je potrebno poduprijeti slikama i primjerima programskog koda.

Sažetak

Trenutni način života u svijetu konstantno motiviše ljude da kreiraju nove načine uštede vremena, olakšavanje života i pronalaska novih izvora zarade. Potrebno je bilo kreirati način da se momentalno nađe prijevoz od tačke A do tačke B. Svakako da ne postoji bolji način nego integrisati to u uređaje koje koristimo svakodnevno po više sati, telefone.

Napredak tehnologije, posebno obrade podataka i brzine interneta omogućava pa skoro globalni jedinstven način naručivanja vožnji i obavještavanja o statusu dolaska vozača konstantno. Potpuno se izbacuje vrijeme čekanja na ulici, pozivanja dispečera taksi kompanije. Jednim klikom je moguće unaprijed naručiti vožnju i znati tačno vrijeme dolaska prijevoza.

Ovaj pristup se ne primjenjuje samo na naručivanje vožnji, takođe i na druge svakodnevne potrebe ljudi kao što je naručivanje hrane. Razlika između Uber Eats i ostalih dostava je različitost i mogućnost naručivanja iz širokog spektra restorana, ne samo jednog. Ovaj sistem je centralizovan i uklanja posrednika. Bilo ko u svakom trenutku se jednostavno treba prijaviti, ostaviti svoje podatke i ukoliko je podoban, ima šansu zarađivati.

Ključna je jedna mogućnost svih tehnologija, a to je praćenje lokacija svih učesnika. Bez ovoga cijeli ovaj sistem bi bio manje impresivan i koristan. Cijeli proces bi bio jako nedefinisan s dosta mjesta za iskorištavanje i potencijalne nesigurne situacije.

Sadržaj

Uvod	7
1. Tehnologije izrade (technology stack).....	8
1.1. Cross – platform razvoj mobilnih aplikacija	8
1.2. Firebase kao backend	10
2. Dart.....	12
2.1. Historija	12
2.2. Korištenje Dart programskog jezika	13
2.2.1. Kompajliranje kao Javascript (web).....	13
2.2.2. Stand – alone	13
2.2.3. Ahead – of – time kompajliranje	14
2.2.4. Native	14
2.3. Just – in – time kompajler	14
2.4. Pregled najbitnijih cjelina	15
2.4.1. Objektno orijentisano programiranje	16
2.4.2. Null safety.....	18
2.5. Paralelno programiranje.....	21
3. Firebase	23
3.1. Firebase Authentication	25
3.2. Cloud Functions	26
3.2.1. Osobine Cloud Functions.....	26
3.2.2. Lifecycle	27
3.2.3. Koraci implementacije	28
3.2.4. Korištenje u projektu	28
3.3. Cloud Storage	30
3.3.1. Osobine Cloud Storage	30
3.3.2. Funkcionalnost	31
3.3.3. Koraci implementacije	31
3.3.4. Korištenje u projektu	32
3.4. Firebase Cloud Messaging (FCM)	34
3.4.1. Tipovi poruka	35
3.5. Cloud Firestore	38
3.5.1. Osobine Cloud Firestore	38
3.5.2. Firestore Rules	40

3.5.3.	Cijene korištenja	42
4.	Flutter	43
4.1.	Historija	43
4.2.	Deklarativni UI	44
4.3.	Flutter layout – i	45
4.4.	Stateless i Stateful widgets.....	46
4.4.1.	Stateless widget.....	46
4.4.2.	Stateful widget	47
4.4.3.	Životni ciklus widgeta	49
4.5.	State management	50
4.5.1.	Podizanje stanja.....	50
4.5.2.	Lista pristupa	50
4.5.3.	Provider	51
4.6.	Rutiranje	53
4.7.	Integracija FCM i upravljanje s više ulaza u aplikaciju	56
4.8.	Komunikacija s native platformom.....	61
4.9.	Performanse i poređenja.....	64
4.10.	Pripremanje animacija	68
5.	Zaključak.....	69
6.	Literatura	70

Uvod

Uber Technologies, Inc., poznatiji kao **Uber**, je američka tehnološka kompanija. Uber nudi usluge prijevoza putnika, dostave hrane (Uber eats), prijevoza robe i dobara i uz partnerstvo s kompanijom **Lime**, iznajmljivanje električnih bicikala i skutera. Sjedište kompanije se nalazi u San Franciscu i kompanija broji aktivnosti u više od 900 metropola.

Uber je osnovan 2009. pod imenom **Ubercab** od strane programera **Garett Camp**, U maju 2010., izbačena je beta verzija Uber servisa i mobilnih aplikacija, a oficijalno je lansirana u San Franciscu 2011. Danas Uber usluge koristi preko 93 miliona korisnika mjesečno širom cijelog svijeta [11].

Primarni način korištenja servisa Ubera je koristeći njihove mobilne aplikacije. Korisnici kreiraju lični račun s osnovnim podacima kao i preferiranim načinima plaćanja usluga. Za sve usluge koriste se dvije (u ovom slučaju) mobilne aplikacije. Jednu koriste vozači / dostavljači, dok drugu koriste klijenti koji žele naručiti ili vožnju ili određenu vrstu dostave. Nakon što je proces dostave završen, korisnicima je omogućeno **dodatno nagraditi** vozača / dostavljača.

Uber nudi jako širok spektar usluga s kompleksnom poslovnom logikom kao i naprednim softverskim tehnologijama. U ovom završnom radu obrađeno je usluga prijevoza putnika. Napravljene su dvije mobilne *cross – platform* aplikacije u kojoj s jedne strane putnici imaju uvid u svoj lični nalog, vozače koji trenutno rade, komunikaciju s istim kao i naručivanje vožnji. Vozači svoju aplikaciju koriste na sličan način s tim da aplikacija za vozače ima manje mogućnosti. Koristeći širok spektar tehnologija, cilj je omogućiti putnicima što brži i jednostavniji način prijevoza s jedne lokacije na drugu.

1. Tehnologije izrade (technology stack)

Na samom početku izrade sistema, potrebno je izabrati odgovarajuće alate za izradu potrebnih dijelova. U ovom poglavlju će se razmotriti potrebne tehnologije za izradu sistema, uzimajući u obzir njihove mogućnosti i karakteristike. Za izradu ovog sistema bilo je potrebno razmotriti dostupne tehnologije za sljedeće:

- mobilna aplikacija
- baza podataka
- servisi za slanje *push* notifikacija
- autentifikacija i autorizacija korisnika
- skladištenje korisničkih slika
- *real time* komunikacija
- prikaz mapa na mobilnim uređajima
- serveri za komunikaciju s klijent uređajima

Ovo sve djeluje neizvodivo za jednu osobu u kratkom vremenskom periodu. Pa čak i ako uklonimo vremenski faktor, da li je nešto ovako prevelik zadatak za jednu osobu? Pravljenje ovakvog sistema je izazov za velike timove iskusnih ljudi. Bilo je potrebno pronaći alate koji obavljaju više od jedne stavke iz pobrojane liste. Kako će se pokazati dovoljno je bilo uzeti samo dvije tehnologije koje će zadovoljiti sve potrebe: **Flutter i Firebase**. Flutter i Firebase su relativno mlade tehnologije koje je lansirao Google i koje zajedno savršeno rade i predstavljaju jednu od najučestalijih kombinacija u razvoju mobilnih aplikacija. Flutter se koristi za razvoj mobilnih aplikacija, a sve ostale stavke iz liste Firebase. U nastavku ćemo se ukratko upoznati za šta služe i kako predstavljaju možda najoptimalnije rješenje za kreiranje što povoljnijeg konačnog proizvoda.

1.1. Cross – platform razvoj mobilnih aplikacija

Što se tiče kreiranja mobilnih aplikacija, u suštini nemamo puno izbora. Za **android** postoji samo dvije opcije, korištenjem **Java** i **Kotlin** programskih jezika, dok je za **iOS**

moгуće koristiti **Swift** ili **Objective-C**. Međutim kako je već spomenuto, razvijanje više od jedne aplikacije u kratkom vremenskom roku za jednu osobu bi bilo skoro nemoguće tako da je potrebno okrenuti se *cross – platform* tehnologijama.

Ni u ovom polju ne postoji mnogo opcija, a dvije najkorištenije tehnologije su Google-ov Flutter i Facebook-ov **React – Native**. Uzeći u obzir da sam prije odabira teme završnog rada već poznao rad u Flutter-u odlučio sam se za tu opciju.

Flutter je Google-ov **UI** (*user – interface*) set alata za kreiranje mobilnih, web i desktop aplikacija koristeći isti kod za sve pobrojane platforme. Od zadnje verzije, 2.0, u Flutter-u je moguće pisati aplikacije čak i za ugrađbene sisteme.

Međutim, šta može predstavljati problem pri korištenju *cross – platform* tehnologiji pri razvoju mobilnih aplikacija? Performanse. Dugo su se ove tehnologije smatrale kao lošim zbog loših, pa može se reći i užasnih performansi. Ovi stavovi su uveliko promijenile pojavom, prvo React – Native-a pa onda i Flutter-a. Ukratko će se opisati zašto je izabran Flutter, a ne React – Native.

Mogućnost izvršavanja jednog koda na više platformi nije jedina prednost u odnosu na konkurenciju. Flutter je apsolutni lider po pitanju performansi u polju *cross – platform* razvoja mobilnih aplikacija. Flutter koristi **Dart** programski jezik, također razvijen od strane Google-a i upravo u tome leži odgovor na pitanje zašto je Flutter lider po pitanju performansi. Dart je *natively compiled* programski jezik što doprinosi performansama programskog jezika u mnogome.

Samim tim je i Flutter *natively – compiled* te je u mogućnosti da pruži performanse jako blizu onim koje pružaju *native* tehnologije. Čak i štaviše, u poređenju sa iOS *native* razvojem, Flutter pruža iste performanse u skoro svim slučajevima (postoje određeni izuzeci za koje se Flutter još nije dovoljno razvio), u nekim čak i brže od Swifta što je do sada bilo nemoguće bilo koristeći bilo koji *cross – platform framework*. Mnogo više o Flutter-u i performansama u sljedećim poglavljima.

1.2. Firebase kao backend

Na početku ovog poglavlja spomenuto je kako je potrebno pronaći alat koji može obavljati više funkcija. Potrebno je bilo pronaći *Backend – as – a – service (Baas)*. Firebase je takođe proizvod Google-a i pruža veliki niz usluga s dosta inovativnih rješenja. Neke od najkorištenijih usluga koje pruža Firebase:

- Firebase Authentication
- Cloud Firestore
- Cloud Storage
- Realtime Database
- Google Analytics
- Crashlytics
- ML kit
- Remote Config
- Cloud Messaging
- Cloud Functions
- Hosting
- Performance Monitoring
- Test Lab

To su samo neke od najčešće korištenijih usluga koje nudi Firebase. Šta još predstavlja prednost jeste da su svi Firebase projekti automatski povezani s **Google Cloud** platformom, što otvara čitav novi set mogućnosti korištenja u klijent aplikacijama. Kako se uklapa Firebase u sve ovo? Firebase spada u *serverless* backend-e koji izbacaju server kao posrednika između klijent aplikacije i baze podataka. Kako to spriječava zlonamjerne napade? Firebase u tu namjenu piše veliki broj *client – side* library da se omogući client aplikacijama direktan pristup bazi podataka odakle i dolazi naziv *serverless*. Takođe svaka od baza podataka posjeduje tzv. **pravila** koji ograničavaju pristup korisnicima.

Firebase nudi dvije baze podataka, **Cloud Firestore** i **Realtime Database**. Obje baze su **NoSQL** i real time baze podataka, što omogućava klijentima širok spektar funkcionalnosti bez kreiranja dodatnih servera. Upravo to je jedan od glavnih razloga zašto je Firebase izabran kao backend za ovaj projekat.

Putnicima u aplikaciji je omogućeno prijavljivanje koristeći postojeće račune od Google-a i Facebook-a. Sve to je implementirano koristeći **Firebase Authentication** [2], servis koji besplatno nudi prijave koristeći Google, Facebook, Twitter, Apple, Github, telefonski broj, email, anonimno korištenje i još mnoge druge.

Cloud Storage nudi mogućnost skladištenja fajlova, u našem slučaju korisničkih slika [4].

Cloud Functions po svojoj izvedbi dosta liče na kreiranje servera. Koristi *event – driven* logiku koja omogućava jednostavan pristup podacima nakon dodavanja/brisanja i ažuriranja u bazu, kao i fajlovima [5]. Zajedno s **Cloud Messaging** servisom koriste se za slanje push notifikacija na mobilne uređaje korisnika. Cloud Messaging je potpuno besplatan, što je bitno napomenuti [3]. Takođe moguće je praviti API-e za cijeli backend sistem koristeći Cloud Functions.

Spomenuto je da je svaki Firebase projekat povezan s Google Cloud platformom, što omogućava korištenje **Google Maps SDK** za mobilne uređaje, kao i **Places Autocomplete** za pretragu lokacija.

Koristeći real time bazu podataka, Cloud Firestore, lagano je bilo omogućiti dopisivanje između putnika i vozača, kao i praćenje lokacija vozača na mapi. Zajedno s ostalim servisima Firebase-a mobilne aplikacija ima sve što je potrebno da radi.

2. Dart

Dart je Google-ov programski jezik dizajniran za klijentski razvoj *cross – platform* aplikacija. Pored klijentskih aplikacija, moguće ga je koristiti i za razvoj servera. Dart je objektno orijentisani programski jezik, koji koristi *garbage – collection* i ima tzv C-ovsku sintaksu. Kompajlira se direktno u mašinski kod ili u **Javascript** kod.

Cilj kreatora Dart-a jeste veoma produktivno programiranje na svim platformama. U samim ranim fazama razvoja ovog jezika, cilj je bio stvoriti jezik koji će pružiti što bolje zadovoljstvo korisnicima, imajući na umu i performanse, koje su postignute zahvaljući njegovom *native* kompajliranju. Popularnost Dart-a je uveliko porasla 2017. godine kada ga je Google počeo koristiti kao fondaciju Flutter-a. Od tada, razvoj Dart-a raste konstantno i počinje se koristiti i kao *server - side* programski jezik, čak i u ugradbene sisteme.

Dart je statički tipovani programski jezik, što znači da tip vrijednosti varijable odgovara varijablinom statičnom tipu. Bitno je naglasiti da Dart podržava i dinamičko određivanje tipa varijabli.

Da bi se osigurao što sigurniji rad aplikacija, što manje greški, Dart je kao i drugi jezici uveo *sound null safety* [7]. *Null safety* predstavlja način programiranja u kojem vrijednosti varijabli ne mogu biti nedefinisane ukoliko korisnik eksplicitno ne navede da to mogu biti. Takođe to povlači činjenicu da nije moguće koristiti vrijednost tako definisanih varijabli ukoliko se ne provjeri da li im je prije korištenja dodijeljena vrijednost. Za razliku od *null safety* sistema u drugim jezicima, u Dart programskom jeziku nije moguće varijablu koja je deklarirana kao *nullable* kasnije proglasiti kao *not nullable*.

2.1. Historija

Dart je prvi puta predstavljen javnosti na **GOTO** konferenciji održanoj u Danskoj u oktobru 2011. godine. Projekat je osnovan od strane **Lars Bak-a** i **Kasper Lund-a**. Prva stabilna verzija, 1.0, objavljena je 14. novembra 2013. godine [10].

Inicijalno Dart je bio zamišljen kao zamjena Javascript-u u web pretraživačima. Planovi su bili da se napravi Dart virtuelna mašina za Chrome. Nakon kritika da Google fragmentira web, ti planovi su odbačeni 2015.-te godine s verzijom 1.9 i prešlo se na podršku na kompajliranje Dart koda u Javascript kod. U augustu 2018. – te godine s verzijom 2.0 Dart postaje statički tipovski jezik, ali ostaje podrška i za pisanje koda kao u dinamičkim jezicima.

Verzija 2.6 Dart programskog jezika donosi i *dart2native*. S ovom verzijom Dart osim podrške za *native* kompajliranje na Android i iOS platformi, uključuje isto i za Linux, macOS i Windows desktop platforme. Do tada, moguće je bilo samo kreirati aplikacije za mobilne platforme s *native* kompajliranjem.

2.2. Korištenje Dart programskog jezika

Kako je već spomenuto, Dart je namijenjen za korištenje na svim platformama. To povlači pitanje kako pokretati napisane aplikacije na različitim platformama [10]. Postoje 4 načina pokretanja Dart koda:

- Kompajliranje kao Javascript (web)
- Stand – alone
- Ahead – of – time kompajliranje
- Native

2.2.1. Kompajliranje kao Javascript (web)

Da bi se koristio u glavnim web pretraživačima, Dart se oslanja na *source – to – source* kompajler u Javascript. Tvrdnje ljudi koji stoje iza ovog projekta su da je Dart je dizajniran za lagano pisanje softvera, pogodan za razvijanje modernih aplikacija i u stanju da pruži odlične performanse.

Kada se pokreće Dart programski kod u web pretraživačima, kod se kompajlira u Javascript koristeći *dart2js* [7] kompajler. Kako je Javascript primarni jezik u web pretraživačima od njihove pojave pa sve do danas, to Dart čini kompatibilnim sa svim poznatijim pretraživačima bez potrebe da se pretraživači prilagođavaju Dart programskom jeziku. Nakon kompajliranja u Javascript i optimiziranja finalnog koda, uklanjanjem bespotrebnih provjera i velikih operacija, moguće je, u određenim situacijama, dobiti brže izvršavanje nego koristeći Javascript pisani kod.

2.2.2. Stand – alone

Dart *software development kit* (SDK) dolazi s Dart virtuelnom mašinom, što omogućava izvršavanje Dart programskog koda u *command – line* interfejsima. Budući da su

alati Dart programskog jezika su pisani većinom u Dart-u, Dart virtuelna mašina [7] predstavlja veoma bitan dio SDK-a. Ovi alati uključuju spomenuti *dart2js* kompajler i *package manager pub* [9]. Pored toga, uključena je standardna biblioteka koja omogućava pisanje sistemskih aplikacija, poput web servera.

2.2.3. Ahead – of – time kompajliranje

Dart programski kod može biti *Ahead – of – time* (AOT) kompajliran u *native* izvršive djelove koda [7]. Aplikacije napravljene u Flutter-u, u *release* verzijama koriste AOT kompajlirani kod.

2.2.4. Native

Dart 2.6 dolazi s *dart2native* [7] kompajlerom koji omogućava *native* kompajliranje za sve platforme. Prije ove verzije ovo je bilo moguće samo za Android i iOS platforme.

2.3. Just – in – time kompajler

Prilikom razvoja softvera, ključna je brzina razvoja i produktivnost programera. Već je spomenuto da Dart koristi AOT kompajler za aplikacije koje su spremne za izvršavanje. Međutim, prilikom razvoja korištenje AOT kompajlera ne bi doprinjelo brzini izrade softvera. S tim u vezi, Dart virtuelna mašina dolazi s još jednom vrstom kompajlera, a to je **Just – in – time** (JIT) kompajler. Ova vrsta kompajlera nudi inkrementalnu rekompilaciju programa i time omogućava korištenje **hot reload**. *Hot reload* omogućava ažuriranje izmjena koda bez ponovnog pokretanja za manje od sekundu. Manje promjene se dešavaju skoro neprimjetno što uveliko pomaže pri razvoju softvera i dovodi do boljeg *development experience*.

Već sada se može primjetiti kako je cijeli ekosistem oko Dart programskog jezika zasnovan na velikoj produktivnosti, lagano prilagođavanje i odličnom iskustvu pri radu s ovim programskim jezikom. Međutim, moguće je primjetiti da dosta drugih jezika nema ni približno ovoliko mogućnosti što povlači pitanje kako i zašto Google i Dart timovi ovo rade. Činjenica je da izvršivi kod koji koristi AOT kompajlere, u većini slučajeva brži od onih koji koriste JIT. Takođe razvijanje velikog broja alata, pogotovo AOT kompajlera, zahtjeva veliki broj resursa, kako ljudi tako i novca. Google je u stanju priuštiti sve ovo dok ostali kreatori

jezika nisu, možemo pogledati primjer Python programskog jezika koji nema AOT kompajler iako je njegova glavna osobina odlične performanse.

2.4. Pregled najbitnijih cjelina

Već je spomenuto da je Dart programski jezik temelj na kojem je izgrađen Flutter framework. Pored dijelova koda poput model klasa, enumeracija čija se izvedba u Dartu ne razlikuje puno, skoro nikako, od ostalih jezika, u ovom projektu su korišteni i dijelovi koda koji su jedinstveni za Dart programski jezik, kako sintaksom tako i postojanjem odgovarajućih tipova podataka i pristupa rješavanju problema.

U prethodnim poglavljima, objašnjen je pojam null safety. Iako null safety nije neophodan za korištenje, njegova funkcionalnost tjera određenu dozu discipline pri programiranju. Važnost null safety se ogleda najviše pri asinhronom programiranju u ovom projektu i odigrala je važnu ulogu u osiguravanju ispravnog rada aplikacije. Prilikom prikazavanja slika korisničkih računa, moguće je da neka slika nije učitana iz cache memorije ili čak ni da ne postoji u memoriji. Tu nam pomaže null safety jer eksplicitno moramo provjeriti da li je odgovarajuća slika spremna za prikazivanje. Ukoliko nije, u općem slučaju prikaže se poruka korisniku da se radi na pribavljanju potrebnih podataka. Iako je cijelu aplikaciju bilo moguće napisati bez ove funkcionalnosti, odlučeno je da se ona obradi kako bi se pokazala važnost pisanja sigurnog koda i kako se to sve uklapa s drugim cjelinama Darta.

Pri pisanju ovako velikih projekata, jako je bitno na početku držati se korektne organizacije fajlova koda, raspoređivanja fajlova u foldere i koristiti određene **patterne** pri programiranju. Potrebno je kreirati što univerzalnije rješenje koje će biti podložno naknadnim promjenama. Praćenje pravila koje olakšavaju organizaciju unutar velikih projekata, vođenje računa o performansama, enkapsulacija i notacija.

Neizbježna cjelina koja je sastavni dio skoro svakog programskog jezika jeste asinhrono programiranje. Asinhrono programiranje u suštini predstavlja dobavljanje, u općem slučaju, neke vrste resursa koje nije moguće pročitati direktno iz programskog koda. Kako su napisane aplikacije u ovom projektu veoma zavisne od komunikacije s bazama podataka, drugim korisnicima, međusobnom komunikacijom između korisnika, asinhrono programiranje ima veliki udio u ovom projektu. Dart ima fenomenalnu podršku za asinhrono programiranje

koja se savršeno uklapa s Firebase-om. Da rezimiramo, cjeline Dart programskog jezika koje su najviše korištene i predstavljaju okosnicu ovog projekta su:

- objektno orijentisano programiranje
- null safety
- const i final
- podrška asinhronom programiranju

2.4.1. Objektno orijentisano programiranje

Sve što se u Dartu može smjestiti u varijable posmatramo kao objekat i svaki objekat je instanca neke klase. Najprostiji tipovi poput brojeva, funkcija i null vrijednosti su objekat. Svi objekti su naslijeđeni iz klase Object, s izuzetkom null vrijednosti ukoliko se koristi null safety.

Dart je objektno orijentisani jezik s klasama i **mixin** baziranim nasljeđivanjima. Svaki objekat je instanca neke klase i sve klase osim **Null** klase su nasljeđene i klase **Object**. Mixin bazirano nasljeđivanje znači da iako sve klase su nasljeđene iz bazne Object klase, tijelo klase može biti korišteno u više klasnih hijerarhija.

Kreiranje instanci neke klase vrši se uz pomoć konstruktora kao i u ostalim objektno orijentisanim jezicima. Postoji više konstruktora u Dart programskom jeziku i kao jedna razlika u odnosu na jezike poput Java, C++, moguće je definisati samo jedan neimenovani (generativni) konstruktor. Dakle, korištenje konstruktora (generativnog) s različitim argumentima nije dozvoljeno. U tu svrhu, Dart koristi **imenovane** (named) konstruktore koji nemaju ograničenja na vrstu i broj argumenata. Interesantno je da koristeći imenovane konstruktore moguće je zadati identične argumente u 2 konstruktora, što nije izvodivo u Javi. Možemo to pogledati na jednostavnom primjeru klase Margine:


```

class Margine {
    double lijevo, desno, gore, dole;

    Margine.simetricne(double horizontalne, double vertikalne) {
        lijevo = desno = horizontalne / 2;
        gore = dole = vertikalne / 2;
    }

    Margine.samoHorizontalne(double lijevo, double desno) {
        this.lijevo = lijevo;
        this.desno = desno;
        gore = dole = 0;
    }
}

```

Isječak koda 1 – korištenje imenovanog konstruktora

Vidimo na jednostavnom primjeru kako smo u mogućnosti poslati iste podatke kao argumente a dobiti različitu inicijalizaciju atributa klase Margine. Ova osobina Dartu doprinosi čitljivosti i jednostavnosti programa. Kao i ostali objektno orijentisani jezici Dart kreira podrazumijevani konstruktor bez parametara.

Ukoliko kreiranu instance neke klase nećemo koristiti na mjestu kreiranja, Dart nudi i opciju **factory** konstruktora koji *vraća* kreiranu instancu klase na mjestu poziva. Poželjno ga je koristiti ukoliko kreiramo veliki broj instanci neke klase zbog performansi. Možemo ga zamisliti kao statičnu metodu koja vraća instancu koristeći neki od predefinisanih konstruktora.

Još jedna opcija kod objektno orijentisanog programiranja u Dartu su **mixini**. Mixini omogućavaju korištenje koda neke klase u više klasnih hijerarhija. Mixin predstavlja svojevrsnu vrstu interfejsa u ostalim jezicima s par razlika. Mixin može sadržavati atributi koje je moguće naknadno mijenjati i set metoda.

Dart ne posjeduje ključne riječi za access modifiere nego koristi donju crtu (`_`) i atributi i metode čije ime počinje s donjom crtom su vidljivi samo unutar biblioteke u kojoj su definisani.

2.4.2. Null safety

Omogućavanjem null safety tipovi postaju non – nullable kao zadani, ne mogu biti bez vrijednosti osim ako mi to eksplicitno ne naglasimo. Gdje vidimo da null safety pomaže u radu? Sve **runtime** greške koje nastaju pristupanjem null vrijednosti, sada su **edit – time** greške. Ne govorimo ni o **compile time** greškama nego o edit – time. Dakle prije samog pokretanja programa imamo uvid u situaciju u kodu gdje smo pristupili nekoj varijabli koja može imati nedefinisanu vrijednost, a da prije toga nismo provjerili da li je toj varijabli dodijeljena neka vrijednost. Da bi dozvolili da neka varijabla može imati nedefinisanu vrijednost stavljamo upitnik (?) pored njenog tipa.

Rečeno je već da je aplikacija veoma zavisna od vanjskih resursa koji su pohranjeni van koda programa. Ukoliko uzmemo primjer slike profila koja se prikazuje na početnom ekranu aplikacije, možemo odma uvidjeti zašto je null safety od velike pomoći programerima. Slika profila je keširana u folderu aplikacije i pri otvaranju aplikacije potrebno je učitati svaki put. Za prikazivanje slike učitane iz keša u Flutter – u, koristimo Dart tip podatka **File**.

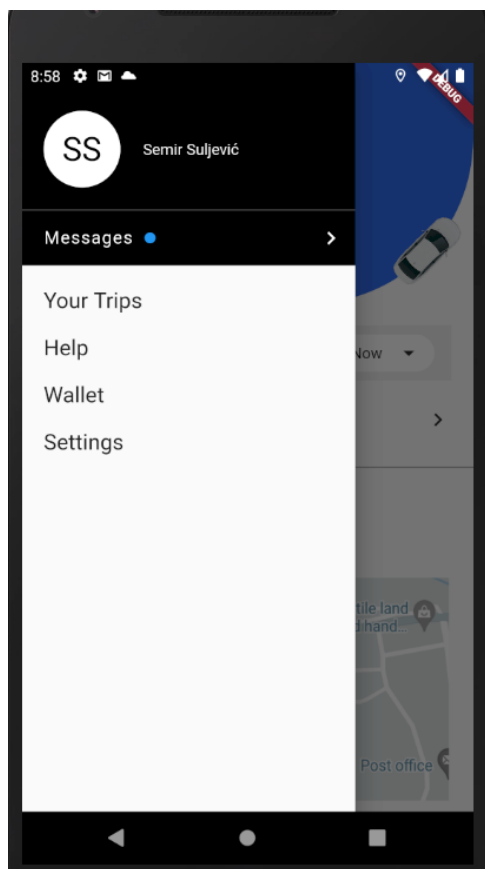
```
Future<File?> loadUserPicture() async {  
  final Directory temp = await getTemporaryDirectory();  
  final File profilePicture = File('${temp.path}/${FirebaseService.id}');  
  
  //profile picture is in the temp directory  
  if(await profilePicture.exists()) {  
    return profilePicture;  
  }  
  //picture does not exist, return null  
  return null;  
}
```

Isječak koda 2 – programski kod koji čita sliku profile korisnika

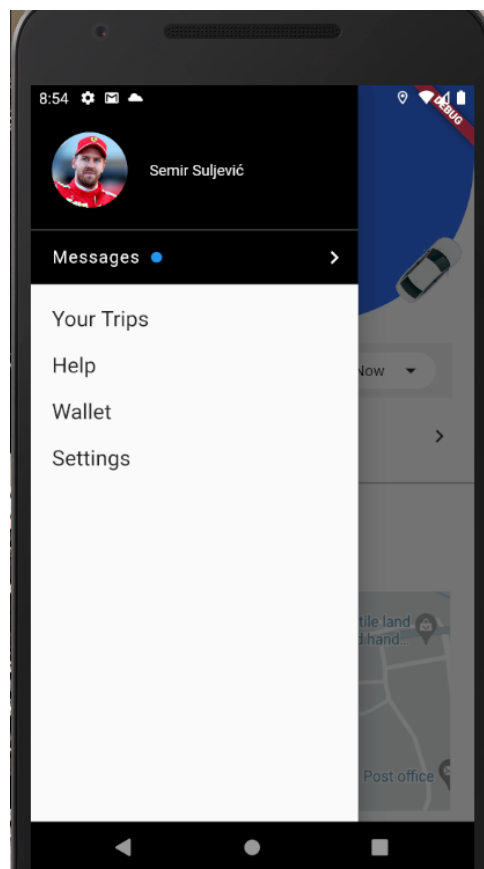
Iz foldera aplikacije, nalazimo putanju gdje bi trebala biti smještena slika korisničkog profila, ukoliko postoji. Ukoliko ne postoji vraćamo null referencu da bi pri prijemu znali kako da postupamo. Uloga null safety je ovdje upravo u prototipu funkcije, gdje govorimo da ova funkcija će vratiti rezultat tipa File ili null referencu. Takođe nije bilo potrebno ispitivati da li pročitana slika zaista postoji, to je učinjeno zbog razumljivijeg koda. Ovako to izgleda u stvarnoj aplikaciji.

U svrhe prikazivanja funkcionalnosti, pri nego se počne s dobavljanjem slike profila, sačeka se 15 sekundi da bi se simulirao slučaj u kojem slika profila još nije učitana. U tom

slučaju, prikazuju se inicijali korisnika umjesto slike. Ukoliko podaci korisnika nisu učitani iz keša prikazuje se poruka da se radi na pribavljanju podataka.



Slika 1 – Slika nije učitana na vrijeme



Slika 2 – interfejs nakon učitavanja

Sada ćemo prikazati kako izgleda Flutter kod koji poziva ovu funkciju i prikazuje odgovarajući interfejs.

```

final File? picture = Provider.of<ProfilePicturesProvider>(context).profilePicture;

return userData == null ?
Container(
  color: Colors.black,
  child: DrawerHeader(
    margin: EdgeInsets.zero,
    padding: EdgeInsets.zero,
    child: Container(
      color: Colors.black,
      child: LinearProgressIndicator(
        backgroundColor: Colors.grey[800],
      )
    ),
  ),
) : Container(
  color: Colors.black,
  child: DrawerHeader(
    margin: EdgeInsets.zero,
    padding: EdgeInsets.zero,
    child: Container(
      decoration: BoxDecoration(
        color: Colors.black,
      ),
      child: Container(
        margin: EdgeInsets.only(top: 20),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.center,
          mainAxisAlignment: MainAxisAlignment.start,
          children: [
            GestureDetector(
              onTap: () async => await Navigator.pushNamed(context,
EditAccount.route),
              child: Container(
                margin: EdgeInsets.only(left: 20),
                child: Row(
                  children: [
                    picture == null ?
CircleAvatar(
                      radius: 35,
                      backgroundColor: Colors.white,
                      child: Text(userData.firstName[0] + userData.lastName[0],
style: TextStyle(fontSize: 30, color: Colors.black)),
                    ):
CircleAvatar(
                      radius: 35,
                      backgroundImage: FileImage(picture),
                      backgroundColor: Colors.transparent,
                    ),

```



Sada ćemo analizirati šta se ovdje tačno događa. U liniji 1 koristimo getter koji nam daje sliku profila iz klase koja je zadužena za upravljanje slikama korisničkih profila. Ova klasa nasljeđuje klasu **Provider** i dostupna je iz svih dijelova Flutter koda (o Provider – u i

patternima nešto kasnije). Za sada je bitno naglasiti da varijabla deklarirana u prvoj liniji koda je *listener* za varijablu koja predstavlja sliku korisničkog profila.

Ukoliko pogledamo liniju označenu strelicom vidjećemo ternarni uslov koji omogućava sve ovo. Ukoliko varijabla *picture* i dalje nije spremna za prikaz, prikazujemo avatar s inicijalima korisnika. Inače prikazujemo avatar s učitanoj slikom profila. Kako je već napomenuto varijabla *picture* je *listener* i kada se završi s učitavanjem, vrijednost varijable se automatski ažurira i Flutter ponovo iscrtava dijelove ekrana koji zavise od varijable *picture*.

Sada je na primjeru prikazano kako *null safety* pomaže u asinhronom programiranju i kako se odlično uklapa s Flutter-om i *reactive* prirodi framework-a. Međutim šta se desi ukoliko nije provjereno da li je varijabla spremna za prikaz? To više nije runtime greška nego compile time greška, što znači da nećemo biti ni u mogućnosti pokrenuti ovaj program dok se ne uvjerimo da je varijabla spremna za prikaz.

2.5. Paralelno programiranje

Mobilni uređaji imaju mnogo slabiji hardver od računarskih konfiguracija tako da je uvijek bitno izvući maksimalne performanse iz svakog programa. U mobilnom programiranju razlikujemo glavnu nit ili *user interface* nit koju hardver koristi za crtanje komponenti na ekranu. Svaki moderni procesor posjeduje viši niti da bi se osiguralo bolje performanse. Zašto je ovo bitno? Ovo je bitno iz razloga da se crtanje može vršiti samo na glavnoj niti i može doći do neželjenih pojava ukoliko osim crtanja po ekranu uređaja se izvršavaju neke akcije koje zahtjevaju dosta hardverskih resursa.

Na glavnoj niti se u suštini izvršavaju sve komande našeg programa. Bile one vezane za iscrtavanje korisničkog interfejsa ili ne. Ukoliko neka komanda izvršava neku aktivnost koja će potencijalno zatražiti mnogo resursa može doći do **blokiranja** glavne niti i što u konačnici dovodi do loših performansi crtanja komponenti na ekranu, tzv. *jank*.

Dart spada u grupu *single – threaded* programskih jezika. Dart program dobije svoj dio memorije i izvršava sve komande na jednoj niti i tom djelu memorije. Po tome je isti kao Javascript i posjeduje tzv. *event loop*. Međutim kao i Javascript, Dart podržava višenitno programiranje koristeći *Isolate*.

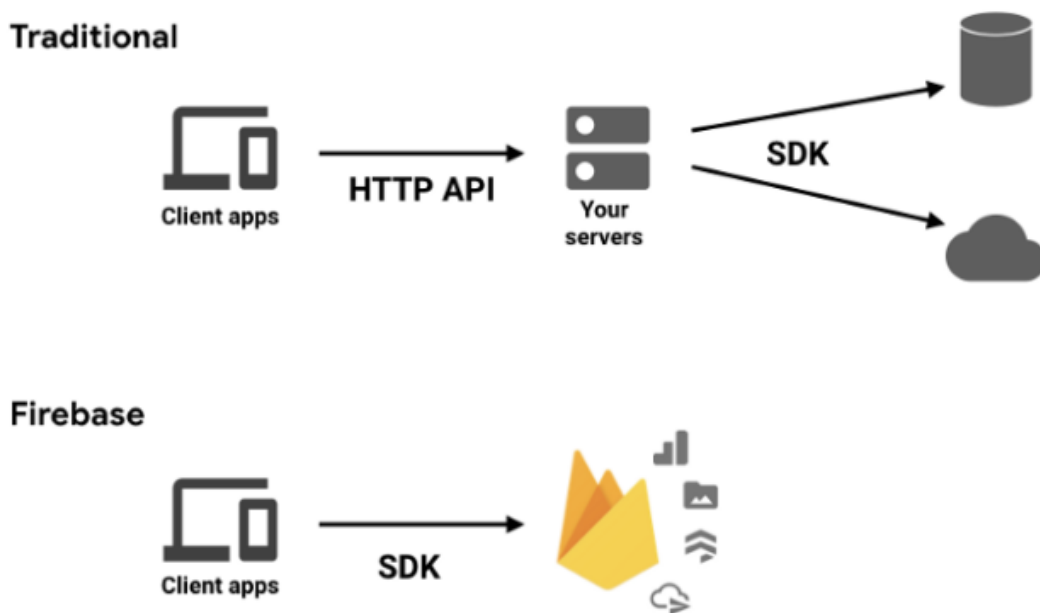
Jezici poput C++, Jave dijele memoriju između svih niti za razliku od Darta. Svaka nit dobija svoj dio memorije koji samo ta nit koristi. Komuniciranje između niti vrši se razmjenom asinhronih poruka primitivnih tipova.

Koristeći višenitno programiranje, moguće je osloboditi glavnu nit od izvršavanja komandi koje mogu izazvati blokiranje glavne niti i tako osigurati maksimalne performanse. Oficijalna dokumentacija ne preporučuje često korištenje višenitnog programiranja ukoliko se ne radi o baš zahtjevnim operacijama poput čitanja velikih fajlova, matematičkih komputacija, obradi slika i slično. Po preporukama dokumentacije i prirodi aplikacije, u projektu nije korišteno višenitno programiranje.

3. Firebase

Firebase je Google-ova platforma za razvoj mobilnih i web aplikacija koja doprinosi brzom razvoju, napretku i rastu aplikacija. Nudi programerima gotove i spremne za korištenje alate koje bi inače morali sami praviti iz temelja. Svi alati se nalaze u Google-ovim serverima i Google je odgovaran za njihovo održavanje. Nudi veliki broj klijentskih SDK koji komuniciraju direktno s alatima bez potrebe ubacivanja *posrednika* između klijentskih aplikacija i backenda.

Ovo se razlikuje od tradicionalnog razvoja aplikacija koje obično uključuju razvoj frontend i backend softvera. S Firebase alatima, tradicionalni backend je izbačen i radi se u potpunosti na razvoju klijentskih aplikacija. Administrativni pristup svakom od alata je moguće koristeći **Firebase konzolu**. Ovakav pristup omogućava backend dijelovima softvera da se potpuno izbací. Ovakav pristup razvoja aplikacija naziva se **serverless**. Firebase prati dva plana plaćanja, a to su Free i Blaze (pay as you go).



Slika 3 – Dijagram tradicionalnog i serverless pristupa²

Moguće je iz ovoga zaključiti da proizvodi kao Firebase potpuno eliminišu profesiju backend developera. Međutim to nije istina iz mnogo razloga i potpuno je moguće imati

² Izvor: <https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0>

backend razvoj unutar Firebase-a. Ovo je moguće iz više razloga jer je server jedina sigurna okolina i jednostavno nije moguće osigurati dovoljnu sigurnost koristeći samo frontend pristup. O tome nešto više kasnije.

U pisanom članku koji je referenciran na slici 4, navedeno je da u vrijeme pisanje članka na korištenje spremno 17 proizvoda. U vrijeme pisanja ovog završnog rada broj proizvoda je čak **27**. To je pokazatelj da Firebase napreduje kao i da potražnja za ovom platformom sve više raste. Slijedi prikaz svih proizvoda koje Firebase nudi unutar konzole.

Build grupa proizvoda uključuje:

- Authentication – identitet i prijava korisnika
- Realtime Database – realtime, cloud hosted, NoSQL baza podataka
- Cloud Firestore – realtime, cloud hosted, NoSQL baza podataka
- Cloud Functions – event driven backend unutar „serverless“ okruženja
- Firebase Hosting – globalni web hosting
- ML Kit – SDK za uobičajeno ML korištenje

Release & Monitor grupa:

- Crashlytics – koncizan uvid u probleme pri radu aplikacija
- Performance Monitoring – metrike performansi pri radu aplikacija
- Test Lab – skalabilno i automatizovano testiranje, na cloud hosted uređaja
- App Distribution – distribucija verzija aplikacije na određene uređaje testera

Engage grupa:

- Predictions
- A / B Testing
- Cloud Messaging
- In – App Messaging
- Remote Config
- Dynamic Links
- AdMob

Analytics grupa:













- Dashboard
- Realtime
- Events
- Conversions
- Audiences
- Funnels
- Custom Definitions
- Latest Release
- Retention
- DebugView

Od ovih 27 proizvoda, u projektu je korišteno 5 a to su: **Firestore Authentication, Cloud Functions, Cloud Storage, Cloud Messaging i Cloud Firestore** i u nastavku ćemo reći nešto više o njima.

3.1. Firestore Authentication

Firestore Authentication alat koriste se za prijavljivanje korisnika i njihovu identifikaciju. Ovaj alat je ključan za većinu aplikaciju i pomaže pri korištenju ostalih alata na pravi način. Najviše u ograničavanju pristupa podacima određenim korisnicima jer bitno je imati na umu da je ovo serverless pristup i ne postoji način da se zabrani pristup, bar ne na konvencionalni način [2].

Glavna karakteristika ovog alata jeste da je vrlo jednostavno uspostaviti sigurno prijavljivanje korisnika, što je mnogo riskantno i teško praviti za programere u svojim aplikacijama. Omogućeno je **12** različitih načina prijavljivanja, prikazanih na slici 4.

Provider	Status
 Email/Password	Enabled
 Phone	Disabled
 Google	Enabled
 Play Games	Disabled
 Game Center	Disabled
 Facebook	Enabled
 Twitter	Disabled
 GitHub	Disabled
 Yahoo	Disabled
 Microsoft	Disabled
 Apple	Disabled
 Anonymous	Disabled

Slika 4 – Sign in provideri za Firebase Authentication

Bitno je naglasiti da je korištenje svih providera potpuno besplatno i nema ograničenja na broj korisnika osim autentifikacije putem telefona, zavisno od plana. Ukoliko se koristi besplatni plan, moguće je 10 hiljada besplatnih verifikacija telefona. Ukoliko se koristi *pay as you go* plan, verifikacija telefona iznosi 0.01 ili 0.06 dolara, zavisno o regiji.

3.2. Cloud Functions

Cloud Functions za Firebase je serverless framework koji omogućava automatsko izvršavanje backend koda nakon nekih događaja omogućenih od strane drugih Firebase alata i HTTPS zahtjeva. JavaScript ili TypeScript kod je smješten u Google Cloud-u i izvršava se u sigurnom okruženju [5]. Cloud Functions izbacaju potrebu za konvencionalnim serverima koje više nije potrebno održavati i skalirati.

3.2.1. Osobine Cloud Functions

1. Integrisane unutar Firebase platforme:

Funkcije mogu biti pozvane kao rezultat raznih događaja unutar Firebase platforme, kao i Google Cloud platforme kao što su Firebase Authentication trigeri, Cloud Storage trigeri i razno.

Korišćeći **Admin SDK** zajedno s Cloud Functions, integrisati s *third – party* servisima i na taj način implementirati webhooks za razne slučajeve korištenja. Minimiziraju boilerplate kod, olakšavajući korištenje Firebase i Google Cloud platforme unutar funkcija.

2. Održavanje od strane Google-a

Deploy-anje JavaScript ili TypeScript koda na Google-ove servere se radi jednom naredbom iz komandne linije. Nakon toga, Firebase automatski vrši skaliranje potrebnih resursa da bi se zadovoljile potrebe korištenja. Nije potrebno upravljanje sigurnošću aplikacija, konfiguraciji servera, zakupljivanjem novih servera ili ukljanjanje starih.

3. Biznis logika ostaje privatna i osigurana

U dosta slučajeva, programeri žele kontrolišu biznis logiku na serverima da bi osigurali moguće napade s klijentske strane. Takođe nekada je potrebno zaštititi kod od reverse inženjeringa. Cloud Functions su potpuno izolirane od klijenta, tako da je sigurnost potpuno osigurana.

3.2.2. Lifecycle

1. Nakon pisanja koda, odabere se event provider (poput Cloud Firestore) i definišu se uslovi pod kojim bi se funkcija trebala pozvati.
2. Deploy-anje funkcije
3. Kada neki od event providera kreira događaj koji odgovara uslovima neke od funkcija, kod se izvršava
4. Ako je funkcija zauzeta izvršavanjem više događaja, Google kreira više instanci da bi osigurao brže izvršavanje. Ako je funkcija u stanju mirovanja, nekoristene instance se čiste.

5. Kada se ažurira kod funkcija, instance starijih verzija se brišu zajedno s kreiranim artifaktima u Cloud Storage-u i Container Registry-u i mijenjaju se novim instancama.
6. Kada se obriše funkcija, sve instance i zip arhive se obrišu, zajedno s vezanim kreiranim artifaktima u Cloud Storage-u i Container Registry-u. Veza između funkcije i event providera se uklanja.

3.2.3. Koraci implementacije

1. Postavljanje okruženja za pisanje Cloud Functions
Instaliranje Firebase CLI i inicijalizacija unutar Firebase projekta.
2. Pisanje funkcija
3. Testiranje funkcija
Korištenje lokalnog emulatora za testiranje funkcije, da bi se izbjeglo deploy-anje nakon svake promjene.
4. Deploy-anje i praćenje rada

3.2.4. Korištenje u projektu

Cloud Functions se uklapaju super u cijelu **reactive programming** paradigmu i event – driven NodeJS okruženje. U ovom dijelu će se ovrnuti na to kako se koristi ovaj alat da se jednostavno uspostavi statistika cijelog sistema i obavijeste putnici i vozači o informacijama o aktivnostima koje se trenutno tiču njih.

Nakon što putnik pošalje zahtjev za vožnju, treba uzeti u obzir par stvari:

- obavještavanje vozača koji se nalaze u blizini
- ažuriranje statističkih podataka
- obavještavanje putnika o mogućim odgovorima

Što se tiče obavještavanja vozača taj dio je potpuno realizovan koristeći klijentske SDK.

Ažuriranje podataka treba biti sigurno, nije ispravno dozvoliti ažuriranje statistike s klijentske strane, s toga je to potrebno izvršiti iz sredine koja je sigurna.

Pogledajmo kako je to izvršeno:

```

export const updateRideRequestCounter =
  functions.firestore.document("ride_requests/{requestID}").onCreate( async (snap
shot) => {

    try {
      var today = new Date();

      await admin.firestore()
        .collection("ride_requests")
        .doc(today.getFullYear().toString())
        .collection(months[today.getMonth()])
        .doc(today.getDate().toString())
        .update({
          "request_counter": admin.firestore.FieldValue.increment(1)
        });
      console.log("Succesfully updated request counter.");
      return true;
    }
    catch(err) {
      console.log("Error: ", err);
      return false;
    }
  })

```

Isječak koda 3 – Kod koji ažurira statističke podatke o zahtjevima za vožnju

Na prethodnom isječku prikazan je kod koji ažurira statističke podatke kad god se pošalje zahtjev za vožnju. Funkcija **updateRideRequestCounter** će se pozvati svaki put kada se doda neki dokument u **ride_requests** kolekciju. Statistički podaci iskorištavaju fleksibilnu prirodu Firestore baze i njenih podkolekcija tako da se nalazi polje za svaki dan svakog mjeseca svake godine koje je brojač zahtjeva za vožnju.

Dio baze podataka koji obavlja ovu funkciju izgleda ovako:

uber-clone-db20a	ride_requests	2021
+ Start collection	+ Add document	+ Start collection
account_settings	2020	Aug
chats	2021 >	Jul
driver_locations	YBdj8Qw5uF0ynygTgJ26	Jun
drivers	nXUHTWXNNv2LkYgoxzY4	
ride_requests >	yAAq103dDkIkPP1h5RL9	
tests		
users		
		+ Add field

Slika 5 – Root kolekcija zahtjeva, dokumenti godina i podkolekcije mjeseci unutar godine

2021	Jun	5
+ Start collection	+ Add document	+ Start collection
Aug	1	+ Add field
Jul	2	request_counter: 1005
Jun >	3	
	4	
	5 >	
+ Add field	6	

Slika 6 – Dokumenti dana unutar mjeseci s poljem request_counter

3.3. Cloud Storage

Cloud Storage za Firebase koriste programere da bi omogućili korisnicima da skladište sadržaj poput slika, videa i fajlova generalno. Jednostavan i isplativ servis za skladištenje objekata od strane Google-a [4]. Koristi se uz pomoć SDK koje je obezbjedio Google.

3.3.1. Osobine Cloud Storage

1. Robusne, snažne operacije

Obezbjeđeni SDK alati za Cloud Storage obavljaju upload i download nezavisno od kvalitete internet veze. Upload i download operacije mogu se pauzirati nakon što nestane konekcije i ponovo nastaviti kada uređaj dobije internet konekciju.

2. Velika sigurnost

Implementirano rješenje imajući na umu Firebase Authentication da obezbjedi jednostavnu i intuitivnu autentikaciju developerima. Omogućava korištenje deklarativnog sigurnosnog modela da se ograniči pristup imajući na umu imena fajlova, veličinu, sadržaj i druge metapodatke.

3. Velika skalabilnost

Kreiran je za exabyte veličine. Bez imalo uloženog rada, skalira iz prototipa u produkcijski storage koristeći istu infrastrukturu koju koriste Spotify i Google Photos.

3.3.2. Funkcionalnost

Programeri koriste Cloud Storage SDK za upload fajlova direktno s klijenata. Ako je konekcija loša, klijent u mogućnost da nastavi upload, štedeći vrijeme i promet klijenta. Cloud Storage smješta fajlove u Google Cloud Storage bucket, omogućavajući im pristup i kroz Google Cloud platformu. Takođe moguće je koristeći server – side procesiranje kao što je filtriranje slika ili video transkoding koristeći Google Cloud Storage API. Skalira automatski, što znači da nema potrebe za promjenom provajdera ni u kojem trenutku.

3.3.3. Koraci implementacije

1. Integrisanje Firebase SDK za Cloud Storage

Brzo i efikasno uključuje klijente koristeći Gradle, CocoaPods ili skripte.

2. Kreiranje reference

Referenciranje putanje za fajl, da bi se omogućio upload, download i brisanje fajla.

3. Upload ili download

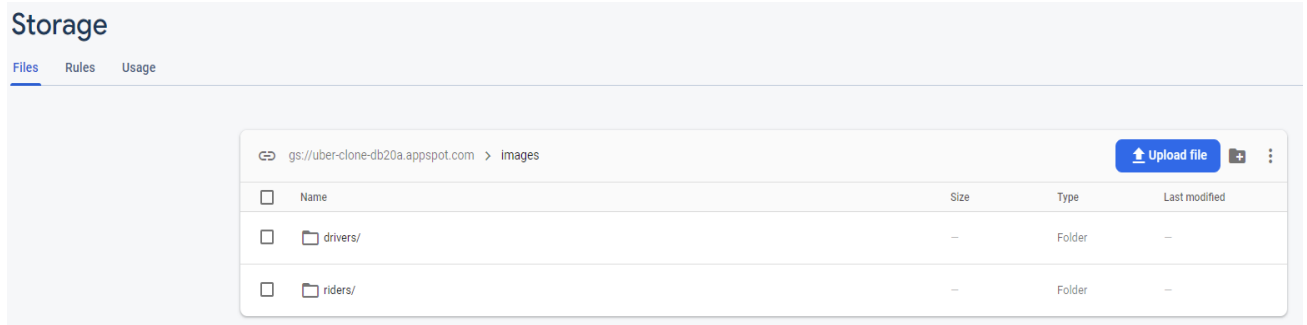
Upload ili download u native tipove u memoriji ili na disku.

4. Osiguravanje fajlove

Osiguravanje fajlova koristeći Firebase Security Rules za Cloud Storage.

3.3.4. Korištenje u projektu

Kao što je spomenuto, Cloud Storage se koristi za skladištenje slika korisničkih profila. Slike se skladište u 2 foldera, **drivers** i **riders**. Ime fajla predstavlja jedinstveni identifikator dodijeljen od strane Firebase Authentication.



Slika 7 – Cloud Storage

Da bi postavili slike na Cloud Storage, potrebno ih je izabrati i poslati s korisničkih uređaja. Referenca na fajl u Cloud Storage-u se zadaje s korisničkog uređaja. Slijedi prikaz koda u kojem se odabranoj slici dodijeljuje referenca i koja se postavlja na Storage.


```

static Future<TaskSnapshot?> uploadPictureFromFile(File file) async {

    TaskSnapshot x = await
storageReference.child("images/riders/${FirebaseAuth.instance.currentUser!.uid}").put
File(file);

    if(x.state == TaskState.running) {
        print('Running..');
    }
    if(x.state == TaskState.canceled) {
        print('CANCELLED');
    }
    if(x.state == TaskState.paused) {
        print('Paused');
    }

    if(x.state == TaskState.error) {
        print('Error');
    }

    if(x.state == TaskState.success) {
        print('Success');
        String url = await x.ref.getDownloadURL();
        await FirebaseFirestore.instance.runTransaction((transaction) async {
transaction.update(FirebaseFirestore.instance.collection('users').doc(FirebaseAuth.in
stance.currentUser!.uid), {
                'profilePictureUrl' : url,
                'profilePictureTimestamp' : Timestamp.now()
            });
        });
        print('updated url');
    }

    return x;
}

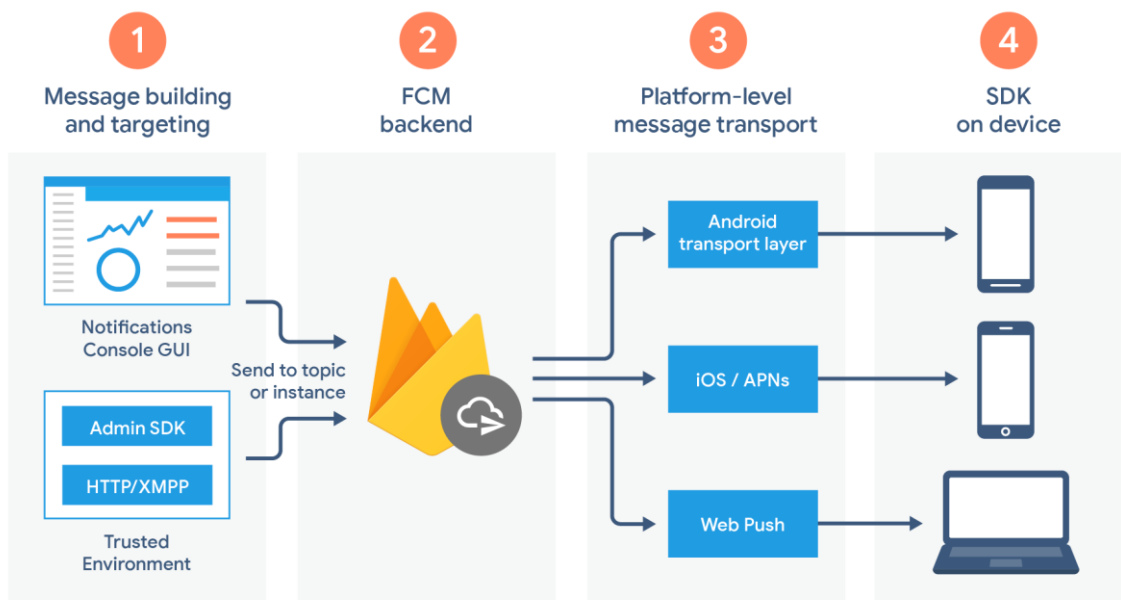
```

Isječak koda 4 – upload slike

3.4. Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) je cross – platform rješenje koje omogućava slanje poruka na klijent uređaje potpuno besplatno. Koristeći FCM moguće je obavijestiti korisničku aplikaciju da imaju novi email ili da su podaci spremni za usklađivanje. Moguće je slati notifikacije korisniku da se poveća interesovanje korisnika. Za korištenje poput instant razmjene poruka, poruka može prenijeti payload do 4 kB klijentskoj aplikaciji [3].

FCM se oslanja na sljedeći set komponenti koji kreiraju, transportuju i primaju poruke:



Slika 8. Set komponenti za dostavljanje poruka³

1. Korištenje alata za kreiranje zahtjeva za poruku. Koristeći **Notification Composer** GUI bazirane opcije za kreiranje zahtjeva za notifikacije. Za potpunu automaciju i podršku za sve vrste poruka, potrebno je kreirati zahtjeve za poruke u nekom server okruženju koje ima podršku za Firebase Admin SDK ili FCM server protokole. Ovo okruženje mogu biti Cloud Functions (u ovom projektu, to je odabrano okruženje) za Firebase, App Engine ili neki drugi server. FCM backend, koji (između ostalih funkcionalnost) prima zahtjeve za poruke, obavlja primanje i slanje poruka korisnicima pretplaćenim na razne teme, generira metapodatke za poruke poput ID poruka.

³ Izvor: <https://firebase.google.com/docs/cloud-messaging/fcm-architecture>

1. Transportni sloj na nivou platforme, koji rutira poruke na ciljane uređaje, obavlja dostavljanje poruka, primjenjuje konfiguracije koje su jedinstvene za platforme. Ovaj transportni sloj uključuje:
 - a. Android transportni sloj (ATL) za Android uređaje s Google Play Servisima
 - b. Apple Push Notification servis (APNs) za iOS uređaje
 - c. Web push protokol za web aplikacije
2. FCM SDK na klijentskim uređajima, gdje se notifikacija prikazuje ili se primjenjuje biznis logika na sadržaj poruke.

3.4.1. Tipovi poruka

FCM pruža dva tipa poruka koje je moguće poslati klijentu:

- **Notification** poruke, često oslovljene kao „display poruke“. Ovaj tip poruka se automatski obrađuje od strane FCM SDK i ne može se promijeniti.
- **Data** poruke, kojim se upravlja klijentskom aplikacijom.

Notification poruke sadrže unaprijed definisan set korisniku vidljivih ključeva. Data poruke, s druge strane, sadrže korisničke definisane ključ – vrijednost parove. Notification poruke mogu sadržavati opcionalni data payload. Maksimalni payload za oba tipa je 4 kB, osim ako se šalje koristeći Notification Composer iz Firebase konzole, koji zahtjeva limit na 1024 karaktera.

Koristeći Data poruke, programer ima potpuno kontrolu nad definisanjem pravila prilikom prijema poruke. Da li to bilo kada se aplikacija koristi ili kada je u pozadini. Uz pomoć ugrađenih Android background servisa (o čemu će biti riječi kasnije), moguće je na jednostavan način poslati neku vrstu apstraktne poruke na klijentske uređaje. FCM je korištene upravo u kombinaciji s tim servisima i Cloud Functions da pošalje poruke vozačima i putnicima novosti o vožnjama (ili drugim stvarima) koje su od bitnosti za njih.

Aplikacija je napravljena na način da putnik pošalje zahtjev za vožnje s informacijama o odredištu i intervalu vremena kada je vožnja potrebna. Na zahtjeve odgovaraju vozači na dva načina, prvo se javljaju da odgovore na zahtjev i putnik se obavještava ko je odgovorio na njegov zahtjev i za koliko je predviđeno vrijeme stizanja. Nakon toga vozač ponovo obavještava putnika da je stigao na lokaciju koja je specificirana u zahtjevu za vožnju i putnik se obavještava da je vozač stigao. Tada putnik ima opciju da kroz notifikaciju brzo odgovori

da vidi vozača ili da prati trenutnu lokaciju. Takođe parametri vožnje se postavljaju, da li putnik želi PIN kodom potvrditi vožnju.

```

export const answerRequest =
  functions.firestore.document("ride_requests/{rideId}").onUpdate(async (snapshot
, context) => {

    const snapshotData = snapshot.after.data();

    const notificationContent = snapshotData.firstName + " " + snapshotData.lastName + " will pick you up in: " + snapshotData.expectedArrival;
    const notificationTitle = "Driver answered your request";
    const token = snapshot.before.data().token;

    try {

      const payload = {
        data: {
          "firstName"      : snapshotData.firstName,
          "lastName"       : snapshotData.lastName,
          "expectedArrival" : snapshotData.expectedArrival,
          "notificationTitle" : notificationTitle,
          "notificationContent" : notificationContent,
          "driverId"        : snapshotData.answeredBy,
          "rideId"          : context.params.rideId,
          "carColor"        : snapshotData.carColor
        },
      }

      const options = {
        android: {
          "priority": "high"
        },
        apns: {
          headers: {
            "apns-priority": "5"
          }
        }
      }

      await admin.messaging().sendToDevice(
        token,
        payload,
        options
      );
      return true;
    }
  }

```

Isječak koda 5 – Funkcija koja šalje poruku na klijent uređaje koristeći FCM

Funkcija na prethodnom isječku koda se poziva kada se ažurira neki dokument u kolekciji **ride_requests**. Odlična mogućnost jeste da je moguće pristupi dokumentu i prije i poslije ažuriranja. Iz ažurirane verzije dokumenta čitamo polje **answeredBy** koje predstavlja id vozača koji je odgovorio na zahtjev. Nakon toga se iz baze čitaju podaci tog vozača i konstruira poruka koja će se poslati. Metoda FCM **sendToDevice** prima 3 parametra, sadržaje poruke, opcije slanja i token na koji se šalje poruka.

U sadržaj poruke su stavljaju podaci koje želimo prikazati krajnjem korisniku, budući da je ovo jako važna poruka, prioritet se postavlja na najveći mogući, a token je upisan prilikom slanja zahtjeva. Ovdje vidimo manu NoSQL baza koje ne posjeduje veze između stranih ključeva kao SQL baze. Potrebno je uraditi još jedno čitanje da bi se dobili podaci korisnika. Međutim postoji i rješenje da se prilikom odgovaranja na zahtjev od strane vozača sve njegove informacije odmah ubace u zahtjev, što bi bilo korektno rješenje budući da ovi dokumenti neće biti od važnosti kasnije.

Bitno je naglasiti da je FCM potpuno besplatan, ali da pri njegovom korištenju treba postupati oprezno jer postoji veliki broj ograničenja [3] i rizik da neke od poruka neće biti dostavljene.

3.5. Cloud Firestore

Cloud Firestore je fleksibilna, automatski skalabilna NoSQL baza podataka za skladištenje i usklađivanje podataka za frontend i backend razvoj aplikacija. Podaci su usklađeni na klijentskim aplikacijama uz pomoć realtime listenera, pruža offline podršku za mobilne i web aplikacije [1]. Firestore nema šemu podataka tako da broj polja i tip u dokumentima može biti različit.

3.5.1. Osobine Cloud Firestore

1. Fleksibilnost

Model Cloud Firestore-a podržava fleksibilne, hijerarhijske strukture podataka. Podaci se skladište unutar dokumenata, organizovanih u kolekcije. Dokumenti mogu sadržavati kompleksne ugniježdene objekte u dodatak i podkolekcija.

2. Izražajni upiti

Moguće je pročitati individualne, odabrane dokumente ili pročitati sve dokumente iz kolekcije koje zadovoljavaju uslove upita. Upiti mogu biti kaskadni i kombinovati filtriranje i sortiranje. Jako bitna napomena je da Firestore automatski indeksira svako polje svake kolekcije tako da su performanse upita proporcionalne količini podataka koje zadovoljavaju upit, a ne količini ukupnih podataka.

3. Realtime ažuriranje

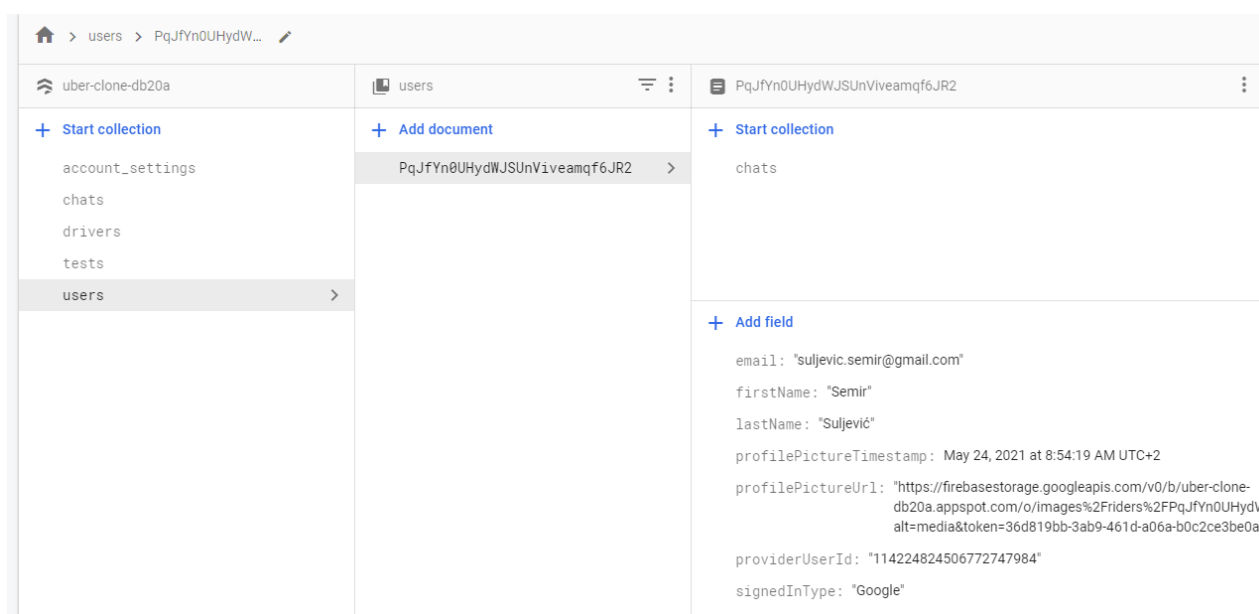
Koristi usklađivanje da ažurira potrebne podatke na klijentskim uređajima. Takođe je dizajnirana da podržava efikasne *one – time fetch* upite.

4. Offline podrška

Firestore prati korištene podatke u aplikaciji, kešira ih tako da su dostupni i kada uređaj nema Internet konekciju. Kada uređaj ponovno dobije Internet konekciju, automatski se vrši sinhronizacija.

5. Dizajnirana za skaliranje

Automatsko repliciranje za multi regionalne aplikacije, batch operacije i podrška realtime transakcijama.



Slika 9 - Prikaz Firestore data modela

Podržani tipovi podataka:

- niz
- logička vrijednost
- bajtovi
- datum i vrijeme (timestamp)
- decimalni broj
- geografska lokacija (geografska visina i širina)
- cijeli broj
- mapa
- null referenca
- referenca na drugi dokumenta
- string

3.5.2. Firestore Rules

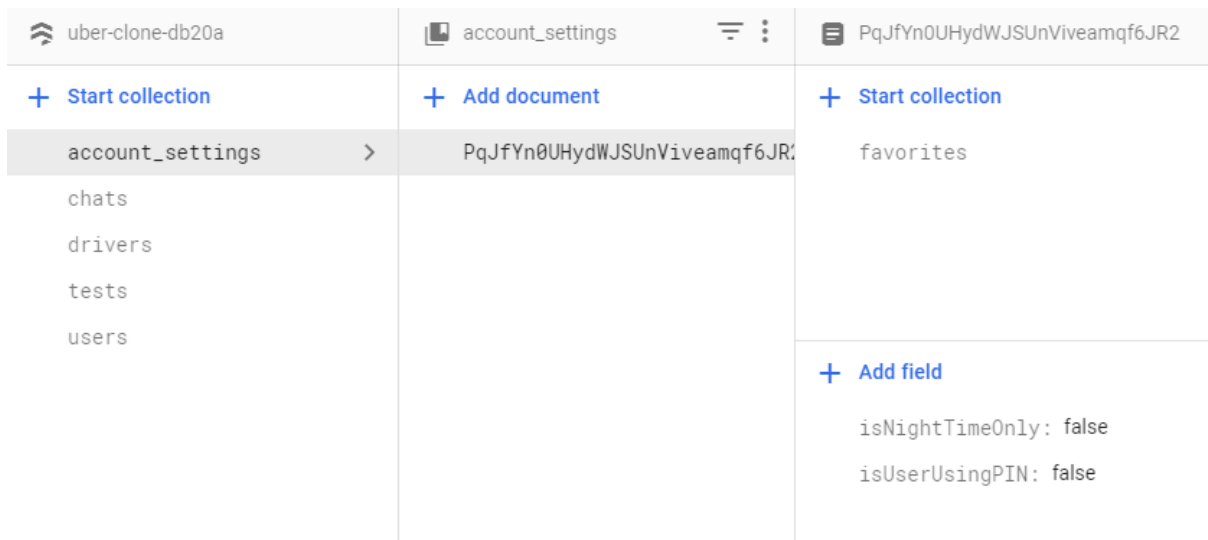
Budući da klijentski uređaji direktno rade s bazom podataka, to je čini ranjivom na razne vrste napada. Svako ko posjeduje izvorni kod, jednostavno bi mogao uticati na bazu podataka bez ikakve kontrole pa čak i obrisati cijelu bazu podataka jednim klikom.

Tu u spas dolazi Firestore Rules, pravila koja definišu pravo pristupa korisnicima. Postoje četiri interakcije s bazom podataka, a to su kreiranje, čitanje, ažuriranje i brisanje podataka (CRUD). U suštini Firestore Rules su ekvivalent SQL triggera.

Firestore Rules mnogo zavise od Firebase Authentication servisa, kao i strukturisanja podataka. Svode se na dozvolu pristupa CRUD radnjama u zavisnosti od jedinstvenog identifikatora koji je dodijelio Firebase Authentication. Firebase ne posjeduje *role – based* model autorizacije za razliku od ASP .NET CORE frameworka, mada slično se može implementirati jednostavno s jednom kolekcijom u kojoj su dokumenti unaprijed definisane uloge.

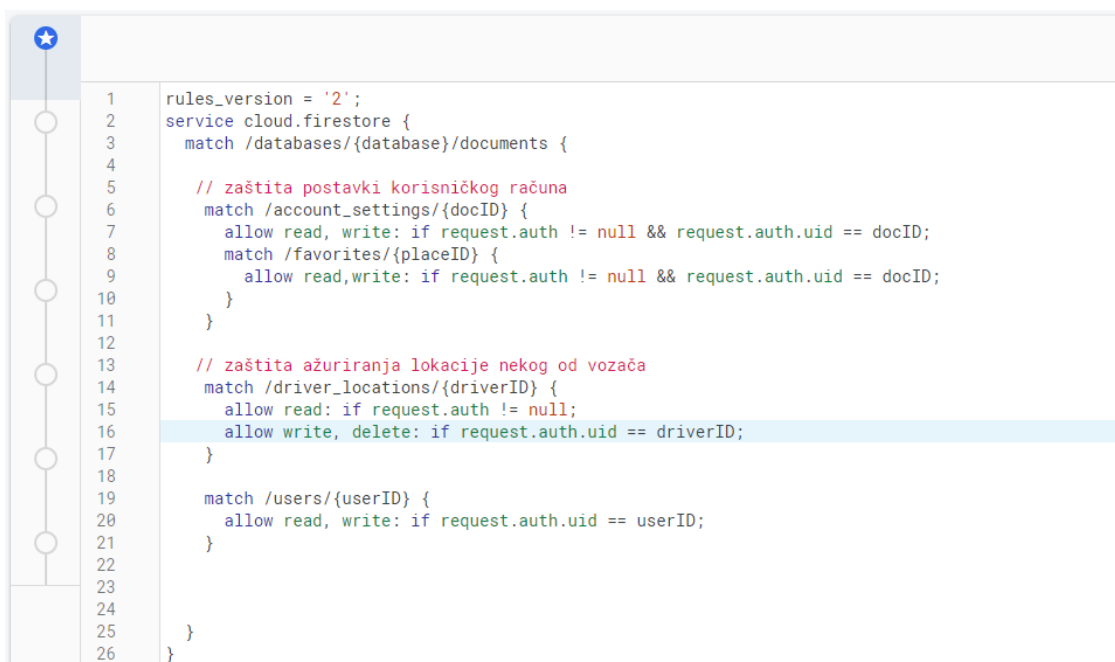
Zašto je bitna struktura podataka i zašto je bitno imati na umu imenovanje kolekcija i dokumenata? Kako se u to sve uklapa Firebase Authentication? Firestore Rules prije svake radnje s bazom provjeri da li zahtjev koji je stigao ima pravo na definisanu radnju. Ukoliko zadovolji, proces se nastavlja inače završava greškom. Većina provjera je besplatna, ali da bi to bilo korisno bitno je imenovati dokumente koristeći jedinstvene identifikatore. Ukoliko to nije slučaj, potrebno je izvršiti radnje čitanje iz baze, što prema modelu naplate Firestore se

naplaćuje. Pokažimo na primjeru kako ograničiti pristup postavkama računama i omiljenim lokacijama korisnika. Postavke računa se nalaze u root kolekciji **account_settings** i svaki dokument predstavlja postavke računa korisnika zajedno s podkolekcijom omiljenih lokacija.



Slika 10 – Prikaz strukture podataka za postavke računa korisnika

Pošto nema servera kao posrednika, svima je dozvoljeno čitati/uređivati ove podatke. Na početku, Firestore je zaštićen tako da svi autorizovani korisnici imaju pravo uređivati bilo koje podatke. Na slici 11 je prikazan proces zaštite pristupa postavkama korisničkog računa kao i čitanja/ažuriranja lokacije nekog od vozača.



Slika 11 – Zaštita postavki korisničkog računa i lokacija vozača

Suština zaštite se svodi na par stvari, a to su:

- struktura podataka
- neka od CRUD radnji
- zahtjev koji je upućen

Pisanje pravila je jako intuitivno i upravlja se koristeći **request** objekat i njegovo polje **auth** koje sadrži **uid** (user id) i **token**. Lahko se zaključuje da imenovanjem dokumenata koristeći jedinstvene identifikatore kreirane od strane Firebase Authentication većina zaštite se svodi na poređenje uid – a iz zahtjeva i imena dokumenta.

Da se nije pratila ova konvencija imenovanja, svaki put bi se radilo čitanje iz baze što bi uticalo i na performanse i cijenu korištenja. Takođe bitno je i spomenuti **resource** objekat koji se odnosi na dokument koji se uređuje ili pokušava upisati u bazu. Ukoliko se pokušava upisati u bazu taj objekat je polje request objekta inače mu se može direktno pristupiti. Kao polja resource objekat sadrži dodatna polja **data** (mapa s poljima), **name** (id) i **__name__** polje koje predstavlja apsolutnu putanju unutar baze.

Pravila pristupa se pišu unutar fajla s **.rules** ekstenzijom i moguće je vršiti provjere na tipove podataka, veličinu stringova i slično.

3.5.3. Cijene korištenja

U zavisnosti od plana plaćanja, postoje određeni limiti na korištenje Firestore baze podataka. U Tabeli 1 su cijene korištenja.

	Besplatna kvota po danu	Cijena nakon besplatne kvote	Cijena
Pročitani dokumenti	50,000	\$0.06	na 100,000 dokumenata
Upisani dokumenti	20,000	\$0.18	na 100,000 dokumenata
Obrisani dokumenti	20,000	\$0.02	na 100,000 dokumenata
Količina podataka	1 GB	\$0.18	na 100,000 dokumenata

Tabela 1 – Cijena korištenja

4. Flutter

Flutter je Google-ov UI toolkit namijenjen za pravljenje *natively* kompajliranih, aplikacija s estetski atraktivnim izgledom za mobilno, web, desktop i embedded pokretanje sve s jednim izvorom koda. Flutter je open source i kao takav, besplatan za korištenje. Njegov način izrade omogućava brzi razvoj aplikacija s odličnim UI/UX-om, odlične performanse, posebno za mobilne aplikacije, trenutno ga klasifikuju kao jednog od primarnih framework-a za razvoj mobilnih aplikacija. Pri predstavljanju 2.2 verzije, Google je objavio da je 1/8 svih aplikacija objavljenih na Google Play-u kreirana u Flutter-u, a to je oko 200 000 aplikacija.

U ovom poglavlju, biti će predstavljene informacije o Flutter-u, historiji, glavnim odlikama, glavne verzije, korištenje u projektu kao i gorući problemi ovog framework-a.

4.1. Historija

Prva verzija Flutter-a objavljena je pod kodnim nazivom **Sky** i korištena je za pravljenje aplikacija za Android operativni sistem. Predstavljen je 2015. godine na Dart developer summit-u s tvrdnjom da obezbjeđuje konzistentnih 120 frejmova u sekundi. 2018. godine na Google Developer Days u Šangaju, Google je objavio Flutter Release Preview 2, što je bila zadnja velika verzija prije stabilne 1.0 verzije. 4. – og decembra 2018. godine, Flutter 1.0 je objavljen na Flutter live događaju kao prva stabilna verzija framework-a. Verzija 1.12 je objavljena 11. – og decembra 2019. godine na Flutter Interactive događaju.

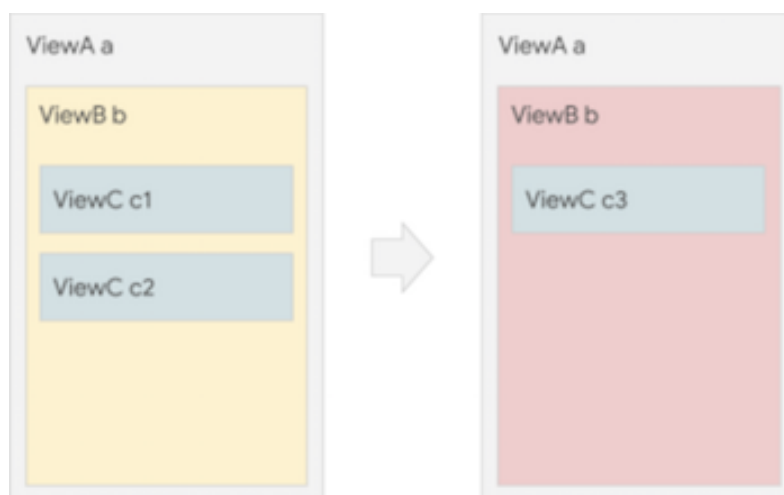
Bitna verzija i promjene u Flutter-u objavljene su 6. – og maja 2020. godine s Dart verzijom 2.8 i Flutter 1.17.0 gdje je dodana podrška za **Metal** API, unaprijeđujući performanse na iOS uređajima (za otprilike 50%). Takođe ova promjena, iako je trebala da unaprijedi performanse donijela je veliki problem koji rješen tek u martu 2021. godine, a to je *jank* pri prvom pokretanju na iOS uređajima. O tome nešto više kasnije.

U martu 2021. Flutter 2.0 je objavljen. Ovo ažuriranje donosi oficijalnu podršku za web bazirane aplikacije s novim CanvasKit render enginom, beta verzije za Windows, MacOS i Linux.

4.2. Deklarativni UI

Za razliku od prethodnika, Flutter ne koristi imperativni stil definisanja izgleda korisničkog interfejsa. Šta to znači? To znači da se izgled ne definiše uz pomoć komponenti koristeći eksterne fajlove i dodjeljivanje tag-ova, id-ova tim komponentama. Za primjer web razvoja, to bi bili HTML i CSS fajlovi, razvoj za android XML fajlovi i slično. Tako recimo za razvoj native android aplikacija logika i kod se piše u java/kotlin fajlovima, dok se izgled definiše pomoću XML fajlova. Nakon toga, programeri dobavljaju reference na te komponente i mutiraju njihovo stanje pomoću raznih metoda i settera. Kod deklarativnog pristupa to nije slučaj, programerima se olakšava posao tako što framework preuzima na sebe mutiranje tih komponenti [8].

Razmotrimo sljedeći primjer mutiranja korisničkog interfejsa:



Slika 12 – Mutiranje korisničkog interfejsa⁴

U imperativnom stilu, obično bi se kreirala referenca na roditelja ViewB komponente i kreirala instanca **b** koristeći selektore ili metode poput **findViewById** i pozivale metode nad tog instancom koje mutiraju njeno stanje. Npr:

```
b.setColor(red)
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)
```

Slika 13 – Mutiranje interfejsa sa slike 16 imperativnim stilom

Takođe možda će biti potrebno ponovno napisati sličnu konfiguraciju ove komponente za slične promjene i uticanja roditelja na stanje te komponente.

⁴ Izvor: <https://flutter.dev/docs/get-started/flutter-for/declarative>

U deklarativnom stilu, konfiguracije interfejsa (poput **widget** u Flutter-u) su nepromjenjive i predstavljaju samo šablon kako će taj dio interfejsa izgledati. Da bi se promjenio određeni dio interfejsa, widget poziva ponovno kreiranja samog sebe (najčešće koristeći metodu **setState()** unutar **StatefulWidget**-a) i kreira novo sadržano drvo. O widgetima mnogo više kasnije.

```
return ViewB(  
  color: red,  
  child: ViewC(...),  
)
```

Slika 14 – Mutiranje interfejsa sa slike 16 deklarativnim stilom

Na ovaj način umjesto da se mutira stara b instanca kada se UI promijeni, Flutter kreira novu instancu widgeta. Framework je zadužen za upravljanje mnogim stvarima (kao održavanje stanja layout-a) koje su sakrivene od programera pomoću **RenderObject** instanci. **RenderObject** instance perzistiraju između frejmova i Flutter widgeti komuniciraju s framework-om da mutiraju **RenderObject** instance između stanja.

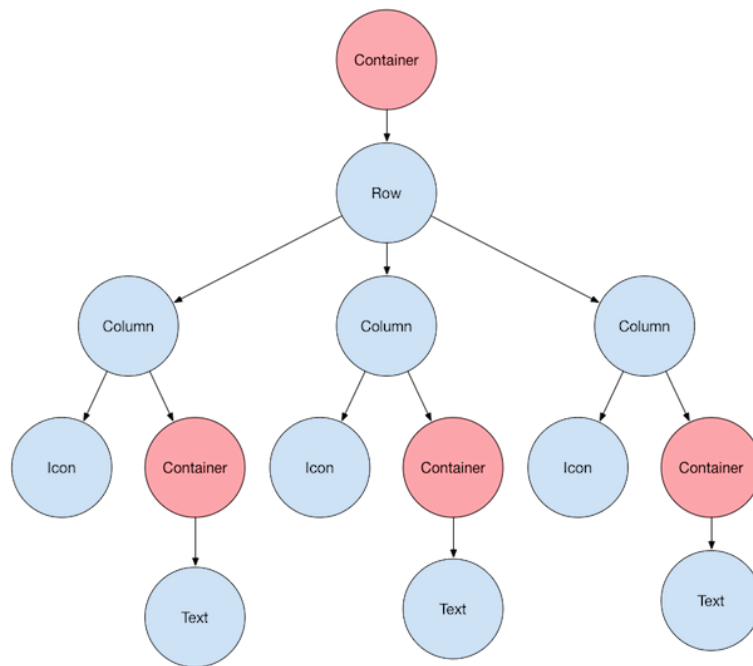
4.3. Flutter layout – i

Kao deklarativni framework, cijeli proces kreiranja layouta se u mnogome razlikuje od imperativnih frameworka. U Flutter-u glavni mehanizam slaganja komponenti (u apstraktnom značenju te riječi) su **widget** komponente. U Flutter-u skoro sve što se koristi je widget pa čak i layout modeli poput **Row**, **Column** predstavljaju widgete [8]. Slike, ikone i tekst na ekranu su sve widgeti. Izgledi se kreiraju kombinovanjem raznih widgeta.



Slika 15 – Primjer kombinovanja widgeta za kompleksan layout

Na ovoj slici korišteno je kombinovanje osnovnih widget-a da bi se kreirao kompleksniji layout, često korišten u dnu ekrana kao navigation bar. Ovako izgleda dijagram prikazanog layouta:



Slika 16 – Dijagram layouta sa slike 15⁵

4.4. Stateless i Stateful widgets

Većina koda koja se tiče izgleda definiše se koristeći **Stateless** i **Stateful** widgete. Ta dva pojma predstavljaju glavnu gradivnu komponentu Flutter-a. Sada ćemo detaljnije objasniti šta to predstavljaju ova dva pojma i prikazati životni ciklus jednog widgeta.

4.4.1. Stateless widget

Widget koji ima nepromjenjivo stanje. Opisuje dio korisničkog interfejsa kreiranjem spoja drugih predefinisanih widget. Proces kreiranje se obavlja rekurzivno sve dok korisnički interfejs nije definisan na niskom nivou hardveru uređaja. Stateless widgete je korisno koristiti za dijelove korisničkog interfejsa koji ne zavise ni od čega, tj. nikada se neće mijenjati.

⁵ Izvor: <https://flutter.dev/docs/development/ui/layout>

```

class Primjer extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      child: Text("Ovo je primjer"),
    );
  }
}

```

Isječak koda 6 – primjer widgeta bez stanja

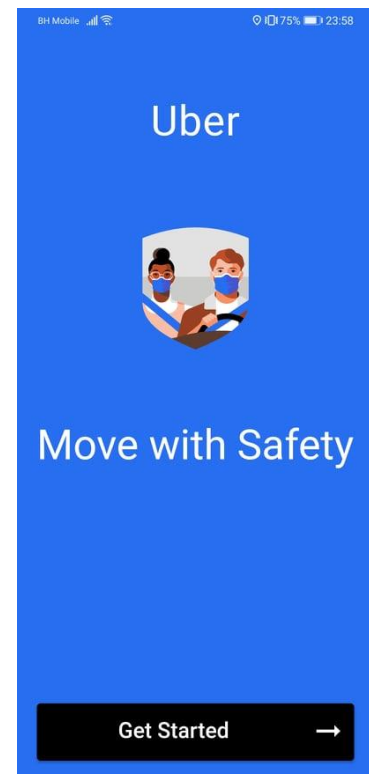
U projektu ovaj tip je korišten manje nego Stateful widget, ali će biti prikazano korištenje na kojem jedna cijela stranica aplikacije je stateless widget.

```

6 class GetStarted extends StatelessWidget {
7
8   static const String route = '/getStarted';
9
10  @override
11  Widget build(BuildContext context) {
12    return Scaffold(
13      body: AnnotatedRegion(
14        value: SystemUiOverlayStyle(
15          statusBarColor: const Color(0xff286ef0),
16          statusBarIconBrightness: Brightness.light
17        ), // SystemUiOverlayStyle
18        child: SafeArea(
19          child: Container(
20            padding: EdgeInsets.fromLTRB(20, 0, 20, 20),
21            color: const Color(0xff286ef0),
22            child: Column(
23              mainAxisAlignment: MainAxisAlignment.center,
24              children: [
25                Expanded(...), // Expanded
26                Expanded(...), // Expanded
27                //Spacer(),
28                Expanded(...), // Expanded
29                ElevatedButton(...) // ElevatedButton
30              ],
31            ), // Column
32          ), // Container
33        ), // SafeArea
34      ), // AnnotatedRegion
35    ); // Scaffold
36  }
37
38 }

```

Slika 17 – Kod početne stranice



Slika 18 – Početna stranica aplikacije

4.4.2. Stateful widget

Widget koji ima promjenljivo stanje. Stanje predstavlja informaciju kojoj se može pristupiti sinhrono kada se widget kreira i koja se može mijenjati u toku životnog ciklusa. Odgovornost korisnika widget-a je da osigurava ažuriranje stanja koristeći **setState**.

Korisna su kada dio interfejsa koji se prikazuje mijenja na bilo koji način i koji pruža interakciju korisniku. Samo se stanje ne čuva direktno u klasi nego u posebnoj klasi koja nasljeđuje widget klasu i ona je zadužena za ažuriranje informacija.

```

class Primjer extends StatefulWidget {
  @override
  _PrimjerState createState() => _PrimjerState();
}

class _PrimjerState extends State<Primjer> {

  int brojac = 0;

  void inkrementiraj() {
    setState(() {
      brojac++;
    });
  }

  void dekrementiraj() {
    setState(() {
      brojac--;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Text("Vrijednost brojača: " + brojac.toString()),
    );
  }
}

```

Isječak koda 7 – primjer widgeta s promjenljivim stanjem

Lahko se može zaključiti u čemu je prednost deklarativnog pristupa u odnosu na imperativni. Cijeli proces djeluje mnogo intuitivnije i sve je vezano uz dart fajlove. Pozivanjem **setState()** mi obavještavamo framework da je došlo do promjene unutar klase koja drži stanje widgeta i prosljedimo nove vrijednosti za varijable koje su promijenile i svu ostalu odgovornost preuzima framework. Koja je funkcija **build()** metode? Sada ćemo se upoznati sa životnim ciklusom widgeta da bi bolje razumijeli ulogu ove metode.

4.4.3. Životni ciklus widgeta

U opisu Stateless i Stateful widgeta mogli smo vidjeti metodu `build()` u kojoj se definišu widgeti koji čine korisnički interfejs. Međutim šta je tačno uloga `build` metode? Sada ćemo predstaviti kako tačno izgleda životni ciklus widgeta.

- **`createState()`**: framework kreira instancu `StatefulWidget`
- **`mounted`** atribut postaje **`true`**: kada `createState()` kreira state klasu `StatefulWidget` – a, **`BuildContext`** se dodijeli tom stanju. `BuildContext` je mjesto u stablu widgeta gdje je trenutni widget kreiran. Kada je **`mounted`** `true`, dozvoljeno je koristiti **`setState`** da se promijeni stanje klase.
- **`initState()`**: prva metoda koja se pozove kada je widget kreiran. Poziva se jednom. Ukoliko se `override` obavezno je da pozove **`super.initState()`**. Moguće je pisati kod unutar ove metode i pristupiti argumentima koji su proslijeđeni.
- **`didChangeDependencies()`**: Poziva se nakon `initState`, može biti pozvana više puta nakon otvaranja tastature, orijentacije uređaja, tako da treba osigurati da kod koji se izvršava bude izvršen samo u prvom pozivu. Ima prednost u odnosu na `initState` jer omogućava pristup `BuildContext`-u koji se koristi u **`Provider`** state management-u.
- **`setState()`**: Poziva se često od strane framework-a i od strane programera. Obavještava framework da je došlo do promjene informacija što ponovo poziva `build` metodu.
- **`didChangeAppLifecycleState(AppLifecycleState)`**: interesantna metoda koja obavještava kada aplikacija nije u korištenju, kada se vrati. Slično kao `onResume` u android native. Ipak ne može se koristiti kao `onResume` zbog prirode framework-a što će biti detaljnije obrađeno kasnije.

Ovaj kratki uvid daje sliku o jednostavnosti korištenja deklarativnog pristupa u odnosu na imperativni. Nema adaptera, eksternih xml, html fajlovan i sve se nalazi u jednom dart fajlu. Šta je bitno naglasiti, da zbog widget prirode ovakav način izrade je identičan i za widgete koji opisuju cijeli ekran i one koji predstavljaju jedno dugme. Međutim ovaj pristup svakako ima i svoje mane. Jedna velika mana ovakvog pristupa jeste nemogućnost, ili bolje rečeno nedostatak, praćenja informacija kada je trenutna ruta u korištenju, a kada ne. To je zbog činjenica da widget može biti dugme, a može biti ruta (zauzima cijeli ekran) kako je već

rečeno. Flutter omogućava implementaciju aplikacije koja je *route aware* ali u poređenju s android native to je svakako kompleksnije rješenje. Svakako da je moguće napraviti, poprično jednostavno, svoj sistem praćenja korištenja ruta, što će biti prikazano kasnije, ali to je jedan od nedostataka koji su primjećeni radom na ovoj aplikaciji.

4.5. State management

Vidjeli smo u prethodnom poglavlju životni ciklus jednog widgeta. Sve djeluje poprilično jednostavno i intuitivno za koristiti, ali šta ukoliko imamo ekran za dopisivanje? Svaka poruka, bilo primljena ili poslana bi trebala biti svojstveni widget. Traka na vrhu ekrana s informacijama i slikom, cijeli proces slanja poruka, konstruisanja poruka trebaju biti neki odvojeni widget, ali međusobno povezani. Koristeći `setState` to je nemoguće uraditi, osim ako bi kao parametar prosljedili neke *callback* funkcije, ali to je previše posla da bi se implementirale neke od najlakših funkcionalnosti.

4.5.1. Podizanje stanja

U Flutteru, ima smisla da stanje koje je bitno za neke widgete držati iznad u hijerarhiji od samog widgeta. Zašto? U framework-u deklarativnog tipa, da bi promijenili interfejs, trebamo ga ponovo kreirati. Nemoguće je imati metodu koje ažuriraju stanje unutar nekih widgeta imperativnim putem. I kada bi se uspjelo ovakvo nešto, radilo bi se direktno protiv framework-a.

4.5.2. Lista pristupa

Flutter ima veliki broj opcija za state management, neki kreirani od strane Google – a, a neki od Flutter zajednice. Aplikacija koristi Provider i GetX u najvećem dijelu i GetIt kao *service locator*.

- Provider
- `setState`
- InheritedWidget & InheritedModel
- Redux
- Fish – Redux
- Bloc / Rx
- GetIt
- MobX
- Flutter Commands
- Binder
- GetX

- Riverpod

4.5.3. Provider

Sada ćemo se dotaći teme čiste arhitekture i šta je potrebno kako bi smo učinili kod održivim. Vidjeli smo u primjeru Stateful widget strukture da je moguće svu logiku i održavanje napraviti unutar klase koja drži stanje. Međutim, vremenom postaje sve teže i teže održavati kod jer se miješa UI kod i kod validacije, biznis logike itd.

Glavni pristup održavanja koda je ostvaren uz pomoć **Provider** paketa, koji je preporučen od strane Flutter tima. Možemo zamisliti Provider kao da pruža objekat neke klase koji ne vidimo direktno unutar widget stabla ali joj možemo pristupiti uz pomoć Provider sintaks. Sada ćemo prikazati jedan Provider koji *upravlja* rutom na kojoj su prikazana prethodna putovanja korisnika.

```
class TripsProvider extends ChangeNotifier{
  bool _shown = false;
  TripType _tripType = TripType.Past;

  bool get shown => _shown;

  void changeShown() {
    _shown = !_shown;
    notifyListeners();
  }

  set shown(bool value) {
    _shown = value;
    notifyListeners();
  }

  TripType get tripType => _tripType;

  set tripType(TripType value) {
    _tripType = value;
    notifyListeners();
  }
}
```

Isječak koda 8 – Provider koji kontroliše kompleksnu logiku rute

Sada je potrebno pri kreiranju rute jednostavno postaviti ovaj Provider kao roditelj widget cijeloj ruti:

```

case UserTrips.route:
  return MaterialPageRoute(
    builder: (_) => ChangeNotifierProvider(
      create: (context) => TripsProvider(),
      child: UserTrips()
    )
  );

```

Isječak koda 9 – Kreiranje rute s Providerom

Sada unutar cijele rute *UserTrips* možemo koristiti objekat klase *TripsProvider* iako nije deklarisan nigdje unutar te rute.

```

return IndexedStack(
  index: Provider.of<TripsProvider>(context, listen: false).tripType.typeToIndex(),
  children: [
    AnimatedContainer(
      duration: const Duration(milliseconds: 100),
      color: Provider.of<TripsProvider>(context).shown ? const Color(0xff2e2e2e) :
const Color(0xff3c4154),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        mainAxisAlignment: MainAxisAlignment.max,
        children: [
          Text('You havent taken a trip yet', style: TextStyle(fontSize: 26)),
          PastTrip(),
          SizedBox(height: 20,),
          PastTrip()
        ],
      ),
    ),
  ],
);

```

Isječak koda 10 – Korištenje Providera unutar UserTrips rute

Ostaje samo objasniti još šta predstavlja klasa koju naslijeđuje *TripsProvider* znači. *ChangeNotifier* omogućava osluškivanje vrijednosti nekih varijabli i automatski poziva ponovno kreiranje widgeta koji zavise od te vrijednosti. Samo je potrebno dodijeliti neku vrijednost unutar provider objekta nekom widgetu i prilikom ažuriranja te vrijednosti pozvati funkciju *notifyListeners* i Flutter odradi ostatak posla za nas.

Bitno je naglasiti da su Provider objekti *scoped*. To znači da neki provider neće biti dostupan svim *child* rutama, nego samo svim widget komponentama unutar rute u kojoj je deklarisan. Ukoliko želimo koristiti u različitim rutama neki Provider potrebno je ga deklarirati na početku unutar *MaterialApp* widgeta.

4.6. Rutiranje

Flutter omogućava veliki broj načina za rutiranje, mnogi kreirani od strane Flutter zajednice. Najbolje i najintuitivnije rješenje je svakako koristeći ugrađene alate. Aplikacija se pokreće unutar **MaterialApp** ili **GetMaterialApp** (koja omogućava korištenjeGetX kontrolera) widgeta koja posjeduje atribut **onGenerateRoute(RouteSettings)**. Nakon što MaterialApp završi proces kreiranja, kreira se instanca **Navigator** klase, koja je posrednik pri rutiranju. Korišten je princip imenovanih ruta koji je direktno povezan s MaterialApp na početku aplikacije da se osigura jednostavan, pregledan i na jednom mjestu definisan sistem rutiranja.

```
class UberRouter {
  static Route<dynamic> generateRoute(RouteSettings settings) {
    switch(settings.name) {

      case AccountSettings.route:
        return MaterialPageRoute(
          builder: (_) => AccountSettings()
        );
      case Chat.route:
        Map<String, dynamic> map = settings.arguments as Map<String, dynamic>;
        return MaterialPageRoute(
          builder: (_) => ChangeNotifierProvider(
            create: (context) => ChatProvider(
              driver: map['driver'],
              userData: map['user']),
            child: Chat(driver: map['driver'] as Driver)
          )
        );
      case Chats.route:
        return MaterialPageRoute(
          builder: (_) => Chats()
        );
      case DriverContact.route:
        return MaterialPageRoute(
          builder: (_) => DriverContact(driver: settings.arguments as Driver,)
        );
      case DriverProfile.route:
        return MaterialPageRoute(
          builder: (_) => ChangeNotifierProvider(
            create: (context) => DriverProfileProvider(id: settings.arguments as
String),
            lazy: false,
            child: DriverProfile()
          )
        );
    }
  }
}
```

Isječak koda 11 – funkcija koja vraća rutu u zavisnosti od imena pozvane imenovane rute

```

child: GetMaterialApp(
  // locale: DevicePreview.locale(context),
  //builder: DevicePreview.appBuilder,
  theme: AppTheme.appTheme(),
  initialRoute: AuthenticationWrapper.route,
  onGenerateRoute: UberRouter.generateRoute,
),
);

```

Isječak koda 12 – dodjela definisane funkcije za rutiranje

Na isječku koda 8 je definisana funkcija koja prima objekat **RouteSettings** kao argument i taj objekat ima dva atributa, **name** koji je tipa String i predstavlja ime rute i atribut **arguments** u kojem se nalaze prosljeđeni argumenti. Da bi se osigurala mogućnost prosljeđivanja različitih tipova i različitog broja argumenata za rute, jasno je da ovaj objekat ne može imati neki konkretan tip te mora biti tipa **dynamic**. Na istom isječku ruti prosljeđujemo 2 argumenta koristeći dart implementaciju mape. Takođe je moguće direktno prosljeđiti argument neki i po potrebi uraditi *downcasting* ili *upcasting*.

Ostalo je još povezati definisanje imena ruta. Pitanje je kako organizovati veliki broj imena ruta i držati ih centralizirane ukoliko bi došlo do mijenjanja njihovih imena ili generalnog održavanja aplikacije. Jednostavan odgovor je držati ime rute kao statični konstanti atribut widgeta koje opisuje izgled rute. Na taj način će sigurno biti ispravno ime na svim pozivima, umjesto string literala i lahko ga je mijenjati jer se taj atribut koristi i u funkciji koja upravlja rutiranjem kao i svim mogućim pozivima te rute. Taj način je predstavljen na slici 28.

```

class Chat extends StatefulWidget {
  static const route = '/chat';
  final Driver driver;

  Chat({required this.driver});
  @override
  _ChatState createState() => _ChatState();
}

```

Isječak koda 13 – definisanje imena rute

Jedna jako interesantna stvar za naglasiti jeste da bi funkcija koja upravlja rutiranjem mogla sadržavati veliki broj **import** naredbi jer joj je potreban pristup svakoj od mogućih ruta. Međutim mogu se *exportovati* svi željeni fajlovi unutar jednog file i koristiti samo jedna import naredba za neograničen broj ruta kao što je prikazano na slici 29.

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:google_sign_in/google_sign_in.dart';
import 'package:provider/provider.dart';
import 'package:uber_clone/components/authentication_wrapper.dart';
import 'package:uber_clone/models/driver.dart';

import 'package:uber_clone/providers/export.dart';
import 'package:uber_clone/screens/export.dart';
```

Isječak koda 14 – import svih file ruta i providera

Definisanje koje fajlove želimo exportovati iz foldera *providers* i *screens*.

```
export 'chat_provider.dart';
export 'chats_provider.dart';
export 'driver_profile_provider.dart';
export 'google_login_provider.dart';
export 'google_login_provider.dart';
export 'home_provider.dart';
export 'location_provider.dart';
export 'profile_pictures_provider.dart';
export 'settings/account_settings.dart';
export 'settings/ride_verification.dart';
export 'trips_provider.dart';
export 'user_data_provider.dart';
```

Isječak koda 15 – fajlovi iz foldera providers

```

export 'account_settings/account_settings.dart';
export 'account_settings/account_settings.dart';
export 'account_settings/ride_verification/ride_verification.dart';
export 'chat/chat.dart';
export 'chats/chats.dart';
export 'driver_contact/driver_contact.dart';
export 'driver_profile/driver_profile.dart';
export 'edit_account/edit_account.dart';
export 'favorites_search/where_to_search.dart';
export 'get_started/choose_account.dart';
export 'get_started/choose_login_type.dart';
export 'get_started/get_started.dart';
export 'google_login/google_login.dart';
export 'help/help.dart';
export 'home/home.dart';
export 'pickup/pickup.dart';
export 'track_driver/track_driver.dart';
export 'user_trips/trips.dart';
export 'wallet/wallet.dart';

```

Isječak koda 16 – fajlovi iz foldera screens

4.7. Integracija FCM i upravljanje s više ulaza u aplikaciju

U projektu se na razne načine koristi neki oblik komunikacije u realnom vremenu koristeći Firestore. Firestore uz pomoć osluškivača omogućava dopisivanje, prikaz dostupnih vozača, trenutnih poziva i tako dalje. Međutim ukoliko želimo na neki način uspostaviti neki oblik komunikacije ukoliko korisnik ne koristi aplikaciju nismo to u mogućnosti koristeći Firestore, bar ne na intuitivne načine. Svako od rješenja koje možda koristi sockete, background servise jednostavno nije u potpunosti zadovoljavajuće. S toga je potrebno koristiti Google-ov **Firestore Cloud Messaging**. Već je spomenuto šta je FCM, a sada ćemo vidjeti i kako se koristi.

Nakon što smo uspostavili komunikaciju za odgovarajući Firebase projekat, potrebno je integrisati **FirebaseMessagingService** native Android kodom. Ovaj servis omogućava konstantu komunikaciju s Android operativnim sistemom i uz pomoć **Google Play Service** održava konstantu konekciju prema Firebase projektu. Koristeći ovaj pristup sada smo u mogućnosti primiti poruke, u apstraktnom smislu, od servera u svim stanjima kojim aplikacija može biti: otvorena, u pozadini, zatvorena. Na ovaj način korisnici mogu zatvoriti aplikaciju, a i dalje biti obaviješteni ukoliko je to potrebno. Ovaj servis posjeduje sljedeće, u ovom slučaju, korisne metode:

- `onDeletedMessages()` – poziva se kada ima više od 100 *non-collapsible* poruka za uređaj ili uređaj dugo nije bio online dugo vremena
- `onNewToken(String)` – poziva se kada aplikacija generiše novi token za korisnika, prilikom instaliranja aplikacije, ponovnog prijavljivanja. Ne može se garantovati da se token neće generisati i u drugim situacijama, zato je važno da se korektno sačuva novi token.
- `onMessageReceived(RemoteMessage)` – poziva se kada uređaj dobije poruku od servera. **RemoteMessage** ima atribut **data**, koji je tipa mape i u njemu se nalazi poruka koju smo poslali iz Cloud Functions okruženja.

Na sljedećem isječku koda je upravo prikazana funkcija `onMessageReceived`, u kojoj se prikazuje notifikacija i pripreme **Intent** objekti. Android Intent je potrebno koristiti upravo da bi se pokrenula aplikacija iz pozadine koristeći dugmad koja se nalaze u notifikaciji.

```

override fun onMessageReceived(message: RemoteMessage) {
    super.onMessageReceived(message)

    val now = Date();
    val notificationId: Int = SimpleDateFormat("ddHHmmss",
        Locale.US).format(now).toInt();

    val noticedDriverAction = Intent(this, DriverNoticedReceiver::class.java)
    val trackDriverAction = Intent(this, TrackDriverReceiver::class.java)

    trackDriverAction.putExtra("driverId",      message.data["driverId"])
    trackDriverAction.putExtra("notificationId", notificationId)
    trackDriverAction.putExtra("rideId",        message.data["rideId"])
    trackDriverAction.putExtra("carColor",      message.data["carColor"])

    noticedDriverAction.putExtra("notificationId", notificationId)

    val driverNoticed: PendingIntent = PendingIntent.getBroadcast(this, 1,
        noticedDriverAction, PendingIntent.FLAG_UPDATE_CURRENT)
    val trackDriver: PendingIntent = PendingIntent.getBroadcast(this, 2,
        trackDriverAction, PendingIntent.FLAG_UPDATE_CURRENT)

    val builder = NotificationCompat.Builder(this, "RideArrival")
        .setSmallIcon(R.drawable.app_icon)
        .setColor(Color.BLACK)
        .setContentTitle(message.data["notificationTitle"])
        .setContentText(message.data["notificationContent"])
        .setPriority(NotificationCompat.PRIORITY_MAX)
        .setAutoCancel(true)
        .setContentIntent(PendingIntent.getActivity(this, 0, Intent(), 0))

    builder.addAction(R.drawable.app_icon, "Track driver location", trackDriver)

    // ride arrival, has additional button
    // to notify driver that the rider has spotted them
    if( !message.data.containsKey("expectedArrival")) {
        builder.addAction(R.drawable.app_icon, "I see the driver", driverNoticed)
    }
    vibrate()

    with(NotificationManagerCompat.from(this)) {
        notify(notificationId, builder.build())
    }
}

```

Isječak koda 17 – prikazivanje notifikacije i priprema Intent objekata

Notifikacija sadrži jedno ili dva *action button*-a koje omogućava direktan pristup ruti koja prati vozača ili obavješćavanje vozača da je primjećen od strane putnika. Budući da se aplikacije prikazuje koristeći native kod, potrebno je pokrenuti koristeći Android Intent objekte. Postavlja se pitanje kako pokrenuti direktno rutu koja prati vozača, a ne tipičnu *Home* rutu? Ovdje stvari postaju interesantne i mogućnosti aplikacije se podižu na novi nivo. Najjednostavnije rješenje je definisati koristiti novi Android Activity koji ima ulogu *android launcher* Activity-a, ali ne možemo je označiti kao glavnu jer bi tada imali dvije.

Imajući u vidu da se aplikacije pokreće iz glavne *main* funkcije, potrebno je naći način kojim ćemo definisati *drugi ulaz* u aplikaciju. To je moguće jednostavno definišući još jednu funkciju koja predstavlja *ulaz* aplikaciju. Prikaz oba ulaza:

```
void main() async{
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}

@pragma('vm:entry-point')
void customEntryPoint() async{
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(
    MaterialApp(
      home: TrackDriver(openedFromNotification: true,),
    )
  );
}
```

Isječak koda 18 – različiti ulazi u aplikaciju

Funkcija *main* će biti pozvana uz pomoć zadane **MainActivity** uvijek, ukoliko mi ne definišemo drugačije. Sada ćemo pogledati Activity koji pokreće *customEntryPoint* funkciju.

```

class CustomLauncher: FlutterActivity() {

    private val CHANNEL = "RideIdFetcher";

    override fun onCreate(savedInstanceState: Bundle?, persistentState:
PersistableBundle?) {
        super.onCreate(savedInstanceState, persistentState)
        FirebaseApp.initializeApp(this);
    }

    private val rideId: String?
        get() = intent.getStringExtra("rideId")

    private val carColor: String?
        get() = intent.getStringExtra("carColor")

    private val driverId: String?
        get() = intent.getStringExtra("driverId")

    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)
        MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
CHANNEL).setMethodCallHandler{
            call, result ->
            if( call.method == "getRideId")
                result.success(rideId)

            if(call.method == "getDriverId")
                result.success(driverId)

            result.success(carColor)
        }
    }
    override fun getDartEntrypointFunctionName(): String {
        return "customEntryPoint";
    }
}

```

Isječak koda 19 – Activity koji pokreće customEntryPoint

Potrebno je izvršiti *override* metode *getDartEntryPointFunctionName* i sada kada se god pokrene ovaj Activity umjesto *main* funkcije pokrenuće se *customEntryPoint* i imamo drugačiji ulaz u aplikaciju. Samo je još pitanje kako pokrenuti ovaj Activity? U tome će nam pomoći Intent objekti koje smo pripremili unutar *onMessageReceived* u servisu. Kod Intent-a koji pokreće Activity:

```

class TrackDriverReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context?, intent: Intent?) {

        if( context == null || intent == null) {
            return;
        }
        val closeDrawer = Intent(Intent.ACTION_CLOSE_SYSTEM_DIALOGS)
        context.sendBroadcast(closeDrawer);

        val ns = Context.NOTIFICATION_SERVICE
        val manager: NotificationManager = context.getSystemService(ns) as
NotificationManager
        manager.cancel(intent.getIntExtra("notificationId", -1))

        val startIntent = Intent(context, CustomLauncher::class.java)

        startIntent.putExtra("driverId", intent.getStringExtra("driverId"))
        startIntent.putExtra("rideId", intent.getStringExtra("rideId"))
        startIntent.putExtra("carColor", intent.getStringExtra("carColor"))

        startIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)

        context.startActivity(startIntent, null)
    }
}

```

Isječak koda 20 – Intent koji pokreće CustomActivity

Activity pokrećemo uz pomoć Intent objekta i imamo mogućnost proslijediti podatke o vozaču koje ćemo poslati i Activity-u. Sada je sve spremno i aplikacije će biti pokrenuta i prikazivati lokaciju vozača. Ostalo je još kako poslati podatke s native strane na Flutter/Dart stranu. O tome upravo u nastavu i pozivanju native platforme.

4.8. Komunikacija s native platformom

I ovom poglavlju ćemo se kratko osvrnuti na pozivanje native koda. Ovo nam je potrebno kada ne postoji dart implementacija neke funkcionalnosti ili za neke stvari koje nije moguće pisati dart kodom poput servisa u pozadini.

Flutter obezbeđuje dva načina pozivanja native funkcija putem **MethodChannel** i **EventChannel**. Prvi nam obezbeđuje da definišemo jednostavno kanal jedinstvenog imena koji može sadržavati više metoda. S druge strane, EventChannel se koristi za komunikaciju putem *event* stream-ova, tj. više kao neki konstanti oslušivač.

Vidjeli smo na isječku koda 15 da *CustomActivity* sadrži tri atributa koji predstavljaju informacije o vozaču. Sada je potrebno pristupiti tim informacijama s Flutter/Dart strane i uspostaviti logiku za praćenje vozača. Nažalost ne postoji način kojim možemo pristupiti Intent-u objektu koji je kreirao Activity pa čak ni direktno Activity-u. Jednostavan način jeste definisati metod kanal koji će davati vrijednosti ovih atributa. Metode koje komuniciraju s definisanim *handlerom* u *CustomActivity*:

```
static const platform = const MethodChannel("RideIdFetcher");

Future<String?> getRideId() async {
  try {
    return await platform.invokeMethod("getRideId");
  } on PlatformException catch(e) {
    print(e.message);
    return null;
  }
}

Future<String?> getDriverId() async {
  try {
    return await platform.invokeMethod("getDriverId");
  }
  on PlatformException catch(e) {
    print(e.message);
    return null;
  }
}

Future<String?> getCarColor() async {
  try {
    return await platform.invokeMethod("getCarColor");
  }
  on PlatformException catch (e) {
    print(e.message);
    return null;
  }
}
```

Isječak koda 21 – metode koje komuniciraju s native stranom

Sada se ove metode pozivaju prilikom kreiranja rute na sljedeći način:

```
@override
void initState() {
  super.initState();

  if(widget.openedFromNotification) {
    getDriverId().then((String? driverId) async{
      if( driverId == null) {
        return;
      }
      String? carColor = await getCarColor();
      if( carColor == null)
        return;
      await load(carColor);
      trackDriver(driverId);
    });
  }

  rootBundle.loadString('assets/map/style.json').then((String value) {
    setState(() {
      mapStyle = value;
    });
  });
}
```

Isječak koda 22 – pozivanje metoda koje komuniciraju s native stranom

Sada smo potpuni povezali Flutter ekvivalentni objekat Android Activity-a s Intentom koji je kreirao Activity i uspješno povezali s pozadinskim servisom Flutter kod. Ovo je bio najuzbudljiviji i zahtjevniji dio projekta, ali donosi i veliku nagradu. Zašto? Ovo je upravo pristup koji koriste sve aplikacije za dopisivanje poput Discord-a i Viber-a. Konstanta komunikacija sa serverima koja ne troši resurse je ostvarena i sve metode unutar Firebase servisa se pozivaju na posebnom threadu tako da je omogućen još jednostavniji način da se izvršavaju spašavanja slika bez da se blokira glavni thread. Ovim je pokazano da Flutter može zamijeniti native Android i iOS razvoj aplikacija za dopisivanje jer pruža identičan pristup primanja poruka, a to je spašavanje putem Firebase servisa u lokalnu bazu podataka i onda prikazivanje na glavnom threadu uz pomoć Flutter koda.

4.9. Performanse i poređenja

Na početku je spomenuto da su velika prepreka za korištenje cross – platform frameworka za pravljenje mobilnih aplikacija bile performanse. Trenutno pored Flutter-a i React Native-a se koriste u malim količinama Xamarin i Ionic. S velikim napretkom na polju razvoja hardvera kao i novijim tehnologijama došlo je do većeg korištenja ovakvih alata. U nastavku će biti izvršeno poređenje Flutter-a i React - Native frameworka s native Android i iOS platformama.

Performanse su mjerenje na sljedećim algoritmima:

- Gauss – Legendre: računanje cifara broja π
- Borwein: računanje $\frac{1}{\pi}$

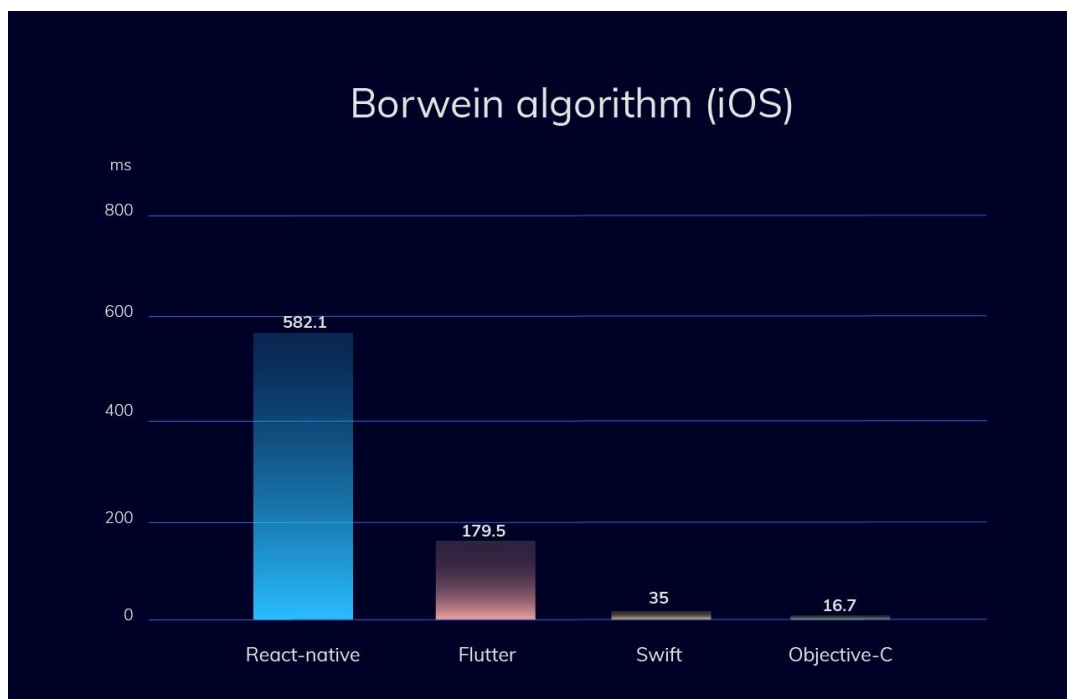
Na sljedećim slikama će biti predstavljen direktan odnos performansi za svaki od algoritama posebno za Android i iOS.

Gauss – Legendre algoritam na iOS (CPU)



Slika 19. Intenzivni test memorije na iOS⁶

Borwein algoritam na iOS

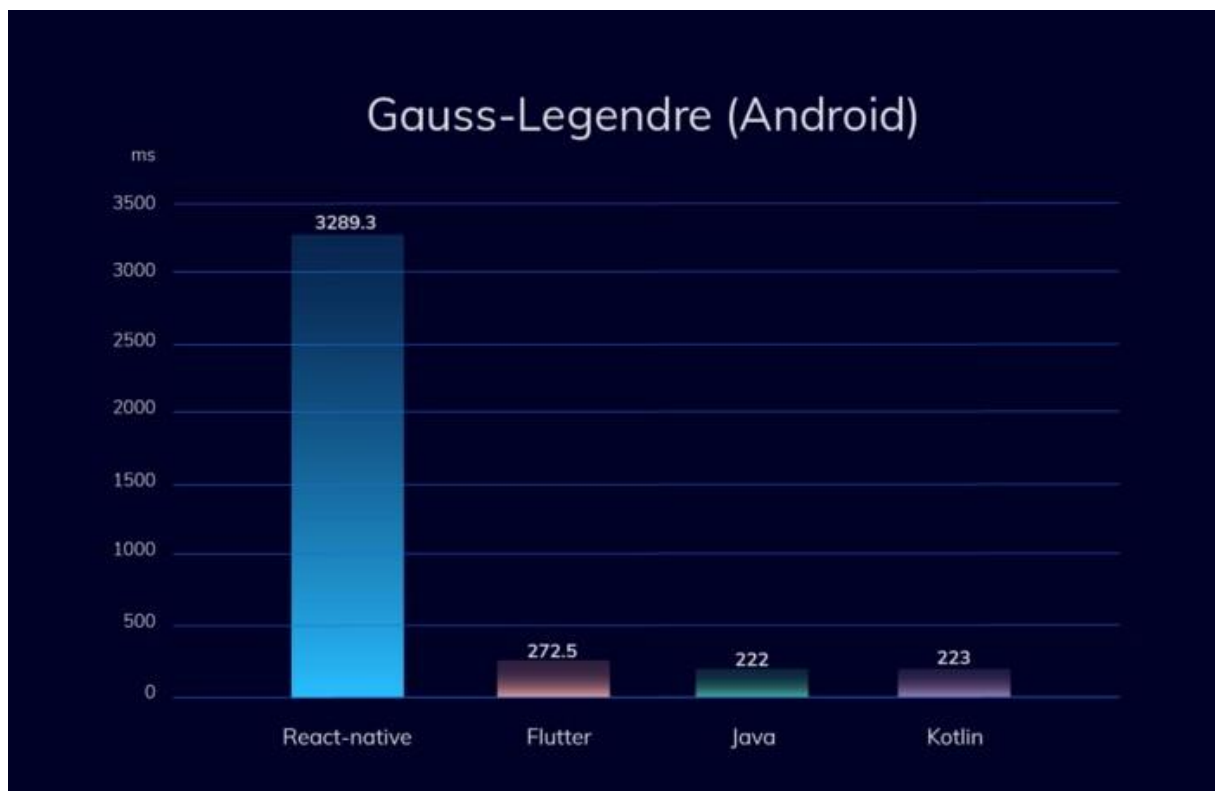


Slika 20 – Intenzivni CPU test na iOS⁶

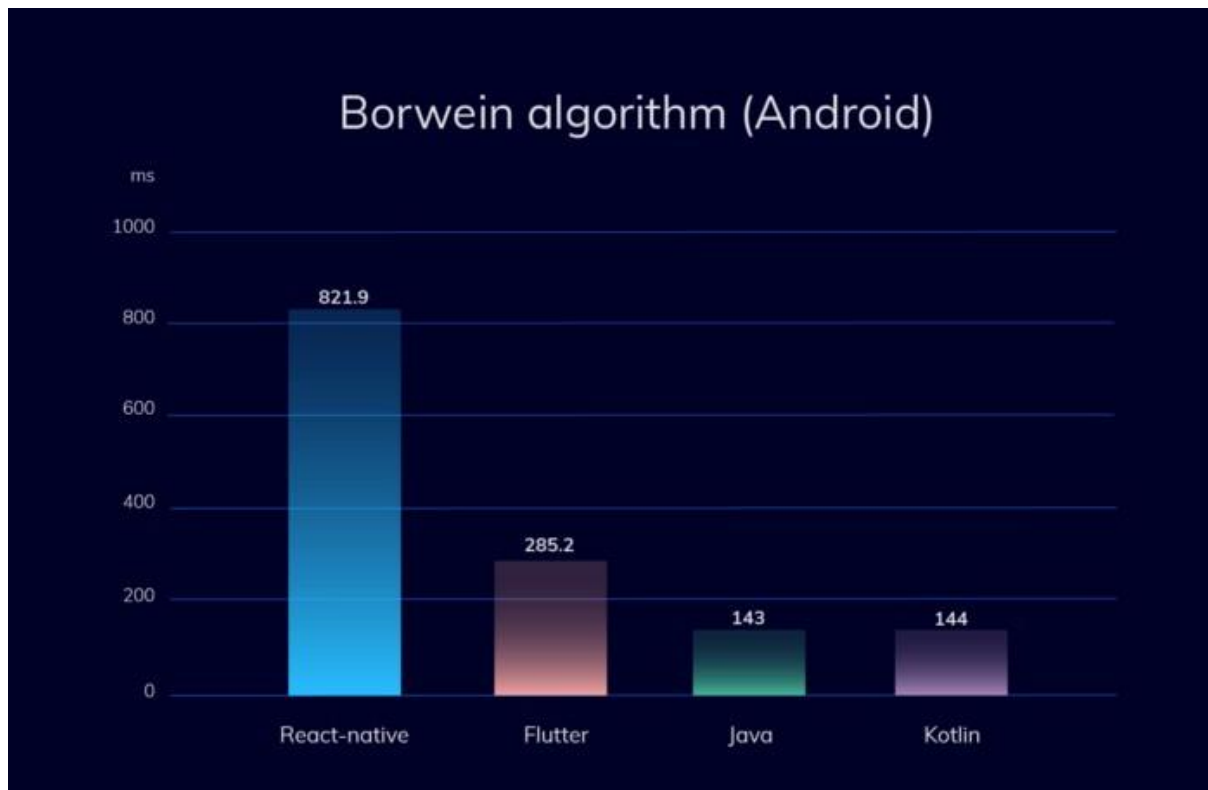
⁶ Izvor: <https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980>

Vidimo na grafovima da je za Gauss – Legendre algoritam React Native čak 16 puta sporiji od Flutter-a. Objective – C je u svim testovima najbolje rješenje gleda performansi za iOS, dok je Flutter brži od Swift-a 15% u prvom testu. Šta je bitno naglasiti je da je Objective – C jezik nižeg nivoa od Dart-a i JavaScript-a i da opet nakon toga nije donio veliku prednost. Takođe Dart posjeduje **dart:ffi** (foreign function interface) [7] koji omogućava korištenje drugih programskih jezika poput C, C++ i Rust koji su najbolje rješenje za ovakve testove zbog svojih performansi.

Gauss – Legendre algoritam na Android uređajima (CPU)



Slika 21 Intenzivni test memorije na Androidu⁶



Slika 22 – Intenzivni CPU test na Android uređajima⁶

I na Android uređajima vidimo sličan odnos snaga između native i cross – platform i da Flutter odnosi pobjedu protiv React Native-a.

Oba algoritma su jako zahtjevna i iziskuju maksimalne performanse. Međutim šta je bitno za dodati da cross – platform framework alati u osnovi nisu ni namijenjeni da budu rješenje za svaku opciju. Koristeći upute kreatora ovih alata kao i očigledne podatke apsolutno je jasno da ukoliko aplikacija iziskuje svaki djelić performansi, gdje su milisekunde bitne, pravo rješenje je native razvoj. Međutim veliki broj aplikacija koji se koriste od korisnika koji, kao u ovom slučaju računaju π s prezinošću od 10 miliona cifara, neće primjetiti razliku od 100 milisekundi u brzini rada njihove aplikacije. Mogu se uzeti za primjer aplikacije za dopisivanje, društvene mreže, e-bankarstvo i slično. Takođe s druge strane postoji veliki broj aplikacija koje su podobne za native razvoj poput intenzivnog procesiranja slika, augmented reality i tako dalje.

Da ne bude sve tako bajno i na strani Flutter-a, postoji *veliki* problem s kojim se Flutter suočava. Problem je prisutan samo na iOS uređajima i vezan je za izvršavanje animacija što će biti predstavljeno u nastavku.

4.10. Pripremanje animacija

Ukoliko aplikacija ima animacije koje ne izgledaju glatko prilikom prvog izvršavanja a kasnije postanu glatke, vjerovatno je došlo do *shader compilation jank*. **Shader** je kod koji se izvršava na GPU-u uređaja. Kada se shader prvi put pokrene, mora se kompajlirati na uređaju. To kompajliranje bi moglo trajati reda nekoliko stotina milisekundi, a jedan frame bi trebao biti iscrtan unutar 16 milisekundi da bi se održalo 60 frame – ova u sekundi. Ta kompilacija bi mogla izazvati preskakanje desetine frame-ova i broj frame-ova u sekundi bi se smanjio sa 60 na 6 [8].

Definicija *prvog izvršavanja* je različita na Android i iOS uređajima. Na Android uređajima prvo izvršavanje je pokretanje nakon instalacije. Sva sljedeća pokretanja ne bi trebala imati problema. Na iOS uređajima prvo izvršavanje je nakon svakog pokretanja aplikacije koja se prethodno nije nalazila u memoriji.

Flutter obezbjeđuje alate za komandnu liniju koji skupljaju sve shader-e koji će možda trebati krajnjim korisnicima u SkSL (Skia Shader Language) formatu. SkSL shader-i se onda mogu zapakovati u aplikaciju, zagrijati (pre – kompajlirati) kada korisnik tek otvori aplikaciju i eliminisati compilation jank u sljedećim animacijama [8].

Da se obezbijedi ovaj proces potrebno je pokrenuti aplikaciju u **profile** režimu i jednostavno koristiti aplikaciju, pokrećući što je više animacija moguće koje bi korisnik koristio. Nakon toga se shader-i zapišu u neki json file i kreira se release apk i ipa sa file-om u kojem se nalaze prekompajlirani shader-i.

Ovo je problem koji se samo dešava na iOS uređajima jer Android kešira animacije nakon prvog korištenja. Kroz rad u proteklih 8 mjeseci primjećeno je da na Android uređajima čak nije ni potreban ovaj postupak jer i pri prvom pokretanju animacije se izvršavaju bez problema. Jedino korištenje koje uzrokuje probleme je otvaranje prozora za biranje gmail naloga za login u aplikaciju. Takođe je taj isti problem, u manjoj mjeri, primjećen pri korištenju native aplikacija.

5. Zaključak

Uz nove tehnologije poput Flutter, Firebase koje olakšavaju posao developera, omogućavaju veću mogućnost ponovnog korištenja koda, moguće je kreirati zaista impresivne softvere u kratkom vremenskom periodu. Apsolutno je jasno da ukoliko želimo ovo primijeniti u stvarnom životu gdje se dnevno obave milioni vožnji Firebase nije prvi izbor, najviše zbog nedostatka sigurnosti i ograničenja na listenere u realnom vremenu.

U ovom projektu koji se radio 4 mjeseca, upoznat je Flutter framework i njegove mogućnosti i ograničenja. Bitno je naglasiti kako je *developer experience* u Flutter-u apsolutno nenadmašiv u odnosu na bilo koji drugi i da je užitek raditi na interfejsu. Takođe, bitno je istaknuti da je moguće uspostaviti isti nivo kontrole nad background servisima kao i u native aplikacijama, što je bio najuzbudljiviji dio rada na projektu.

Naravno sve ovo ne bi bilo moguće bez Google SDK-ove za korištenje mape, Firebase-a i svih ostalih API-ova koji su korišteni u aplikaciji. Google je zaista učinio posao developera mnogo lakšim jer je korištenje ovih SDK-ova isuviše lagano. Moguće je potpuno zanemariti kako su neke stvari implementirane u početku projekta dok želimo što prije pokrenuti naš projekat pa se onda fokusirati na implementacijske detalje kada vrijeme nije kritičan faktor.

U ovom radu se dotaklo samo određenih implementacijskih detalja za koje smo mislili da su najinteresantniji. Cijeli ovaj projekat je popriličito obiman, za jednog početnika, i detaljno objašnjavanje svih funkcionalnosti bi bilo predugo. Ovo putovanje, ako se može tako nazvati, je iskorišteno u svrhe prije svega učenja i najbitnije da je imalo za rezultat upoznavanje cijelog ciklusa razvoja softvera, važnost organizacije, modeliranja i ostalih osobina koje se vežu za pravilan razvoj softvera.

6. Literatura

1. Firebase Firestore dokumentacija

Dostupno na: <https://firebase.google.com/docs/firestore>

2. Firebase Authentication dokumentacija

Dostupno na: <https://firebase.google.com/docs/auth>

3. Firebase Cloud Messaging dokumentacija

Dostupno na: <https://firebase.google.com/docs/cloud-messaging>

4. Firebase Storage dokumentacija

Dostupno na: <https://firebase.google.com/docs/storage>

5. Firebase Cloud Functions dokumentacija

Dostupno na: <https://firebase.google.com/docs/functions>

6. Firebase Messaging Service Kotlin dokumentacija

Dostupno na:

<https://firebase.google.com/docs/reference/android/com/google/firebase/messaging/FirebaseMessagingService>

7. Dart dokumentacija

Dostupno na: <https://dart.dev/>

8. Flutter dokumentacija

Dostupno na: <https://flutter.dev/docs>

9. Dokumentacija paketa

Dostupno na: <https://pub.dev/>

10. Dart historija

Dostupno na: [https://en.wikipedia.org/wiki/Dart_\(programming_language\)](https://en.wikipedia.org/wiki/Dart_(programming_language))

11. Uber historija i statistika

Dostupno na: <https://en.wikipedia.org/wiki/Uber>