UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Lazar Čeliković

# UPRAVLJANJE RAZVOJEM MOBILNIH APLIKACIJA SA FOKUSOM NA PERFORMANSE I KVALITET

master rad

Beograd, 2024.

**Mentor**:

dr Vladimir Filipović, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

**Članovi komisije**:

dr Aleksandar Kartelj, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Staša Vujičić Stanković, docent
Univerzitet u Beogradu, Matematički fakultet

**Datum odbrane**: _____

**Naslov master rada**: Upravljanje razvojem mobilnih aplikacija sa fokusom na performanse i kvalitet

**Rezime**: Mobilne aplikacije i aplikacije generalno često se razvijaju sa primarnim ciljem da se pokrije što je moguće više funkcionalnosti, dok se koncepti kao što su performanse zanemaruju dok problem ne postane evidentan. U ovom radu se razmatra pristup koji koncepte performansi, metrika i održivosti razvoja stavlja u središte i implementira ih na samom početku razvojnog ciklusa. Rad prikazuje kako postavljanje sistema za merenje performansi, poslovnih metrika i analizu grešaka olakšava dalji razvoj u kasnijim stadijumima projekta. Rešenje koje ćemo analizirati razvijeno je uz pomoć razvojnog okruženja ReactNative uz podršku RTKQ biblioteke za upravljanje stanjem podataka. Rad prikazuje detalje implementacije ovih sistema kao i benefite koji se mogu dobiti korišćenjem istih. Pomenuti koncepti biće prikazani i analizirani na primeru mobilne aplikacije za vremensku prognozu.

**Ključne reči**: mobilna aplikacija, performanse, dogadjaji, Sentry, React Native

# Sadržaj

**1 Uvod** **1**

**2 Pregled tehnologija za razvoj mobilnih aplikacija** **3**
2.1 Tipovi mobilnih aplikacija . . . . . 3

**3 Osnove radnog okvira React Native** **9**
3.1 Način rada radnog okvira React Native . . . . . 9
3.2 Most u React Native radnom okviru . . . . . 11

**4 Aplikacija za vremensku prognozu** **14**
4.1 Korisnički interfejs aplikacije . . . . . 14

**5 Sistem za merenje performansi aplikacije** **20**
5.1 Alati za merenje performansi . . . . . 20
5.2 Sentry . . . . . 22
5.3 Optimizacija listi za prikazivanje . . . . . 28

**6 Sistem za praćenje događaja u mobilnoj aplikaciji** **33**
6.1 Značaj poslovnih metrika . . . . . 33
6.2 Arhitektura sistema za prećenje događaja . . . . . 34

**7 Sistem za praćenje grešaka unutar aplikacije** **40**
7.1 Funkcionalnosti platforme Sentry . . . . . 41
7.2 Poboljšana arhitektura za praćenje grešaka . . . . . 45

**8 Zaključak** **48**

**Bibliografija** **50**

# Glava 1

# Uvod

U današnjem digitalnom dobu, mobilne aplikacije predstavljaju ključan element poslovanja i komunikacije sa korisnicima. Upravljanje razvojem mobilnih aplikacija postaje sve kompleksnije, zahtevajući duboko razumevanje tehnologija, procesa i alata koji utiču na performanse i kvalitet aplikacija. Ovaj master rad istražuje različite aspekte ovog dinamičnog polja, sa posebnim fokusom na primenu radnog okvira React Native i sisteme za praćenje performansi, događaja i grešaka unutar mobilnih aplikacija.

Prvo poglavlje, Pregled tehnologija za razvoj mobilnih aplikacija, pruža osnovni pregled glavnih tehnologija, alata i platformi koje su ključne za efikasan razvoj mobilnih aplikacija. Analiziraće se prednosti i izazovi različitih pristupa razvoja mobilnih aplikacija, sa ciljem identifikacije najboljih praksi.

Drugo poglavlje, Osnove radnog okvira React Native, detaljno istražuje arhitekturu, karakteristike i primene React Native okvira u razvoju mobilnih aplikacija. Biće objašnjeno kako ovaj radni okvir funkcioniše „ispod žita" i zbog čega je to od presudnog značaja za dobro razumevanja performansi naše aplikacije.

Treće poglavlje, Aplikacija za vremensku prognozu, služi kao konkretni primer primene React Native okvira u praksi. Fokus će biti na implementaciji aplikacije za vremensku prognozu, uključujući integraciju sa spoljnim API-jima i strategije za poboljšanje korisničkog iskustva.

Četvrto poglavlje, Sistem za merenje performansi aplikacije, istražuje važnost merenja, analize i optimizacije performansi mobilnih aplikacija. Biće razmatrani ključni metodi, alati i tehnike za merenje performansi, sa fokusom na identifikaciju i rešavanje potencijalnih problema koji mogu uticati na performanse aplikacija. Biće dat pregled biblioteka FlatList i FlashList i detaljna analiza performansi koje nam

pružaju. Takođe, u ovom poglavlju ćemo se upoznati sa alatom Sentry kog ćemo koristiti dalje kroz rad.

Peto poglavlje, Sistem za praćenje događaja u mobilnoj aplikaciji, analizira značaj praćenja korisničkih interakcija i događaja unutar mobilnih aplikacija. Biće data konkretna implementacija jednog ovakvog sistema. Nakog toga će biti analizirana primena dobijenih podataka za unapređenje korisničkog iskustva.

Šesto poglavlje, Sistem za praćenje grešaka unutar aplikacije, istražuje važnost efikasnog otkrivanja, praćenja i rešavanja grešaka u mobilnim aplikacijama. U ovom poglavlju ćemo se vratiti alatu Sentry i dati pregled onoga što nam on pruža kako bismo praćenje grešaka olakšali maksimalno.

Ovaj rad ima za cilj da pruži dublje razumevanje ključnih aspekata upravljanja razvojem mobilnih aplikacija, sa naglaskom na performanse i kvalitet. Kroz analizu tehnologija, primere implementacije i prakse za optimizaciju, rad će ponuditi vredan uvid u efikasno vođenje procesa razvoja mobilnih aplikacija u savremenom poslovnom okruženju.

# Glava 2

# Pregled tehnologija za razvoj mobilnih aplikacija

Mobilne aplikacije su doživele neverovatnu evoluciju od svog početka. Svedoci smo kako su one iz jednostavnih alata za komunikaciju ili zabavu prerasle u složene sisteme koji nam pomažu u svakodnevnom životu. Danas, aplikacije nisu samo način da ostanemo povezani s drugima, već i sredstvo za upravljanje finansijama, učenje novih veština, pa čak i za praćenje zdravlja i fitnesa. Razvoj tehnologija poput veštačke inteligencije i mašinskog učenja dodatno je unapredio funkcionalnost i intuitivnost aplikacija, pružajući korisnicima personalizovano iskustvo koje je prethodnih godina bilo nezamislivo. Napredak ne samo da je promenio način na koji korisnik ima interakciju sa uređajima, već je i potpuno preoblikovao digitalni pejzaž, otvarajući nove mogućnosti za razvoj i inovacije u budućnosti. Ovaj napredak mobilnih aplikacija pratio je i razvoj tehnologija koje se koriste u izradi istih. Tokom vremena došlo je do razvoja većeg broja tipova mobilnih aplikacija kao i do velikog broja radnih okvira koji omogućavaju lakši razvoj i održavanje aplikacija.

## 2.1 Tipovi mobilnih aplikacija

Mobilne aplikacije mogu se kategorisati u nekoliko osnovnih tipova, svaki sa svojim specifičnim funkcijama i ciljevima. Jedna od podela je na nativne aplikacije (eng. native mobile applications), veb aplikacije (eng. web mobile applications), hibridne mobilne aplikacije i progrsivne veb alikacije (eng. progressive web applications - (PWA)). Svaki od ovih tipova ima svoje prednosti i mane, te izbor tipa aplikacije zavisi od specifičnih potreba i ciljeva projekta. U nastavku rada biće analizirana

svaka od pomenutih kategorija.

### Nativne mobilne aplikacije

Nativne mobilne aplikacije [12] (eng. native mobile applications) su programi razvijeni za specifičan operativni sistem, kao što je iOS [5] ili Android [4], koristeći programske jezike koji su specifični za svaku od platformi. Ti jezici su Swift i Objective C za iOS ili Kotlin i Javu za Android. Ove aplikacije se instaliraju direktno na mobilni uređaj preko prodavnice aplikacija na uređaju i optimizovane su da pružaju maksimalnu efikasnost i iskoristivost hardverskih karakteristika uređaja. Prednsoti ovog tipa mobilnih aplikacija su:

- Budući da je ovaj tip aplikacija optimizovan za specifičnu platformu, po pravilu se mogu izvući neuporedivo bolje performanse u poređenju sa ostalim tipovima. Ovo se ostvaruje jer nema međuslojeva u komunikaciji već se ista vrši direktno između operativnog sistema i mobilne aplikacije kao što je prikazano na slici 2.1
- Fluidne animacije i intuitivan interfejs koji potiče od samog operativnog sistema rezultuju poboljšanim korisničkim iskustvom.
- Mogućnost pristupa punom setu hardverskih i softverskih funkcija samog uređaja kao što su kamera, GPS, senzori.
- Nativne mobilne aplikacije se objavljuju na prodavnicama aplikacija kao sto su Play prodavnica (eng. Play Store) i App prodavnica (eng. App store). Tokom ovog procesa se vrši detaljno ispitivanje aplikacija i na ovaj način se povećava sigurnost.

Mane ovog tipa mobilnih aplikacija su:

- Razvijanje posebne aplikacije za svaki operativni sistem povećava troškove kao i vreme koje je potrebno da se aplikacija pusti u korišćenje.
- Različiti operativni sistemi često iziskuju različite programske jezike za razvoj, što može otežati pronalaženje razvojnih timova.
- Svaka promena ili ažuriranje aplikacija zahteva odvojeno slanje na odobrenje prodavnici aplikacija za svaki operativni sistem. Ovaj proces je često vremenski zahtevan.

Slika 2.1: Komunikacija nativne aplikacije i operativnog sistema

## Veb aplikacije

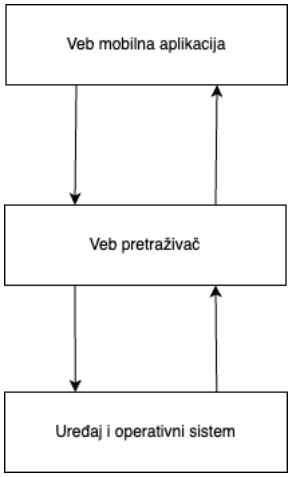
Veb aplikacijama (eng. web applications) pristupa se preko internet pregledača i ne zahtevaju preuzimanje i instalaciju na uređaj kao tradicionalne aplikacije. One su dizajnirane da budu kompatibilne sa različitim platformama i pružaju jedinstveno iskustvo korisnicima na različitim uređajima. Veb aplikacije predstavljaju korisnu opciju za aplikacije koje zahtevaju brzu dostupnost i lako održavanje, ali kada je u pitanju duboka integracija sa uređajem i složene interakcije, ovaj tip aplikacije često zaostaje za mogućnostima koje pružaju nativne aplikacije. Prednosti veb mobilnih aplikacija:

- Ovaj tip mobilnih aplikacija će raditi na bilo kom uređaju koji ima veb pregledač, što uklanja potrebu za posebnim verzijama za različite operativne sisteme.
- Ovaj tip mobilnih aplikacija razvija se samo jednom za sve platforme, čime se troškovi razvoja i ažuriranja značajno smanjuju u odnosu na nativne aplikacije.
- Korisnici veb aplikacija imaće pristup najnovijim izmenama onog momenta kada te izmene budu puštene na produkciju, što znači da se uklanja potreba za preuzimanjem ažururanja, što znatno olakšava distribuciju i održavanje.

Mane ovog tipa mobilnih aplikacija su:

- Veb mobilne aplikacije zavise od brzine i kvaliteta internet konekcije, a takođe ne mogu u potpunosti iskoristiti sve hardverske mogućnosti uređaja kao što to mogu nativne mobilne aplikacije.
- Korisnički interfejsi ovog tipa mobilnih aplikacija često su manje fluidni i intuitivni u odnosu na nativne mobilne aplikacije, što može uticati na ukupno korisničko iskustvo.

- Funkcionalnosti i performanse mogu varirati u zavisnosti od pretraživača koji korisnik upotrebljava, što može rezultovati u nekonzistentnosti u korisničkom iskustvu. Performanse delimično variraju i zbog činjenice da postoji međusloj komunikacije između same aplikacije i operativnog sistema što je i prikazano na slici 2.2.




Slika 2.2: Komunikacija veb aplikacije i operativnog sistema

## Progresivne veb mobilne aplikacije

Progresivne veb aplikacije [6] (eng. Progressive Web Applications - PWA) su vrsta aplikacija koje kombinuju najbolje osobine tradicionalnih veb aplikacija i nativnih mobilnih aplikacija. One se izvršavaju u veb pregledaču, ali pružaju korisničko iskustvo koje je blisko iskustvu koje pružaju nativne mobilne aplikacije. PWA su dizajnirane da budu brze, pouzdane, čak i u uslovima loše internet konekcije. Osnovne prednosti ovog tipa aplikacija su:

- Zahvaljujući servisnim radnicima [24] (eng. service workers), progresivne veb aplikacije mogu raditi bez ili na slabim mrežnim konekcijama, pružajući osnovnu funkcionalnost kada nema internet konekcije.

- Korisnici mogu „instalirati" progresivne veb aplikacije na svoje uređaje, omogućavajući im da pristupe aplikaciji sa početnog ekrana, slično kao kod nativnih aplikacija.

- Progresivne veb aplikacije imaju podršku za notifikacije (eng. push notification), što omogućava bolje zadržavanje korisnika (eng. retention).

- Progresivne veb aplikacije zahtevaju HTTPS[1] za pokretanje, što omogućava sigurniju razmenu podataka između korisnika i aplikacije.

Mane ovog tipa mobilnih aplikacija su:

- Iako progresivne veb aplikacije imaju pristup nekim hardverskim funkcijama, one ne mogu potpuno iskoristiti sve kapacitete uređaja kao što to mogu nativne aplikacije.
- Ovaj tip aplikacija nije jednako podržan na svim platformama, s iOS uređajima koji imaju određena ograničenja u pogledu funkcionalnosti u odnosu na Android uređaje.
- Razvoj ovog tipa mobilnih aplikacija može biti složeniji od tradicionalnih veb aplikacija zbog potrebe za implementacijom servisnih radnika i upravljanjem keširanim podacima za offline rad.

U poređenju sa tradicionalnim web aplikacijama, PWA pružaju bolje korisničko iskustvo, veću pouzdanost i više funkcionalnosti koje su bliske nativnom iskustvu, čineći ih izuzetno privlačnim izborom za razvoj aplikacija koje treba da budu efikasne i dostupne širom različitih platformi i uslova povezivanja.

## Višeplatformske mobilne aplikacije

Višeplatformske mobilne aplikacije [11] (eng. crossplatform mobile application) predstavljaju spoj nativnih i veb tehnologija, omogućavajući razvoj aplikacija koje se mogu instalirati na uređaj, ali se izvršavaju u veb kontejneru. Primer ove komunikacije dat je na slici 2.3. Ove aplikacije koriste kombinaciju HTML, CSS, i JavaScript za izradu korisničkog interfejsa, dok istovremeno koriste mostove (eng. bridge) poput Cordova ili Ionic [19] za pristup nativnim funkcijama uređaja. Ovaj tip aplikacije biće implementiran u okviru ovog rada pa ćemo imati prilike detaljnije da se upoznamo sa karaktiristikama istih. Osnovne prednosti ovog tipa mobilnih aplikacija su:

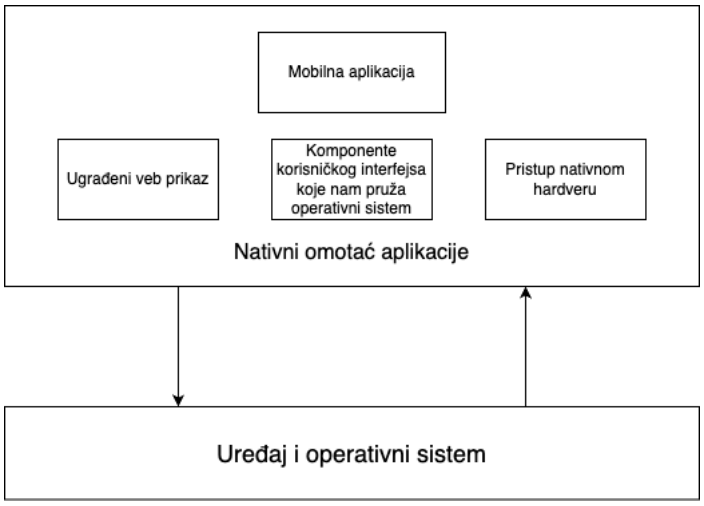
- Razvoj se vrši koristeći veb tehnologije koje su mnogim programerima već poznate, što smanjuje vreme i troškove razvoja.

[1] eng. Hypertext transfer protocol secure

- Jedan kod može se koristiti za više platformi (iOS, Android, Windows Phone), što dodatno smanjuje troškove i olakšava održavanje.
- Kroz različite biblioteke, višeplatformske mobilne aplikacije mogu koristiti hardver uređaja (kamere, GPS, senzore i druge funkcije).

Mane ovog tipa mobilnih aplikacija su:

- Zbog dodatnog sloja apstrakcije i oslanjanja na veb tehnologije, performanse višeplatformskih aplikacija mogu biti lošije u odnosu na nativne aplikacije, posebno za zahtevne zadatke. Ovaj uticaj na performanse je idealan razlog zbog koga bi trebalo razmotriti implementiranje sistema za praćenje istih na početku razvoja aplikacije.
- Iako se ne trude da imitiraju izgled i osećaj nativnih aplikacija, višeplatformske mobilne aplikacije često ne mogu potpuno iskopirati fludinost i odziv nativnog korisničkog interfejsa.
- Višeplatformske mobilne aplikacije zavise od platformi kao što su Cordova ili Ionic, što može dovesti do problema ako se ti alati ne ažuriraju redovno ili ne podržavaju najnovije verzije operativnih sistema.




Slika 2.3: Komunikacija hibridne mobilne aplikacije i operativnog sistema

# Glava 3

# Osnove radnog okvira React Native

React Native [3, 22] (eng. React Native) predstavlja JavaScript radni okvir (eng. framework) za kreiranje mobilnih aplikacija za iOS i Android operativne sisteme. Zasnovan je na React radnom okviru, sa izmenom da cilja mobilne uređaje. Dakle, ovaj radni okvir omogućava nam da kreiramo mobilne aplikacije koje imaju nativni izgled kao i korisničko iskustvo, a sve to korišćenjem dobro poznatih veb tehnologija. React Native razvijen je 2015. godine od strane kompanije Meta Inc., ranije poznata kao Facebook Inc. Od tada, sam radni okvir se dosta izmenio i postao je jedan od najpopularnijih alata za ravoj mobilnih aplikacija.

## 3.1 Način rada radnog okvira React Native

Osnovna ideja iza ovog radnog okvira jeste da kombinuje dve zasebne celine - JavaScript kod sa jedne i nativni kod sa druge strane (Java/Kotlin za Android [7] i Ojective-C/Swift za iOS [17]) i učini da njih dve međusobno sarađuju. Nativni kod će se izvršavati na samom uređaju, dok će za JavaScript kod biti neophodna virtuelna mašina. Korišćenje virtuelne mašine nam nije problematično na iOS uređajima iz razloga što oni poseduju ugrađeni JavaScript pokretač (eng. JS engine) nazvan JavaScriptCore [25]. Android uređaji nemaju ugrađen ovaj pokretač, međutim React Native rešava ovaj problem tako što donosi pokretač za Android uređaje. Potrebno je napomenuti da dodavanje pokretača rezultuje time da se veličina same aplikacije povećava za Android.

### Komunikacija između uređaja i JavaScript-a

Kako se programski jezici koji se koriste na samim uređajima razlikuju od JavaScript programskog jezika neophodno je da se kreira neki protokol kako bi oni mogli međusobno da komuniciraju. Ovo se postiže preko formata koji obe strane razumeju, a to je JSON [18]. Sva komunikacija između ovih delova sistema obavlja se preko mosta (eng. Bridge).

### Niti korišćene u React Native radnom okviru

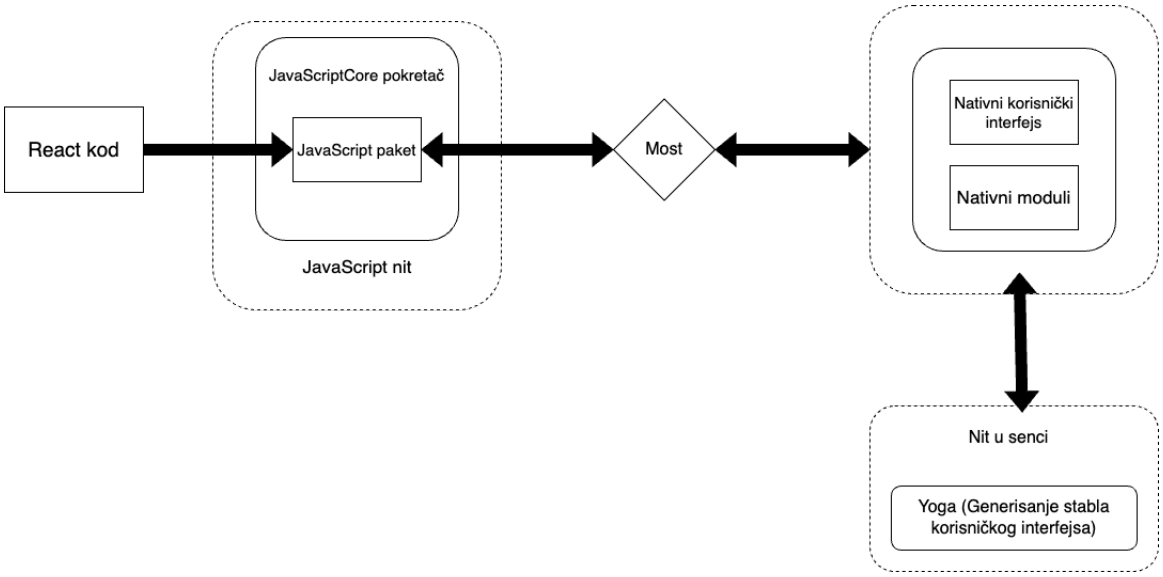
Kada korisnik pokrene React Native aplikaciju, sam uređaj će kreirati tri glavne niti [14] i ostale ukoliko za istim postoji potreba. Niti koje će biti kreirane su Glavna nit (eng. Main Thread), JavaScript nit (eng. JavaScript thread) i nit u senci (eng. Shadow Thread).

- Glavna nit [15] (eng. Main Thread) - ova nit poznata je i kao nit korisničkog interfejsa. To je osnovna nit svake iOS ili Android aplikacije i predstavlja glavnu nativnu nit na kojoj će se naša aplikacije izvršavati. Njena odgovornost jeste da obrađuje korisničke interakcije sa uređajem kao i da ažurira sam korisnički interfejs na ekranu uređaja. U React Native radnom okviru sve nativne komponente se prikazuju na ovoj niti i iz ovog razloga je veoma bitno da se izbegavaju složene operacije na ovoj niti kako ne bi došlo do blokiranja korisničkog interfejsa i smanjenja performansi.

- JavaScript nit (eng. JavaScript Thread) - na JavaScript niti će se izvršavati sav JavaScript i React kod rezvijane aplikacije. JavaScript nit obrađuje logiku aplikacije, API pozive, vrši proračune i upravlja stanjem aplikacije.Takođe, JS nit je ključna za sve što nije povezano sa korisničkim interfejsom. Ova nit radi zasebno od glavne niti što nam omogućava da radimo kompleksna izračunavanja bez da blokiramo glavnu nit.

- Nit u senci (eng. Shadow Thread) - ova nit poznata je jos i kao nit rasporeda (eng. Layout Thread) . Odgovornost ove niti jeste da preračunava pozicije elemenata i da generiše stablo za prikazivanje koje je kodirano u JavaScript niti. Kada se ovo izračunavanje završi ti podaci se šalju glavnoj niti koja vrši prikazivanje. Njena uloga jeste da oslobodi glavnu nit ovog posla, koji može biti veoma zahtevan. Ovaj pristup nam omogućava da imamo fluidno korisničko iskustvo.

Na slici 3.1 može se videti slikoviti prikaz svega objašnjenog u prethodnom paragrafu.

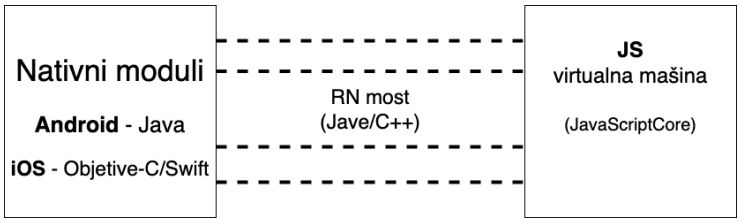



Slika 3.1: Arhitektura React Native radnog okvira

## 3.2 Most u React Native radnom okviru

U arhitekturi radnog okvira React Native, most [3, 165 - 185] (eng. bridge) igra ključnu ulogu u omogućavanju komunikacije između JavaScript niti i nativnih delova aplikacije koji se izvršavaju na glavnoj niti. Ovaj most funkcioniše kao posrednik koji prenosi informacije između dve sredine koje inače ne bi mogle direktno da komuniciraju zbog različitih programskih jezika (JavaScript sa jedne i Objective-C i Java sa druge strane). Komunikacija se obavlja asinhronom razmenom JSON poruka izmedju ova dva okruženja. JavaScript kod se izvršava u svom okruženju, dok se nativne komponente upravljaju kroz specifične platforme kao što su iOS ili Android. Primer ove komunikacije prikazan je na slici 3.2. Most omogućava asinhronu komunikaciju između ova dva okruženja prenošenjem poruka, što znači da informacije mogu preći iz jednog okruženja u drugo bez blokiranja glavne niti ili usporavanja aplikacije. Kada JavaScript kod zahteva pristup nativnim funkcionalnostima (kao što su pristup kameri ili GPS-u) ili kada treba da se ažurira korisnički interfejs, most prenosi ove zahteve iz JavaScript-a u nativni sistem, i obratno, prenosi odgovore nazad. Ovaj mehanizam omogućava React Native aplikacijama da koriste prednosti kako

veb tehnologija, tako i nativnih performansi, iako ponekad može predstavljati usko grlo u performansama ako se prevelik broj zahteva mora preneti preko mosta.




Slika 3.2: Funkcionisanje mosta

## Primer rada mosta

U nastavku će biti analiziran primer kako most izvodi komunikaciju između nativnog dela i JavaScript dela mobilne aplikacije.

1. Okida se nativni događaj. Recimo da je u pitanju klik na dugme.
2. Serijalizovana poruka šalje se sa nativne strane kroz most sa svim neophodnim podacima
3. JavaScript prima poruku, deserijalizuje je i odlučuje šta je sledeći korak koji je potrebno preduzeti. U ovom slučaju to je akcija koja je pridružena dugmetu.
4. Šalje se poruka sa JavaScript strane kroz most sa svim potrebnim informacijama vezanim za traženu akciju.
5. Nativna strana prima poruku, deserijalizuje je i vrši ponovno renderovanje korisničkog interfejsa.

## Nedostaci mosta

Asinhrona priroda prenosa poruka između nativnog i JavaScript dela mobilne aplikacija dovodi do problema koji se manifestuju u ivičnim slučajevima (eng. edge cases). Na primer, pretpostavimo da postoji polje za unos teksta gde korisnik unosi broj kartice. Prema specifikaciji potrebno je da se formatira unos tako što će biti ubačen blanko znak posle svakog četvrtog znaka u broju kartice. Ova logika biće implementirana na JavaScript strani. Problem koji se ovde uočava jeste taj da kada korisnik pritisne peti broj po redu nativna strana aplikacije to javlja JS strani. Broj

se nadovezuje na prethodni broj kartice i to se javlja nativnoj strani koja će ovo prikazati. Rezultat prethodnog izvršavanje jeste to da će prvo biti prikazan broj bez razmaka, a tek kasnije će se izvršiti formatiranje i preko mosta će biti poslata sledeća poruka koja će nativnoj strani reći da treba da prikaže i razmak. Naravno, ukoliko most nije previše opterećen u tom momentu korisnik ne bi mogao da primeti situaciju kada razmaka nema. Međutim, u situaciji kada se ogroman broj operacija izvršava paralelno, situacija gde imamo ponašanje koje nije po specifikacije postaje realna. Drugi problem koji se uočava jeste da se svaka poruka mora serijalizovati na jednoj i deserijalizovati na drugoj strani, što dovodi do povećane potrošnje resursa.

### Rešenja prethodno navedenih problema

React Native tim adresirao je prethodne probleme i pristupio kreiranju načina da se isti prevaziđu. Kao rezultat toga, od verzije 0.68 može da se koristi potpuno nova arhitektura koja odbacuje mehanizam mosta i počinje da koristi JavaScript interfejs (eng. JavaScript Interface - JSI). JSI predstavlja sloj opšte upotrebe koji može biti ugrađen u bilo koji JS pokretač i pomoću njega može se kreirati direktna konekcija ka nativnim interfejsima. Ovaj napredak postignut je tako što su nativni Java ili Obj-C metodi izloženi JS-u preko objekta domaćina (eng. HostObject). JS će čuvati referencu na ovaj objekat i preko njega će pristupati nativnim interfejsima.

# Glava 4

# Aplikacija za vremensku prognozu

U poglavlju koje sledi u nastavku bliže će biti opisana aplikacija za vremensku prognozu koja će se koristiti za demonstraciju koncepata koji se analiziraju u okviru ovog master rada, kao i ključne tehnologije korišćene prilikom izrade iste.

## 4.1 Korisnički interfejs aplikacije

U ovoj sekciji detaljno će biti predstavljen korisnički interfejs aplikacije. Po definiciji, korisnički interfejs jeste jedan od najznačajnijih delova same aplikacije. Vrlo je bitno da isti bude jednostavan za korišćenje, intuitivan, ali i prijatan za oko, odnosno lepo dizajniran. Sve prethodno nabrojane osobine korisničkog interfejsa osiguraće zadovoljstvo korisnika aplikacije. U ovom konkretnom slučaju aplikacija se sastoji od dva glavna ekrana i oni su predstavljeni na slici 4.1. Prvi ekran jeste taj na kome se pokazuju svi parametri vremenske prognoze, dok je drugi ekran odgovoran za pretragu destinacije za koju nam je potrebna vremenska prognoza. Pored ova dva ekrana postoji i ekran za izbor jezika unutar aplikacije.

(a) Glavni ekran aplikacije (b) Ekran za pretragu destinacije

Slika 4.1: Ekrani aplikacije

## Ekran za vremensku prognozu

Glavni ekran aplikacije predstavlja ekran sa svim parametrima vremenske prognoze. Sastoji se od pet sekcija koje će biti analizirane u nastavku poglavlja.

1. Prva sekcija predstavlja podatke o trenutnoj lokaciji i temperaturi. Takođe, u okviru ove sekcije nalaze se podaci o maksimalnoj i minimalnoj dnevnoj temperaturi kao i o tome kakvo je trenutno vreme.

2. Nakon ovoga nalazi se dugme koje nas navigira na ekran za pretragu drugih destinacija na kojima želimo da vidimo vremensku prognozu.

3. Zatim sledi lista koja prikazuje temperature na datoj lokaciji u naredna dvadeset četiri časa. U pitanju je horizontalna, skrolabilna optimizovana lista. U kasnijim poglavljima biće dodatno reči o optimizacijama listi.

4. Ispod liste temperatura se nalazi komponenta koja prikazuje grafike kretanja određenih parametara kroz vreme. Posmatra se period od dvadeset četiri časa i korisnik je u mogućnosti da selektuje parametar koji ga interesuje.

5. Poslednja sekcija predstavlja vremensku prognozu za naredne dane. Realizovana je takođe kao optimizovana lista, sa izmenom da je sada vertikalna lista u pitanju.

### Ekran za pretragu destinacije

Drugi ekran u aplikaciji služi za pretragu destinacije na kojoj korisnik želi da pogleda vremensku prognozu. Korisnički interfejs na ovom ekranu je realizovan jednostavnije u odnosu na prvi ekran. Kako je prikazano na slici, izdvajaju se dve glavne celine.

1. Na samom vrhu ekrana nalazi se navigacijsko zaglavlje koje je izmenjeno tako da bude komponenta za unos teksta. U okviru ovog zaglavlja korisnik preko tastature unosi željenu lokaciju i preko aplikacijskog programskog interfejsa (eng. Application Programming Interface - API) dohvatamo niz lokacija za uneti teskt.

2. Uloga druge sekcije jeste da se prikažu podaci koje nam je API vratio. Realizovana je takođe kao optimizovana lista.

### Lokalizacija i internacionalizacija

U kontekstu mobilnih aplikacija, lokalizacija (eng. localization) i internacionalizacija (eng. internationalization) su ključni koncepti koji se odnose na prilagođavanje aplikacije različitim jezičnim i kulturnim postavkama kako bi se omogućilo korisnicima širom sveta da je koriste na način koji im je najprihvatljiviji.

Internacionalizacija je proces dizajniranja aplikacije na način koji omogućava jednostavno prilagođavanje različitim jezičnim i kulturnim postavkama bez značajnih izmena u samom kodu. To uključuje korišćenje standarda kao što su Unicode za podršku različitim jezicima, podršku za različite formate datuma, valuta i drugih lokalnih specifičnosti. Sa druge strane, lokalizacija se odnosi na prilagođavanje internacionalizovane aplikacije specifičnom jeziku i kulturnim normama određenog regiona. Ovo uključuje prevođenje teksta, prilagođavanje formata datuma i vremena,

valute, kao i prilagođavanje ikonica, slika i drugih sadržaja kako bi se bolje uklopili u ciljno okruženje.

Kao što je napomenuto ranije u ovom poglavlju, aplikacija podržava promene jezika. U ovom momentu podržana su samo dva jezika - engleski i srpski. U ovoj sekciji biće prikazana implementacija i analziran pristup kojim je internacionalizacija i lokalicazicija uvedena.

Implementacija je izvršena pomoću biblioteke i18next. Nakon instaliranja iste u projekat bilo je neophodno izvršiti konfigurisanje kao što je prikazano na slici 4.2.

```
const currLang = useSelector(selectCurrentLanguage);

use(initReactI18next).init({
  compatibilityJSON: 'v3',
  resources: locales,
  lng: currLang,
  interpolation: {escapeValue: false},
  react: {
    useSuspense: false,
  },
});

changeLanguage(currLang);
```

Slika 4.2: Konfiguracija biblioteke za lokalizaciju

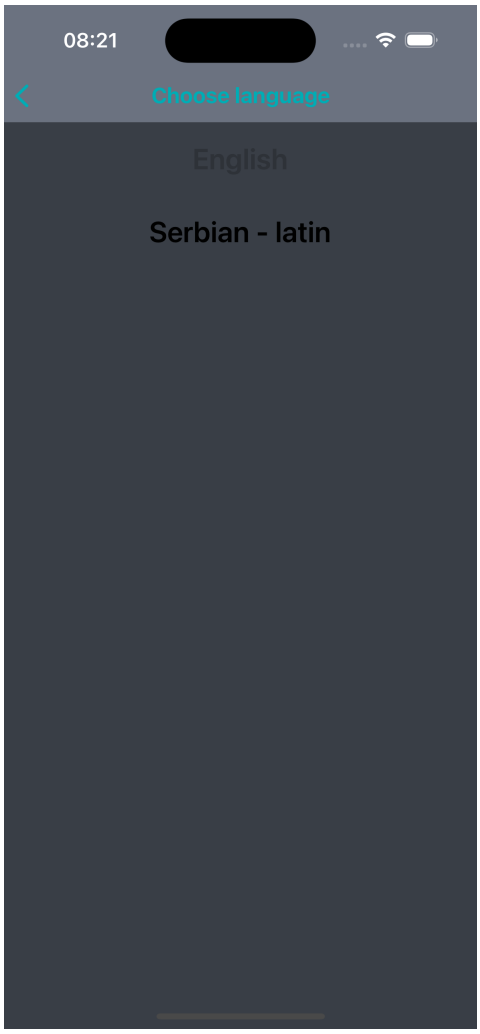
Kao što je prikazano na slici 4.2, biblioteci se daju fajlovi sa svim podržanim jezicima putem parametra *resources*. U ovom konkretnom slučaju to su JSON fajlovi koji sadrže prevode na srpski i engleski jezik. Putem parametra *lng* se postavlja početni jezik i on je podešen automatski na region koji je postavljen unutar podešavanja uređaja.

Pored automatskog postavljanja jezika, aplikacija poseduje i zaseban ekran na kom su izlistani svi dostupni jezici i tu korisnik može menjati jezik po volji. Ovaj ekran je prikazan na slici 4.3.

Takođe, bitno je napomenuti da je implementacija izvedena tako da prilikom dodavanja novog jezika nije potreban rad sa inženjerske strane. Neophodno je dodati novi fajl za jezik koji se uvodi i prevesti sav tekst koji se koristi. Rezultat ove funckionalnosti prikazan je na slici 4.4.

## WeatherAPI

U svrhe dohvatanja podataka potrebnih za izradu ove aplikacije korišćen je WeatherAPI [23]. U pitanju je API koji nam daje nekoliko krajnjih tačaka (eng. endpo-




Slika 4.3: Ekran za izbor jezika

int). Konkretno, za rad opisan ovim radom korišćene su dve krajnje tačke Weather API-ja:

1. Forecast API - ovaj API služi za dohvatanje podataka o vremenskoj prognozi. Odgovor servera sastoji se iz tri celine. Podaci o trenutnoj lokaciji, podaci o trenutnom vremenu i podaci o vremenskoj prognozi za naredne dane. Primer API poziva za Forecast API se vidi na slici 4.5.

2. Search API - uloga Search API krajnje tačke jeste da vrati sve lokacije koje počinju nekim prefiksom koji je korisnik uneo prilikom pretrage.

(a) Aplikacija na engleskom (b) Aplikacija na srpskom

Slika 4.4: Aplikacija na više jezika

```
export const getForecast = (
  location: string = 'Belgrade',
): AxiosPromise<ForecastType> =>
  customAxios.get(
    `https://api.weatherapi.com/v1/forecast.json?key=${API_KEY}&days=3&q=${location}&aqi=yes&alerts=no`,
  );
```

Slika 4.5: API za vremensku prognozu

# Glava 5

# Sistem za merenje performansi aplikacije

U savremenom razvoju softvera, efikasnost i performanse jedan su od ključnih aspekata za uspeh na tržištu. Uspeh neke aplikacije direktno zavisi od zadovoljstva korisnika, a to zadovoljstvo nemoguće je postići ukoliko aplikacija nije performantna. Ovo jeste razlog za razmatranje i uvođenje sistema za merenje performansi čiji je zadatak da pruži jasnu sliku o ponašanju aplikacije u različitim uslovima.
U ovom poglavlju će biti analizirani različiti sistemi za merenje performansi. Detaljno će biti opisan sistem Sentry [21]. Biće predstavljeni neophodni koraci za integraciju ovog sistema u aplikaciju, nakon čega će biti analzirano šta ovaj sistem pruža u kontekstu praćenja i analize performansi mobilne aplikacije. Takođe, pažnja će biti posvećena i optimizovanim listama kao jednom od ključnih koncepata u razvoju mobilne aplikacije.

## 5.1 Alati za merenje performansi

Kao što je prethodno navedeno, postoji veliki broj dostupnih platformi koje omogućavaju integraciju sistema za merenje performansi u mobilnu aplikaciju. Pre nego što se autor upusti u taj korak neophodno je upoznati se sa svim alatima i odlučiti koji od tih istih najviše odgovara potrebama konkretne aplikacije. U nastavku ove sekcije biće predstavljeno nekoliko najpopularnijih alata sa svojim prednostima i manama.

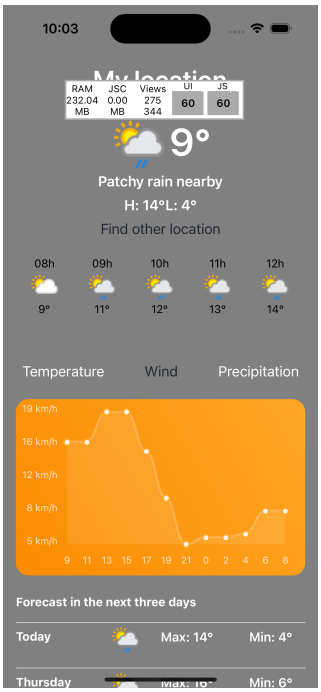
## React Native Performance Monitor

React Native Performance Monitor je ugrađeni alat koji dolazi u okviru React Native radnog okvira. On nam pruža osnovne informacije o performansama koje uključuju:

1. Praćenje broja frejmova u sekundi (eng. Frame Per Second - FPS) - možemo pratiti dva seta FPS metrika. Jedan jeste za JavaScript nit, dok je drugi za glavnu, odnosno nit korisničkog interfejsa. Na ovaj način možemo u realnom vremenu da pratimo pomenute FPS metrike i identifikujemo zbog čega dolazi do pada performansi u nekom određenom momentu. Optimalno je da obe ove metrike budu blizu 60 FPS.

2. Upotreba memorije - druga metrika koju React Native Performance Monitor pruža jeste praćenje količine radne memorije koju aplikacija koristi u nekom momentu. Pomenuta metrika nam je veoma korisna za pronalaženje curenja memorije ili drugih problema vezanih za upravljanje memorijom.

Primer izgleda i funkcionalnosti koje pruža ovaj ugrađeni program za otklanjanje grešaka (eng. debugger) prikazan je na slici 5.1.

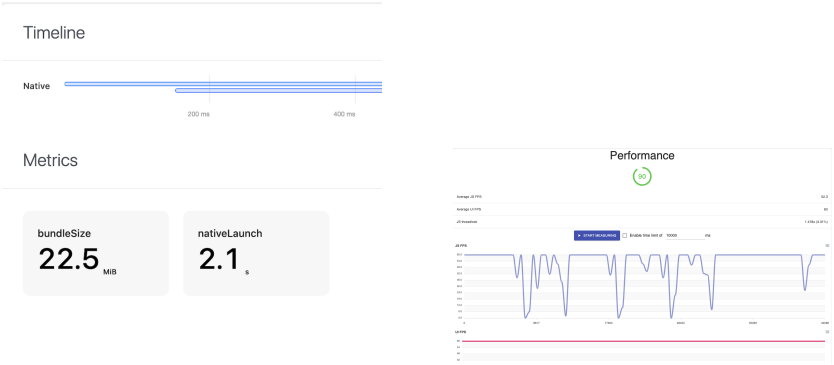



Slika 5.1: React Native Performance Monitor

Iako je React Native Performance Monitor koristan za osnovno praćenje performansi aplikacije, on nije dovoljno detaljan i ne pruža dublji uvid u performanse aplikacije.

### Flipper

Flipper [8] (eng. Flipper) je platforma za otkrivanje i analizu grešaka (eng. debugging) koja je posebno dizajnirana za mobilne aplikacije, uključujući one razvijene pomoću React Native radnog okvira. Jedan od ključnih dodataka unutar Flipera jeste *performance monitoring* dodatak koji omogućava programerima detaljan uvid u performanse aplikacije. Ovaj dodatak pruža kompletan set alata za analizu kako se aplikacija ponaša u realnom vremenu, omogućavajući programerima da identifikuju sporije delove aplikacije, analiziraju upotrebu resursa i dijagnostifikuju probleme koji utiču na performanse. Uz mogućnost vizualnog prikaza performansi, kao što su FPS (eng. Frame Per Second) grafici i merni podaci o memoriji, Flipper nudi jednostavan i intuitivan interfejs za praćenje ključnih performansi aplikacije. Prikaz ovih podataka može se videti na slici 5.2. Takođe, alat podržava razne dodatke koji se mogu lako integrisati i koristiti za specifične potrebe projekta, što Flipper čini nezamenjivim alatom u arsenalu inženjera za ravoj mobilnih aplikacija.




(a) Osnovne metrike (b) Merenje FPS-a

Slika 5.2: Flipper metrike

## 5.2 Sentry

U ovoj sekciji detaljno će biti analiziran alat Sentry. On predstavlja veoma snažan alat za nadgledanje grešaka i merenje performansi i veoma je popularan među

programerima koji se bave optimizacijom i održavanjem zdravlja aplikacije u realnom vremenu. U kontekstu merenja performansi mobilnih aplikacija, Sentry nudi napredne funkcionalnosti koje programerima omogućavaju da precizno detektuju i analiziraju izvore problema koji utiču na korisničko iskustvo. Ovaj alat ne samo da automatski hvata greške unutar aplikacije, već pruža i detaljne uvide u performanse aplikacija, beležeći vremena odziva, broj grešaka i kroz analizu transakcija omogućava duboko razumevanje performansi aplikacije. Pomoću alata Sentry, timovi mogu brzo reagovati na probleme, optimizovati aplikacije i značajno poboljšati stabilnost i brzinu aplikacija.

## Intregracija sistema Sentry

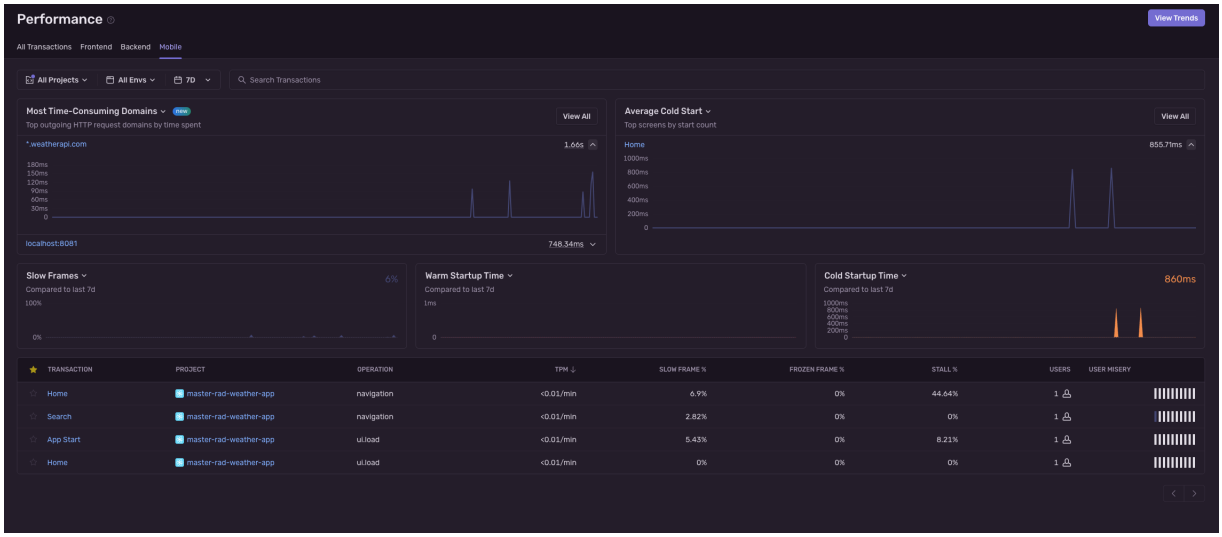
Prvi korak ka korišćenju sistema Sentry jeste integracija istog u mobilnu aplikaciju. Ovaj korak je vrlo jednostavan i detaljno je objašnjen u dokumentaciji samog servisa. Za sam početak, neophodno je da se instalira biblioteka preko paketnog menadžera (eng. package manager). Nakon instalacije, potrebno je konfigurisati Sentry u glavnom fajlu aplikacije, gde se inicijalizuje Sentry sa odgovarajućim DSN (eng. Data Source Name) koji povezuje aplikaciju sa Sentry projektnim okruženjem. Ova konfiguracija omogućava Sentry-ju da automatski hvata izuzetke i greške u aplikaciji, kao i da pruža opcionalne funkcionalnosti poput ručnog slanja izveštaja o greškama, praćenje performansi i analizu korisničkih sesija. Unutar same konfiguracije bitno je podesiti procenat transakcija koje želimo da prosleđujemo na Sentry. Ovo se postiže uzorkovanjem i koristi se opcija *tracesSampleRate* koja uzima vrednosti od 0 do 1. Drugi bitan parametar konfiguracije jesu sve integracije koje su nam potrebne. Pomenuti parametar predstavlja vezu između biblioteka koje koristimo u aplikaciji i Sentry alata. Na primer, u aplikaciji koja se razvija u okviru ovog master rada je potrebno da se doda integracija za navigacione transakcije kako bi Sentry znao kako da obradi iste. Sve pomenute informacije možemo videti i analizirati na slici 5.3.

## Metrike alata Sentry za merenje performansi aplikacije

Kada je uspešno odrađena integracija alata Sentry sa mobilnom aplikacijom, vreme je da se analizira šta sve ovaj servis stavlja na raspolaganje. Ekran koji se vidi kada u okviru alata Sentry korisnik navigira na odeljak za performanse izgleda kao na slici 5.4.

```
const routingInstrumentation = new Sentry.ReactNavigationInstrumentation({
  enableTimeToInitialDisplay: true,
});

Sentry.init({
  dsn: 'https://626ccbd77d09e9a1949cf8029d1a015b@o4506208597180416.ingest.us.sentry.io/4506208611729408',
  integrations: [new Sentry.ReactNativeTracing({routingInstrumentation})],
  tracesSampleRate: 1,
});

function App(): JSX.Element {
  return (
    <Provider store={store}>
      <SafeAreaView style={{flex: 1}}>
        <NavigationComponent routingInstrumentation={routingInstrumentation} />
      </SafeAreaView>
    </Provider>
  );
}

export default Sentry.wrap(App);
```

Slika 5.3: Sentry integracija u kodu




Slika 5.4: Metrike za performanse koje nam pruža alat Sentry

U nastavku će biti analizirane dostupne metrike [20] i biće predstavljeno šta one u stvari označavaju.

1. Vreme inicijalnog pokretanja (eng. Cold Startup time) - jedna od primarnih performansnih metrika u razvoju mobilnih aplikacija jeste vreme inicialnog pokretanja. Ova metrika meri vreme potrebno da se aplikacija pokrene i postane potpuno funkcionalna nakon što je korisnik pokrenuo aplikaciju s potpuno ne-

aktivnog ili „hladnog” stanja, odnosno kada aplikacija nije u memoriji uređaja. Ovo uključuje vreme potrebno za učitavanje aplikacije u memoriju, inicijalizaciju aplikacije, učitavanje početnih resursa i prikaz prvog ekrana korisniku. Brže vreme pokretanja može značajno poboljšati percepciju aplikacije od strane korisnika, smanjujući frustraciju i povećavajući zadovoljstvo. Pored samog korisničkog iskustva, ova metrika se u praksi koristi i za definisanje uskih grla prilikom pokretanja aplikacije. Često se tu dovlače resursi i vrše kalkulacije koje nisu potrebne za samo pokretanje aplikacije i neophodno ih je odložiti za kasnije. Kao referentna vrednost za produkcijske verzije aplikacija uzima se 400ms.

2. Vreme reaktiviranja aplikacije (eng. Warm startup time) - ova metrika predstavlja vreme da se aplikacija reaktivira i postane potpuno funkcionalna iz tzv. „toplog” stanja, kada je aplikacija već delimično učitana u memoriji uređaja, ali nije aktivna. Ovo može uključiti situacije kada je aplikacija minimizirana ili kada korisnik prelazi iz aplikacije u aplikaciju. Ova metrika se fokusira na brzinu kojom aplikacija može ponovo postati aktivna i spremna za interakciju, što je ključni faktor za korisničko iskustvo, posebno u situacijama kada korisnici očekuju brz odgovor aplikacije nakon prekida korišćenja. Praćenje i optimizacija ove metrike su važni za poboljšanje fluidnosti i responzivnosti aplikacije, što može značajno uticati na zadovoljstvo korisnika i percepciju performansi aplikacije. Inženjeri koriste ovu metriku za identifikaciju problema u upravljanju resursima ili kodnim blokovima koji mogu usporiti reaktivaciju aplikacije, te rade na njihovom otklanjanju kako bi optimizovali ukupno korisničko iskustvo.

3. Spori kadrovi (eng. Slow frames) - dobre performanse aplikacije se često poistovećuju sa fluidnim radom iste. To se, pored posmatranja golim okom, može izmeriti i jedna od metrika koja se koristi za to jeste broj sporih kadrova. Odnosi se na broj kadrova (eng. frames) u animacijama ili prikazivanju sadržaja na ekranu koji se ne izvršavaju u optimalnom vremenskom intervalu, što dovodi do zaostajanja (eng. lagging) i seckanja (eng. stuttering). Tipično, aplikacije bi trebalo da održavaju stopu od 60 sličica po sekundi (eng. frame per second - FPS) za glatko i prirodno korisničko iskustvo. Kadrovi koji se izvršavaju sporije od ove stope klasifikuju se kao „spori”. Ova metrika je ključna za razvojne timove kako bi razumeli kako njihova aplikacija performira u stvarnim uslovima korišćenja, posebno kada su u pitanju zahtevnije operacije

kao što su animacije, listanje sadržaja ili tranzicije između ekrana. Praćenje sporih kadrova omogućava programerima da identifikuju i optimizuju delove aplikacije koji narušavaju fluidnost, tako što prilagođavaju animacije, optimizuju prikazivanje komponenti ili redukuju opterećenje centralnog ili grafičkog procesora.

4. Zamrznuti kadrovi (eng. Frozen frames) - ova metrika odnosi se na period kada aplikacija ne reaguje i ne osvežava svoj prikaz, što rezultuje zaustavljanjem interaktivnosti i vizuelnog prikaza na korisničkom interfejsu. Ova pojava se obično dešava kada aplikacija naleti na teške ili blokirajuće operacije koje zaguše glavnu nit, čineći da aplikacija postane neoperativna za milisekunde ili čak sekunde. Takvi zamrznuti kadrovi mogu značajno narušiti korisničko iskustvo, jer korisnici očekuju brzu i glatku interakciju sa aplikacijama. Praćenje i analiziranje zamrznutih kadrova neophodno je za otkrivanje i razrešenje uzroka ovih performansnih problema. To uključuje identifikovanje i optimizaciju teških procesa koji opterećuju CPU (eng. Central Processing unit) ili preduzimanje koraka za asinhrono izvršavanje dugotrajnih zadataka. Time se osigurava da korisnički interfejs ostaje responzivan i prijatan za korišćenje, što je ključno za održavanje visokog nivoa zadovoljstva korisnika.

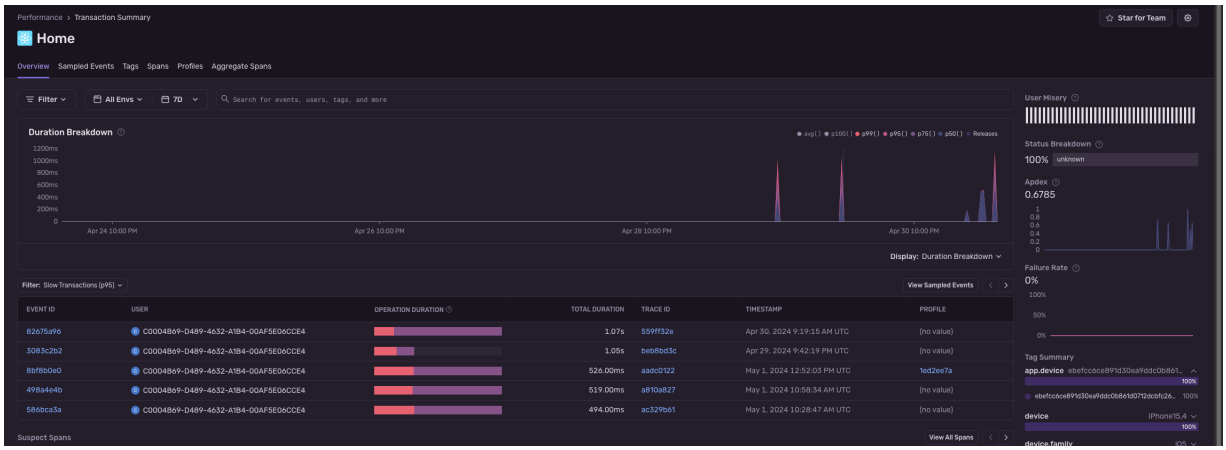
5. Korisničko nezadovoljstvo (eng. User misery) - ovo je interna Sentry metrika. Predstavlja broj različitih korisnika koji su iskusili vreme učitavanje četiri puta veće od podešene granice na nivou projekta.

Kao što se vidi na slici, korisnički interfejs alata Sentry pruža korisniku potpunu mogućnost da postavlja metrike koje su njemu relevantne. Dakle, prilikom postavljanja sistema za praćenje performansi, razvojni tim mora da definiše metrike koje želi da prati i da ih postavi u samom alatu. Sentry podržava i kreiranje sopstvenih metrika ukoliko razvojni tim zaključi da im je potrebna metrika koja nije podržana od strane samog alata.

## Performanse ekrana

Alat Sentry daje podršku za praćenje performansi pojedinačnih ekrana u aplikaciji i to se vidi na slici 5.5.

Na prethodnoj slici prikazano je da da za svaki ekran postoji lista transakcija koje uključuju navigiranje na isti i učitavanje svih potrebnih podataka za prikaz.



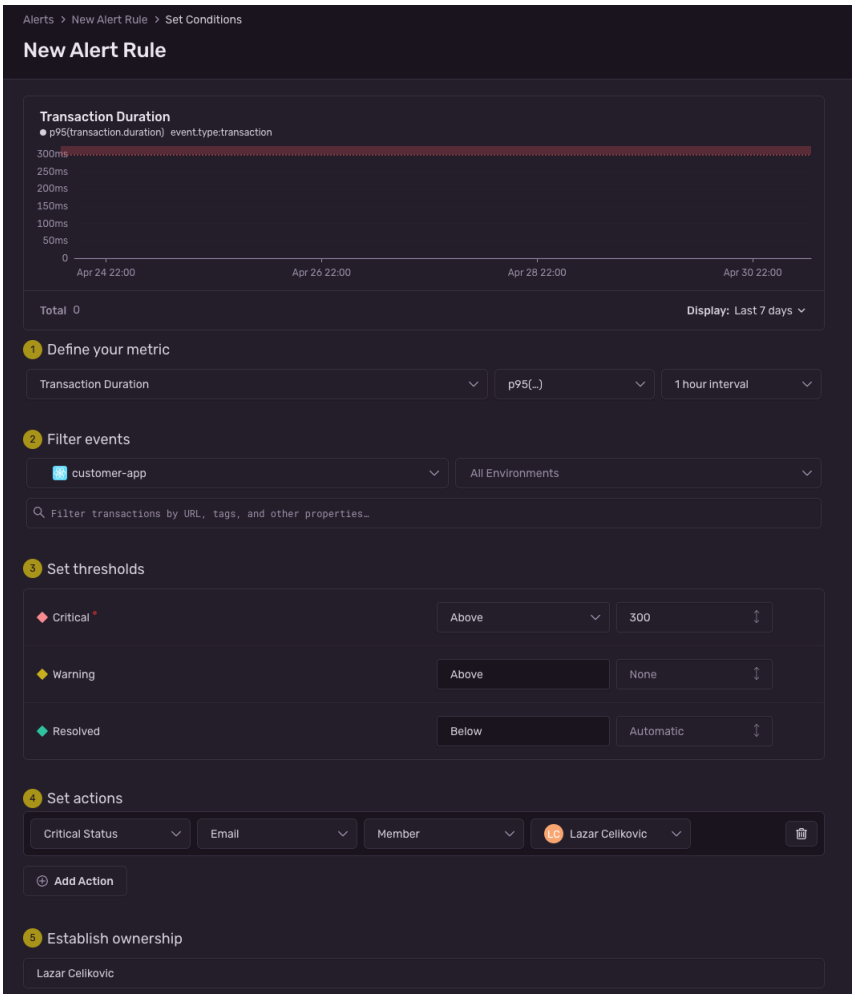

Slika 5.5: Metrike za glavni ekran aplikacije

Sentry daje informacije o tome koliko je trajalo dohvatanje svih potrebnih podataka sa servera (prikazano narandžastom bojom), i trajanje renderovanja korisničkog interfejsa (prikazano ljubičastom bojom).

## Praćenje metrika

Nakon što je uspešno integrisan sistem za praćenje performansi i postavljene sve metrike koje razvojni tim želi da prati neophodno je postaviti sistem za obaveštavanje. Naime, vrlo je nepraktično da neko konstantno proverava sve metrike koje se prate i da reaguje ukoliko naiđe na nešto problematično. Ovakav pristup zahtevao bi da neko konstantno prolazi i gleda metrike kao i da zna tačno kako iste treba da se ponašaju. Jasno je da je ovakav pristup dugoročno neodrživ. Sentry ovo prevazilazi konceptom obaveštenja ili uzbune (eng. Alert). Naime, korisnik alata može da definiše granicu kada je neka metrika na nezadovoljavajućem nivou i da kaže alatu da ukoliko ista ikada padne ispod te granice da se podigne uzbuna i da obavesti razvojni tim o problemu. Na ovaj način bismo morali jednom da postavimo ove okidače i da reagujemo samo kada se podigne uzbuna. U međuvremenu, praktično bismo mogli da zaboravimo na ove metrike.

U konkretnom slučaju, na slici 5.6 dadata je nova uzbunu za metriku *Dužina trajanja transakcije*. U suštini, pravilo koje je dodato glasi: „Ukoliko u intervalu od sat vremena, preko 95% transakcija ima trajanje veće od 300ms okini uzbunu i obavesti korisnika putem mejla.”




Slika 5.6: Dodavanje uzbune za metriku

## 5.3 Optimizacija listi za prikazivanje

Liste su jedan od ključnih elementa u dizajnu mobilnih aplikacija, omogućavajući korisnicima efikasno i intuitivno pregledanje sadržaja. One služe kao ključna komponenta za organizovanje podataka u lako dostupan i razumljiv format, što je posebno važno u mobilnom okruženju gde je prostor ograničen i korisnička pažnja kratkotrajna. U programiranju mobilnih aplikacija, efektivna implementacija lista može značajno poboljšati korisničko iskustvo, pružajući brz pristup raznovrsnim informacijama. Bilo da se radi o listi kontakata, podešavanja ili kao u slučaju aplikacije razvijane u ovom radu, vremena po danima ili satima, liste omogućavaju korisnicima da lako navigiraju kroz aplikaciju i brzo pristupe željenim informacijama. Uz to, napredne tehnike poput lenjog učitavanja (eng. lazy loading) i beskonačnog listanja (eng. infinite scrolling) mogu dalje optimizovati performanse i korisnički doživljaj,

čineći liste još moćnijim alatom u arsenalu svakog programera mobilnih aplikacija.

## Komponenta FlatList i njeni problemi

Komponenta FlatList [2] u React Native radnom okviru je dizajnirana za efikasno prikazivanje dugačkih lista podataka, pružajući osnovne funkcionalnosti za upravljanje velikim količinama elemenata. Međutim, uprkos svojoj nameni, FlatList može susresti izazove sa performansama u određenim scenarijima. Problemi se najčešće javljaju kada lista sadrži složene elemente ili kada se učitava veliki broj podataka, što može dovesti do usporenja i povećane potrošnje memorije.

Kada svaki element liste zahteva intenzivno procesiranje ili ima kompleksne vizualne komponente, FlatList može doživeti pad performansi. Takođe, česti zahtevi za ponovno prikazivanje (eng. rerender) izazvani promenama u stanju ili osobinama (eng. props) koji se ne upravljaju pažljivo mogu dodatno opteretiti procesor i memoriju. Da bi se ovi problemi minimizovali, preporučuje se pažljiva optimizacija renderovanja elemenata, upotreba tehnika kao što su memoizacija za skuplje komponente, i efikasno upravljanje stanjem da se izbegnu nepotrebna ažuriranja. Implementacija paginacije (eng. windowing) gde se renderuju samo trenutno vidljivi elementi takođe može značajno poboljšati performanse FlatList komponente. Ovim pristupima osigurava se da FlatList ostane efikasan alat za upravljanje listama u aplikacijama koje razvijate koristeći React Native. Primer konkretne upotrebe vidi se na slici **??**. Implementacije sa slike **??** biće kasnije upotrebljena kao referenca za poređenje sa interfejsom drugih biblioteka.

```
  return (
    <View style={styleList}>
      <FlatList
        data={hourlyData}
        renderItem={renderItem}
        horizontal
        showsHorizontalScrollIndicator={false}
        keyExtractor={(_, index) => index.toString()}
      />
    </View>
  );
};

export default DayWeatherList;
```

Slika 5.7: Flatlist komponenta u kodu

Sa slike iznad vidi se API FlatList komponente i zaključuje se da je isti veoma

intuitivan. Naime, dva glavna polja jesu *data* i *renderItem*. Preko *data* polja prosleđujemo niz podataka komponenti, dok se preko polja *renderItem* govori na koji način treba da se prikaže svaki element tog niza. Polje *keyExtractor* služi kako bi listu optimizovali do neke mere, odnosno da svaki element liste ima svoj identifikator. Na ovaj način, komponenta razlikuje svaki pojedinačni element i pomoću tog znanja se odlučuje koje elemente treba ponovo renderovati kada dođe do promena u podacima.

## Prvi pokušaj rešavanja ovog problema

Kako je problem sa performansama FlatList komponente bio prilično značajan, brzo je adresiran u krugovima programera mobilnih aplikacija. Kao rešenje ovog problema kreirana je RecyclerListView komponenta [16]. RecyclerListView predstavlja veoma dobro optimizovanu komponentu za prikaz ogromnog broja podataka. Inspirisana je RecyclerView komponentom na Androidu i UICollectionView na iOS operativnom sistemu.

RecyclerListView komponenta je veoma performantna i bogata funkcionalnostima kao i ugrađena FlatList komponenta. Funkcioniše tako što se na pametan način vrši recikliranje komponenti koje su prethodno već prikazane i na taj način izbegava nepotrebno ponovno prikazivanje. Ova tehnika poznata je kao reciklaža ćelija (eng. cell recycling) i dozvoljava nam da koristimo već prikazane komponente koje nisu više prisutne na ekranu za prikaz novih.

Problem ove komponente jeste veoma neintuitivan programski interfejs. Kao programer, morali ste potrošiti ogromnu količinu vremena da razumete kako sama komponenta radi da biste mogli da iskoristite prednosti koje ista pruža. Komponenta zahteva precizne predikcije za veličinu elemenata i ako iste nisu dobre performanse dosta padaju. Takođe, sav prikaz se vrši na JavaScript niti što može dovesti na seckanja.

## FlashList biblioteka

Rešenje problema pomenutih u prethodnim sekcijama ogleda se u biblioteci po imenu FlashList [22]. Ona predstavlja performantnu alternativu za FlatList komponentu. Kao što smo rekli, FlatList komponenta nije imala zadovoljavajuće performanse, ali je bila laka za korišćenje, dok je RecyclerListView komponenta bila performantna, ali je njen API bio izrazito neintuitivan. Ideja biblioteke FlashList

jeste da iskoristi najbolje iz oba pristupa i da kreira komponentu koja će biti veoma performanta, ali će zadržati API tako da korisnicima bude laka za korišćenje. Na slici 5.8 se vidi kako izgleda interfejs i može se zaključiti da skoro da i nema promena u odnosu na FlatList biblioteku.

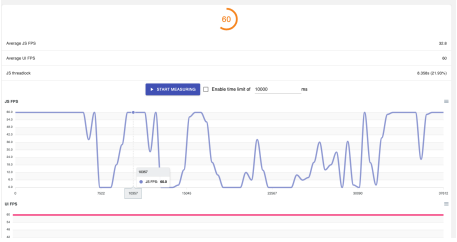
```
  return (
    <View style={styleList}>
      <FlashList
        data={hourlyData}
        renderItem={renderItem}
        estimatedItemSize={40}
        horizontal
        showsHorizontalScrollIndicator={false}
      />
    </View>
  );
};

export default DayWeatherList;
```

Slika 5.8: Flashlist komponenta u kodu

Sa slike iznad vidi se kako izgleda korišćenje FlashList komponente u samom kodu. Prva stvar koja se primećuje jeste da je API skoro identičan sa FlatList komponentom. Jedina razlika jeste u *estimatedItemSize* polju koje je specifično za FlashList biblioteku.

FlashList komponenta koristi estimatedItemSize polje kako bi odredila broj elemenata koji treba da budu prikazani na ekranu tokom inicijalnog renderovanja i prilikom listanja. Način funkcionisanja je sledeći. Naime, umesto da se komponenta uništava kada neki element nestane sa ekrana, FlashList komponenta renderuje istu komponentu sa drugačijim parametrima. Prilikom korišćenja, bitno je da se ne koristi key polje u elementima liste jer će ovo sprečiti FlashList komponentu da radi recikliranje.

Na slici 5.9 se vide performanse prilikom korišćenja FlatList i FlashList komponenti. Primećuje se da je razlika u performansama drastična. Vidi se da prilikom korišćenja FlatList komponente postoji relativno često blokiranje JS niti, dok kod FlashList komponente to nije slučaj. Blokade se dešavaju na momente i to bas u momentu kada se dohvataju podaci sa servera (eng. Backend).




(a) FlatList performanse

(b) FlashList performanse

Slika 5.9: Performanse FlatList i FlashList listi

# Glava 6

# Sistem za praćenje događaja u mobilnoj aplikaciji

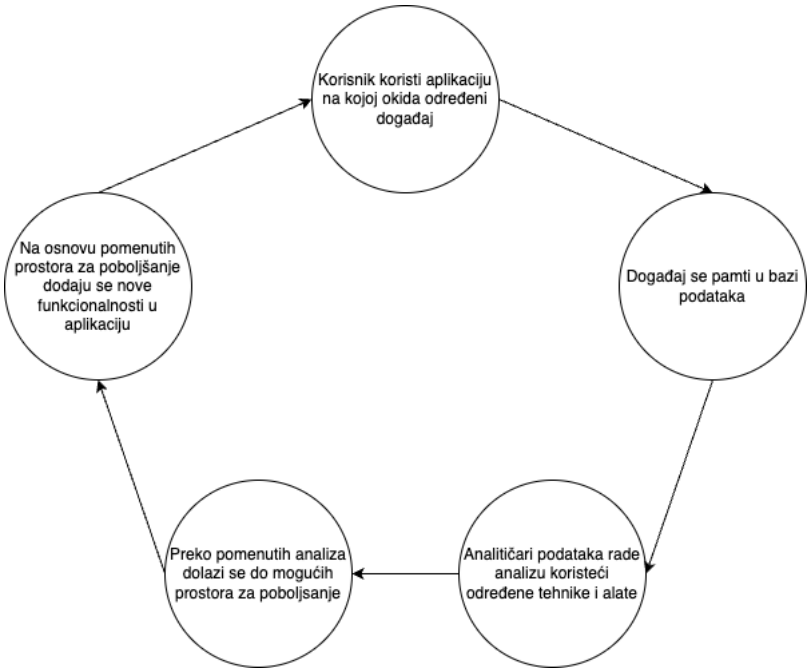
U savremenom poslovanju, praćenje poslovnih metrika igra ključnu ulogu u donošenju informisanih odluka i optimizaciji poslovnih odluka. Poslovne metrike omogućavaju kompanijama da prate performanse, identifikuju trendove i prepoznaju oblasti koje zahtevaju poboljšanja. U ovom poglavlju, fokus će biti stavljen na značaj poslovnih metrika i biće predstavljen predlog za implementaciju sistema za praćenje događaja (eng, events) unutar React Native aplikacije.

Na primeru implementacije jednog ovakvog sistema rezmatraće se najbolje prakse za prikupljanje podataka u realnom vremenu, kako bi se obezbedilo tačno i pravovremeno praćenje ključnih indikatora performansi. Takođe, detaljno će biti opisani koraci za implementaciju skalabilnog i fleksibilnog sistema koji može da se prilagodi specifičnim potrebama kompanije, omogućavajući menadžmentu da donosi odluke zasnovane na preciznim i relevantnim informacijama. Na kraju, biće razmatrani načini za vizualizaciju i interpretaciju prikupljenih podataka, što će pomoći kompanijama da steknu dublji uvid u svoje operacije i značajno poboljšaju svoje performanse.

## 6.1 Značaj poslovnih metrika

Često, prilikom izrade softvera kraj se proglašava onog momenta kada su sve funkcionalnosti implementirane. Ovakav pristup je po pravilu pogrešan. Greška se ogleda u činjenici da ne postoji sistem koji nije moguće unaprediti. Takođe, često je veoma teško uočiti prostor za poboljšanje ukoliko nemamo mehanizme koji nam to omogućavaju. Mehanizam kome se najčešće pribegava jeste čuvanje svih relevantnih

događaja u bazama podataka [13]. Ovi događaji se kasnije obrađuju i iz tog oblika se izvlače poslovni zaključci koji će kasnije voditi ka definisanju prostora za poboljšanje i novim funkcionalnostima u sistemu.




Slika 6.1: Ciklus unapređivanja aplikacije

Na slici 6.1 vidi se ciklus unapređivanje proizvoda. Na samom početku postoji proizvod koji radi i koji korisnik upotrebljava. Taj korisnik tokom korišćenja proizvoda okida neki događaj i ta informacija se smešta u bazi podataka. Informacije iz baze koriste analitičari podataka kako bi došli do zaključaka preko kojih se definišu delovi proizvoda koji se mogu unaprediti. Preko ovih delova proizvoda i analiziranih podataka definišu se konkretne funkcionalnosti koje inženjer implementira. U ovom momentu ponovo kreće ciklus.

Da bi bilo u moguće sprovesti sve ove korake u delo od esencijalnog značaja je da postoji sistem u okviru mobilne aplikacije koji će omogućiti prikupljanje i čuvanje korisničkih interakcija na način pogodan za kasniju obradu. Jedan takav sistem biće opisan u narednoj sekciji.

## 6.2 Arhitektura sistema za prećenje događaja

Na samom početku, pre pravljenja jednog ovakvog sistema neophodno je definisati koje probleme on treba da reši. Ovakav pristup je često veoma koristan jer

podstiče na izlazak iz okvira implementacije i zahteva fokus na ono što je potrebno da se napravi, odnosno na problem koji je potrebno rešiti.

Osnovne funkcionalnosti sistema za praćenje događaja su:

- Lako dodavanje novih događaja
- Mogućnost istovremenog upisivanja događaja na različite servise za čuvanje podataka
- Lak način za unapređivanje događaja
- Lako testiranje sistema pre puštanja na produkciju
- Mogućnost da se na isti događaj reaguje na različite načine

U nastavku će biti predstavljena arhitektura sistema koji zadovoljava sve gorepomenute kriterijume.

## Osnovni entiteti

Prilikom razvijanja sistema za praćenje događaja uočavaju se dva osnovna entiteta. U pitanju su događaj i pratilac događaja. Događaj je osnovni gradivni entitet na kom će se zasnivati sve dalje što se bude razvijalo. Njegova implementacija je prikazana na slici 6.2 i on je implementiran kao interfejs koji mora imati polje eventName koje predstavlja ime tog događaja. Pored imena, svaki pojedinačni događaj može biti obogaćen dodatnim poljima koja bliže određuju isti. Ovo je omogućeno tako što je sam događaj ništa više nego TypeScript objekat. Dakle, događaj je skup polja koja imaju neke vrednosti.

```
export type AnalyticsEventWildcardType = '*';

You, 2 months ago | 1 author (You)
export interface AnalyticsEventInterface<
  EventName = AnalyticsEventWildcardType,
> {
  eventName: EventName;
}
```

Slika 6.2: Interfejs za događaj

Drugi bitan entitet je pratilac događaja i on je prikazan na slici 6.3. Pratilac događaja predstavlja klasu koja u sebi sadrži sve događaje na koje isti reaguje. Takođe, za svaki od tih događaja vezana je i funkcija koja govori kako će ovaj pratilac

reagovati na taj specifičan događaj. Ovakvih pratilaca može u sistemu biti neograničeno i svaki od njih može da reaguje na proizvoljan broj događaja na proizvoljan način. Na primer, može postojati jedan pratilac koji će biti korišćen za testiranje sistema i on će reagovati na sve događaja tako što će da ih ispisuje u konzolu. U isto vreme, može postojati drugi pratilac koji će takođe reagovati na sve događaje, međutim, on će ih prosleđivati u bazu podataka.

```
You, 2 months ago | 1 author (You)
export class AnalyticsTracker {
  getEventNames = (): AnalyticsEventName[] =>
    Array.from(this.eventProcessors.keys());

  isListeningTo = (eventName: AnalyticsEventName): boolean =>
    this.eventProcessors.has(eventName) ||
    this.eventProcessors.has(AnalyticsEventWildcard);

  private eventProcessors: LogFuncMap;

  name: string;

  private initProcessorsMap = (processors: LogFuncMap): LogFuncMap => {
    if (!processors.has(AnalyticsEventWildcard)) {
      return processors;
    }

    const logDefault = processors.get(AnalyticsEventWildcard)!;
    processors.delete(AnalyticsEventWildcard);
    const eventNames = getAllEventsExcept(Array.from(processors.keys()));
    eventNames.forEach(eventName => {
      processors.set(eventName, logDefault);
    });

    return processors;
  };

  init: AnalyticsTrackerInitFunc;

  constructor(config: AnalyticsTrackerConfig) {
    this.name = config.name;
    this.eventProcessors = this.initProcessorsMap(config.processors);
    this.init = config.init;
  }

  log: AnalyticsTrackerLogFunc = event => {
    if (this.isListeningTo(event.eventName)) {
      this.eventProcessors.get(event.eventName)!(event);
    }
  };
}
```

Slika 6.3: Klasa za pratilac događaja

Nakon što su definisana ova dva entiteta potrebno je nešto što će da ih poveže

i zapravo obezbedi željenu funkcionalnost. U konkretnom slučaju to će da bude AnalyticsEventBus klasa koja je prikazana na slici 6.4. Ona će da bude realizovana kao obrazac Unikat [10] (eng. Singleton). Dakle, kreiraće se samo jedna instanca ove klase i biće korišćena kroz celu aplikaciju. Ova klasa posedovaće niz svih pratilaca događaja koji su kreirani u okviru aplikacije. Takođe, omogućavaće dodavanje novih pratilaca. Glavna funkcionalnost ove klase biće funkcija za obradu nekog događaja. Kada se okine događaj, ova klasa će se pobrinuti da svaki pratilac koji treba da reaguje na taj događaj to zapravo i uradi. Kako sve to radi biće predstavljeno malo kasnije.

Na slici 6.5 je prikazan UML dijagram koji definiše odnose između kreiranih klasa i interfejsa.

## Primer korišćenja i funkcionisanja

U ovoj sekciji biće prikazan način korišćenja prethodno kreiranog sistema. Takođe, biće detaljno objašnjeno kako ceo proces funkcioniše.

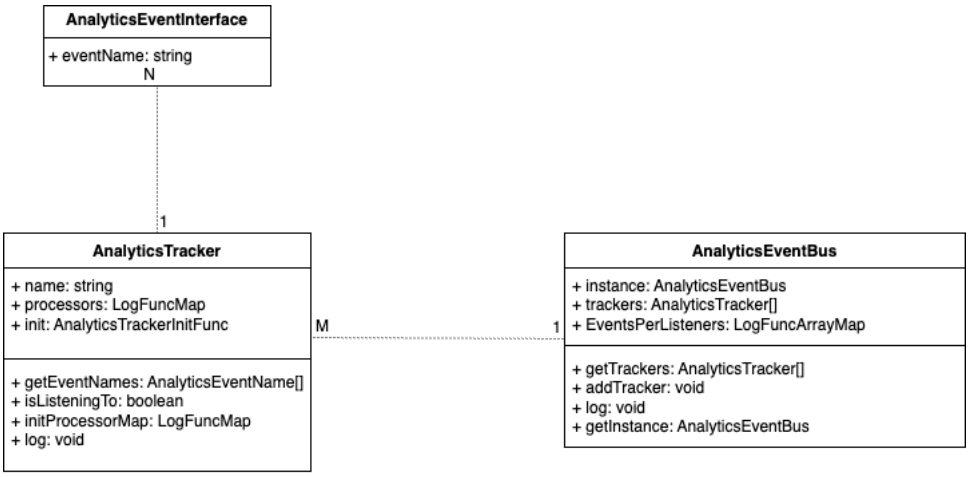
Na slici 6.6 se vidi kako se reaguje na jedan događaj. Na početku dohvatamo instancu unikat klase analyticsEventBus. Nad ovom instancom klase poziva se *log* metoda kojoj se prosleđuje jedan konkretan događaj na koji je potrebno da se reaguje. Vidimo da događaj, pored obaveznog polja *eventName* ima i polje *type* koje ga bliže određuje.

U ovom momentu, kreće da se izvršava *log* metoda klase AnalyticsEventBus. Ranije je pomenuto da svaki od pratilaca događaja, ukoliko reaguje na neki događaj mora da poseduje funkciju koja se poziva kada se okine taj događaj. U *log* metodi dohvataju se sve funkcije koje treba da se pozovu za taj događaj. Dakle, prolazi se kroz sve pratioce događaja i ukoliko oni reaguju na dati događaj dodaje se funkcija u niz. Nakon ovoga se prolazi kroz dobijeni niz i poziva se svaka funkcija iz istog pri čemu je parametar uvek isti, naš okinuti događaj. U konkretnoj implementaciji ovaj korak je optimizovan i ovaj niz se ne računa svaki put kada se događaj okine već samo prilikom dodavanja novog pratioca događaja. Kada se doda novi pratioc događaja, prolazi se kroz sve događaje na koje on reaguje i radi se mapiranje istuh u objekat koji je zajednički za sve pratioce. U ovom objektu ključevi su imena događaja, a vrednost su sve funkcije koje treba pozvati prilikom okidanja konkretnog događaja.

```
You, 2 months ago | 1 author (You)
export class AnalyticsEventBus {
  private static instance: AnalyticsEventBus | null = null;

  private trackers: AnalyticsTracker[] = [];

  private EventsPerListeners: LogFuncArrayMap = new LogFuncArrayMap();

  getTrackers = (): AnalyticsTracker[] => this.trackers;

  addTracker = (
    tracker: AnalyticsTracker,
    environment: 'debug' | 'release',
  ): void => {
    if (tracker.init) {
      tracker.init(environment);
    }
    this.trackers.push(tracker);
    const eventNames = tracker.getEventNames();

    eventNames.forEach(event => {
      if (!this.EventsPerListeners.get(event)) {
        this.EventsPerListeners.set(event, []);
      }

      this.EventsPerListeners.get(event)!.push(tracker.log);
    });
  };

  log: AnalyticsTrackerLogFunc = event => {
    this.EventsPerListeners.get(event.eventName)?.forEach(listenerLog =>
      listenerLog(event),
    );
  };

  public static getInstance = () => {
    if (!AnalyticsEventBus.instance) {
      AnalyticsEventBus.instance = new AnalyticsEventBus();
    }
    return AnalyticsEventBus.instance;
  };
}

const analyticsEventBus = AnalyticsEventBus.getInstance();

export default analyticsEventBus;
```

Slika 6.4: Glavna klasa za sistem za praćenje događaja

Prethodno opisanom optimizacijom se značajno štede performanse. Pretpostavimo da je broj događaja označen sa M, broj pratioca sa N i recimo da je okinuto P događaja. Ukoliko bi bio korišćen pristup koji je prvi opisan, prilikom okidanja P događaja svaki put bi preračunavali niz funkcija koje treba da se pozovu. Jedno računanje je složenosti O(N*M) jer se svaki put prolazi kroz N pratioca i za svakog pratioca prolazi se kroz M događaja. Dakle, ukupna složenost kreiranja bi bila O(P*N*M) [9]. Dok bi u drugom pristupu složenost bila O(N*M) jer se računanje




Slika 6.5: UML dijagram

```
const renderOptionButtons = () => (
  <Row style={{marginVertical: 8, justifyContent: 'space-between'}}>
    <Spacing size={1} right>
      <Button
        color={      You, last month • [WIP] Change design and add events
          currentOption === GRAPH_TYPE.TEMPERATURE
            ? colors.primary
            : colors.white
        }
        title="Temperature"
        onPress={() => {
          setCurrentOption(GRAPH_TYPE.TEMPERATURE);
          analyticsEventBus.log({
            eventName: CLICKED_GRAPH_TYPE_EVENT,
            type: 'Temperature',
          });
        }}
      />
    </Spacing>
    <Spacing size={1} right>
      <Button
        color={
          currentOption === GRAPH_TYPE.WIND ? colors.primary : colors.white
        }
        title="Wind"
        onPress={() => {
          setCurrentOption(GRAPH_TYPE.WIND);
          analyticsEventBus.log({
            eventName: CLICKED_GRAPH_TYPE_EVENT,
            type: 'Wind',
          });
        }}
      />
    </Spacing>
  </Row>
);
```

Slika 6.6: Primer korišćenja događaja

radi samo jednom, prilikom dodavanja pratioca.

# Glava 7

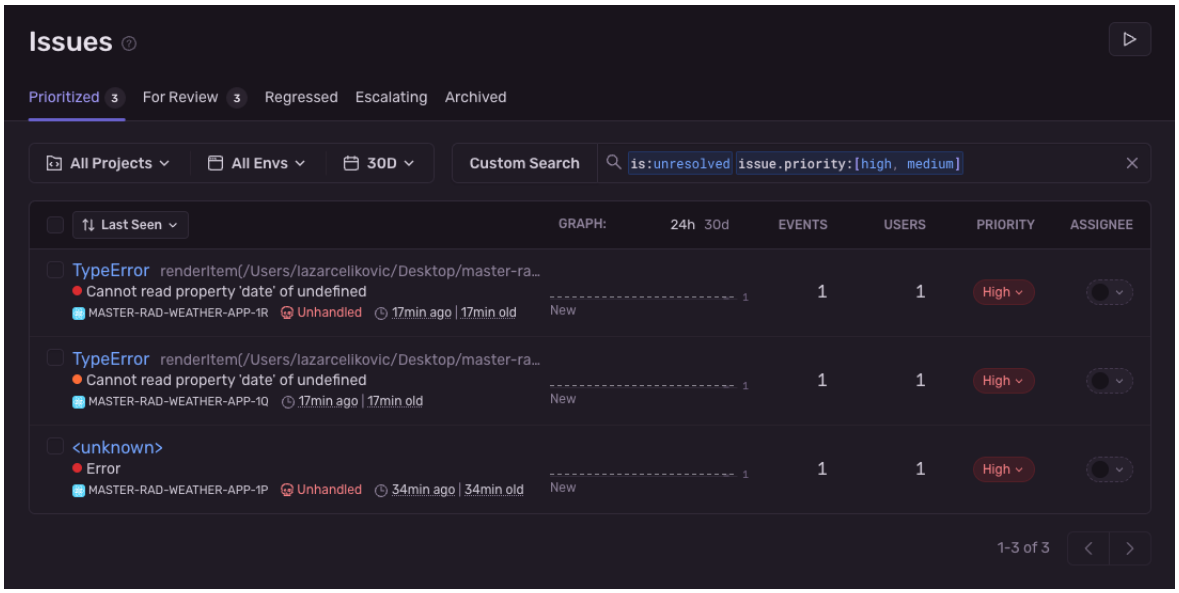
# Sistem za praćenje grešaka unutar aplikacije

Pored prethodno pomenutih sistema za merenje performansi i poslovnih metrika aplikacije, od esencijalnog značaja je da se postavi sistem za praćenje grešaka. Realnost je da većina aplikacija ima greške, odnosno da funkcionisanje odstupa od specifikacije. Veoma je bitno da postoji mogućnost da se te greške uoče na sistematičan način i da se adresiraju kako bi ih razvojni tim u budućnosti otklonio i obezbedio fluidno iskustvo za korisnike. Ovaj problem je veoma izazovan jer su aplikacije često veoma velike i većina slučaja upotrebe nije trivijalna i potrebno je da se ispuni priličan broj uslova da bi se reprodukovale. Sve prethodno pomenuto testiranje i pronalaženje ovih problema čini težim. U svrhe olakšavanja pronalaženja problema unutar aplikacije biće odrađena integraciju sistema Sentry. Ovaj sistem je već korišćen u poglavlju o performansama. Tada je odrađena integracija koja je dozvolila da se prate performansne metrike. Sada će ta ista integracija omogućiti da se beleže sve greške koje su se dogodile u mobilnoj aplikaciji uz propratne informacije koje će omogućiti da odredimo u kom delu sistema se problem dešava.

## Način funkcionisanja sistema Sentry u svrhe praćenja grešaka

Kao što je i pre rečeno, biblioteka Sentry integriše se u React Native mobilnoj aplikacije tako što se cela aplikacija obmotava samim SDK-om (eng. Software development kit). U ovom master radu to je urađeno u App.tsx fajlu. Dakle, kada se desi neka greška unutar aplikacije ona će biti propagirana do samog vrha. Biblioteka Sentry će registrovati grešku, pridružiće joj što je moguće više parametara kako bi

razvojni tim mogao da zaključi šta se zapravo desilo što brže i efikasnije. Nakon što je ovo odrađeno, ta greša se šalje na Sentry plaformu gde će biti zapravo analizirana.




Slika 7.1: Primer liste grešaka na plaformi Sentry

Na slici 7.1 vidi se glavna stranica platofrme Sentry. Na istoj su izlistane sve greške koje su se desile unutar aplikacije. U nastavku će biti analizirane glavne funkcionalnosti.

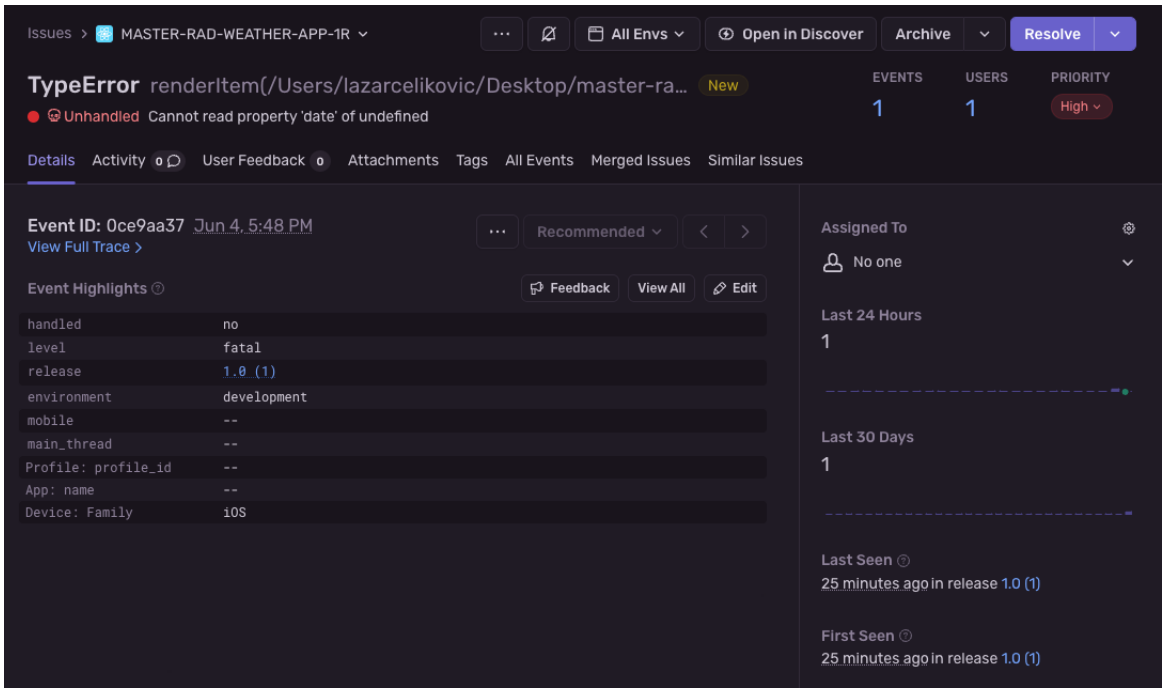
## 7.1 Funkcionalnosti platforme Sentry

Kao što je prikazano na slici 7.1, platforma Sentry pruža ogromnu količinu podataka o svakoj pojedinačnoj grešci koje je logovana na istoj. Na glavnoj stranici su izlistane sve greške koje su se desile unutar aplikacije i date su najosnovnije informacije o svakoj pojedinačnoj grešci. Ove informacije uključuju momenat dešavanja, naziv i opis greške, broj pojavljivanja greške i broj korisnika koji su je iskusili. Takođe, prikazan je prioritet greške kao i osoba kojoj je dodeljeno rešavanje. Ovo polje je posebno interesantno jer sama plaforma omogućava dodatnu integraciju sa servisom GitHub. Uz pomoć ovog servisa ovo polje će automatksi biti popunjeno, odnosno rešavanje problema biće dodeljeno onom članu razvojnog tima koji je poslednji vršio izmenu linije koda u kojoj se desila greška.

Zanimljiviji i svakako korisniji deo ove plaforme jeste sama stranica o grešci. Klikom na svaku pojedinačnu grešku korisnik će biti navigiran na posebnu stranicu gde će dobiti sve dostupne informacije o istoj kako bi mogao da je reši. Ova stranica je veoma obimna i pruža ogromnu količinu informacija, tako da će analiza biti odrađena kroz manje celine.

## Najosnovnije informacije o grešci

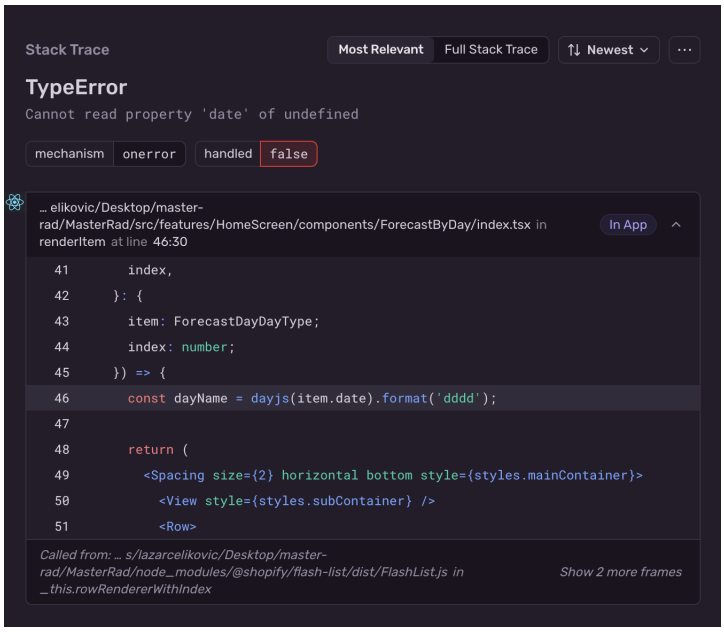
Prva sekcija na stranici daje najosnivnije informacije o grešci koja se analizira. Tu se može videti sam tekst greške, fajl u kome se greška dogodila, broj pojavljivanja greške kao i broj korisnika kod kojih se greška manifestovala. Takođe, vidi se broj pojavljivanja greške kako u poslednja 24 časa, tako i u poslednjih mesec dana, kao i podatke o verziji aplikacije u kojoj je prvi i poslednji put uočena greška. Pomoću ovih informacija može se odrediti interni prioritet greške i na taj način odlučiti da li će ista biti prioritizovana ili ne. Ove informacije prikazane su na slici 7.2.




Slika 7.2: Osnovni podaci o grešci

Nakon ovih informacija, uočava se deo koda u kom se greška dogodila. Ovo je veoma korisna informacija jer sužava pretragu i daje ključne informacije kako bi razvojni tim mogao da krene sa procesom rešavanja. Ovde takođe postoji opcija da se vidi cela putanja greške (eng. stacktrace). Ova opcija je naročito korisna kada je

u pitanju greška unutar nekog eksternog sistema ili biblioteke. U tom slučaju, sama greška se manifestuje u aplikaciji, ali je rešavanje potrebno sprovesti van iste. Primer jedne ovakve putanje vidimo na slici 7.3.




Slika 7.3: Informacije o delu koda gde se greška javlja

## Koncept mrvica hleba

Mrvice hleba [1] (eng. breadcrumbs) unurar sistema Sentry predstavljaju sekvencu događaja ili akcija koji su se dogodili neposredno pre greške unutar aplikacije. Koncept je potekao od ostavljanja mrvica hleba koje će pomoći da se dođe do tačke kada je nešto krenulo po zlu. Ove mrvice mogu biti prikupljane automatski, ali takođe korisnik može dodati lične. Pomoću njih olakšan je posao razvojnom timu prilikom reprodukcije i rešavanja greške. Dakle, u ovoj sekciji može se pratiti tačna interakcija korisnika sa aplikacijom pre nego što je došlo do greške. Platforma Sentry prikazuje API pozive koji su pokrenuti, interakciju sa elementima korisničkog interfejsa (klik na dugme, selektovanje radio dugmeta, popunjavanje tekstualnog polja, ...) i ovo se vidi na slici 7.4. Ove informacije su od ključnog značaja kako bi razvojni tim mogao da razume slučaj upotrebe kao i tok akcija koje dovode do nekog problema.

Slika 7.4: Breadcrumbs

Često, greške koje se desavaju unutar sistema se ne nalaze u glavnim tokovima korišćenja (eng. flows), već u nekim specifičnim slučajevima upotrebe. Na primer, korisnik je kliknuo na push notifikaciju ili deeplink [3, 133 - 139] i preko neke od ovih funkcionalnosti bio preusmeren na određenu stranicu i tom prilikom dešava se greška na stranici na koju je preusmeren. Kada razvojni tim pokuša da reprodukuje ovo ponašanje greška se ne desi. Ovo je veoma čest slučaj i poenta je u tome da u velikim sistemima postoji mnogo načina da se dodje do istog dela sistema i praktično je nemoguće testirati ih sve. U ovim situacijama može biti korisno da razvojni tim doda lične mrvice hleba kako bi olakšao određivanje tačne putanje kojom je korisnik došao do greške. Sentry plaforma nam pruža ovu mogućnost i to vidimo na slici 7.5.

```
Sentry.addBreadcrumb({
  category: "auth",
  message: "Authenticated user " + user.email,
  level: "info",
});
```

Slika 7.5: Dodavanje novog breadcrumb-a

### Informacije o uređaju

Neretko, same greške imaju povezanost sa trenutnim stanjem uređaja. Na primer, ukoliko je internet konekcija slaba može doći do povećanog vremena čekanja za informacije sa servera ili čak do situacija kada te informacije uopšte ne stignu nazad. Takođe, ukoliko je nivo baterije pri kraju to takođe može da izazove razne probleme u funkcionisanju aplikacije. Platforma Sentry pruža i ove informacije kao što se vidi na slici 7.6.

Slika 7.6: Informacije o uređaju

## 7.2 Poboljšana arhitektura za praćenje grešaka

Sada, kada je prikazano šta sve plaforma Sentry nudi i kako je moguće da se odradi osnovna inicijalizacija, vreme je da se razmisli o unapređivanju postojećeg rešenja. Za početak, potrebno je da se definiše koje sve tipove grešaka razvojni tim želi da prati. To su sve one greške do kojih dolazi do prekida rada aplikacije ili do neželjenih ponašanja u korisničkom iskustvu. Ti tipovi grešaka su:

- Neobrađene greške (eng. ungandled errors) - ove greške se dešavaju u momentu kada aplikacija naiđe na grešku i ne zna kako da je obradi. Ove greške nisu

veoma česte, ali ukoliko se dese veoma često dovode do potpunog prestanka rada aplikacije.

- JavaScript greške (eng. JS erros) - ove greške predstavljaju najveći deo svih grešaka unutar React Native aplikacija. U pitanju su stvari kao što su sintaksne greške, pristupanju poljima objekata pri čemu sam objekat ima vrednost undefined.

- Mrežne greške (eng. Network errors) - ova grupa obuhvata sve greške koje se događaju usled nemogućnosti aplikacije da komunicira sa serverom. Kao što je ranije pomenuto, to su problemi sa slabom internet konekcijom, ali i potpunim odsustvom mreže.

Kada je korak određivanje greški koje je potrebno pratiti uspešno realizovan, može se preći na pravljenje omotača oko React Native aplikacije koji će služiti da obrađuje sve greške koje se dese unutar aplikacije na način koji razvojni tim definiše. Ovo će biti realizovano preko ErrorBoundary komponente. ErrorBoundary je jedan od standardnih obrazaca kod React i React Native aplikacija. U pitanju je komponenta koje služi kao omotač oko cele aplikacije. Kada se greška dogodi, biva propagirana kroz od dna do vrha stabla komponenti i na kraju biva uhvaćena u okviru ovog omotača. Tu se ista obrađuje i normalizuje i nakon toga se šalje na Sentry plaformu.

Sentry pruža ugrađenu podršku za ErrorBoundary. Na slici 7.7 ćemo videti kako bi izgledala implementacija.

```
function App(): JSX.Element {
  return (
    <Provider store={store}>      You, 3 months ago • [WIP] Add redux for forecast
      <Sentry.ErrorBoundary
        fallback={<Text variation="body1">An error has occurred</Text>}>
        <SafeAreaView style={{flex: 1}}>
          <NavigationComponent
            routingInstrumentation={routingInstrumentation}
          />
        </SafeAreaView>
      </Sentry.ErrorBoundary>
    </Provider>
  );
}

export default Sentry.wrap(App);
```

Slika 7.7: Error boudary komponenta

Naravno, moguće je napisati i sopstvenu komponentu koja će imati istu funkciju. U većini realnih aplikacija koje imaju veliki broj korisnika i funkcionalnosti to je i poželjno jer takav pristup daje potpunu kontrolu nad obradom grešaka. U ovom konkretnom slučaju ugrđena podrška itekako pruža željene funkcionalnosti tako da se ostaje pri tom pristupu.

# Glava 8

# Zaključak

Na samom početku rada ukratko su opisane mnogobrojne tehnologije za razvoj mobilnih aplikacija. Tu je istaknuto da se konstantno pojavljuju nove tehnologije u ovoj sferi programiranja. Takođe, primećeno je da se stvari ne menjaju preko noći i da se velika većina ovih koncepata primenjuje duže vreme.

Zatim, analiziran je radni okvir React Native. Ovde je prikazano kako ove aplikacije zaista funkcionišu u okviru samog operativnog sistema. Analiziran je koncept mosta, njegove prednosti i nedostaci. Takođe, dat je kratak pregled trenutnih optimizacija koje će u budućnosti verovatno dovesti do krupnijih arhitekturalnih izmena u okviru ovog radnog okvira.

Razumevanje kako React Native funkcioniše „ispod žita" je dalo odličnu osnovu da se u nastavku rada govori o performansama aplikacije. Čest je stereotip da mobilna aplikacija ne može imati kvalitetne performanse ukoliko nije pisana u jeziku koji razume sam operativni sistem, odnosno ukoliko nije nativna (Swift ili Objective-C za iOS i Java za Android). Jedan od ciljeva ovog rada bio je da pokaže da ovo nije slučaj i da jedan radni okvir zasnovan na JavaScript programskom jeziku itekako može da ispunjava ove kriterijume. Stoga, rad obuhvata i diskusiju i analizu mehanizama pomoću kojih se mogu pratiti same performanse aplikacije kao i neke od biblioteka koje pomažu da se to i realizuje. Istina je da su nativne aplikacije same po sebi brže, međutim uz malo truda, definitivno je moguće pisati React Native aplikacije koje im pariraju.

U nastavku je analiziran pojam analitike. Postavljen je jedan veoma funkcionalan i modularan sistem za praćenje korisničkih interakcija sa mobilnom aplikacijom. Takođe, ovde je analiziran značaj posedovanja jednog ovakvog sistema i zaključeno je da nije moguće razvijati jedan uspešan proizvod bez uvida u način na koji je isti

korišćen. Ovo daje jednu skroz novu dimenziju i mogućnost da se detaljno analizira koji delovi sistema ne ispunjavaju očekivanja korisnika i gde postoji mogućnost za poboljšanje.

Na samom kraju je uspostavljen sistem za obradu i praćenje grešaka.

Svakako, bitno je naglasiti da aplikacija koje je prikazana u okviru ovog rada ima nedostataka. Jedan od glavnih jeste nedostatak arhitekture za pisanje testova. Na primer, ukoliko bi ista postojala bilo bi moguće napisati jedinične testove (eng. unit tests) koji bi testirali manje pomoćne funkcije, ali bi bilo moguće i napisati testove koji testiraju cele ekrane ili čak celokupne korisničke putanje (eng. user journey) kroz integracione i end-to-end testove. Takođe, poboljšanje se može ostvariti i na frontend nivou. Moguće je dodatno optimizovati sam korisnički interfejs i dohvatanje podataka sa servera kako bi se broj iscrtavanje korisničkog interfejsa smanjio na minimum. Na ovaj način se dodatno mogu optimizovati performanse.

Implementacija aplikacije za vremensku prognozu predstavljena u ovom master radu, koja je dostupna javno na adresi https://github.com/Hos1g4k1/masterRad, predstavlja veoma stabilnu osnovu za dalji razvoj. Takođe, svi sistemi prikazani u radu su pisani tako da se mogu iskoristiti u drugim projektima, tako da ovaj rad predstavlja veoma dobru referencu za buduće projekte.

# Bibliografija


[1] Breadcrumbs. https://docs.sentry.io/platforms/javascript/enriching-events/breadcrumbs/, 2024.

[2] Deep dive into react native's flatlist. https://blog.logrocket.com/deep-dive-react-native-flatlist/, 2024.

[3] Paul Akshat and Nalwaya Abhishek. *React Native for Mobile Development*. 2019.

[4] Android team. Android documentation. https://source.android.com/docs, 2024.

[5] Apple team. iOS documentation. https://developer.apple.com/documentation/, 2024.

[6] Tal Ater. *Building Progressive Web Apps: Bringing the Power of Native to the Browser*. 1 edition, 2017.

[7] Bealdung. Java VS Kotlin. urlhttps://www.baeldung.com/kotlin/java-vs-kotlin, 2024.

[8] Flipper. React native flipper. https://fbflipper.com/docs/features/react-native/, 2024.

[9] Geeks for Geeks. Big o notation tutorial – a guide to big o analysis. https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/, 2024.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition, 1994.

[11] JetBrains team. Cross platform mobile applications. https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-mobile-development.html, 2024.

[12] Shaun Lewis and Mike Dunn. *Native Mobile Development*. 11 2019.

[13] Anil Maheshwari. *Data Analytics Made Accessible*. Internet Archive, 2020.

[14] Miroslav Marić. *Operativni sistemi*. 2 edition.

[15] Medium. React native architecture. `https://medium.com/@sohammehta56/react-native-old-and-new-architecture-61cefd6b9912`, 2024.

[16] Talha Naqvi. Recyclerlistview component. `https://github.com/Flipkart/recyclerlistview`, 2022.

[17] Netguru. Swift vs Objective-C. `https://www.netguru.com/blog/objective-c-vs-swift`, 2024.

[18] JSON Org. JSON format. `https://www.json.org/json-en.html`, 2024.

[19] Pagepro team. Ionic vs Cordova. `https://pagepro.co/blog/react-native-vs-ionic-and-cordova-comparison/`, 2024.

[20] Sentry. Sentry performance metrics. `https://docs.sentry.io/platforms/react-native/performance/instrumentation/performance-metrics/`, 2024.

[21] Sentry. Sentry platform. `https://docs.sentry.io`, 2024.

[22] Shopify. Flashlist component. `https://shopify.github.io/flash-list/`, 2022.

[23] WeatherAPI. Weather api. `https://www.weatherapi.com`, 2024.

[24] web.dev team. Service workers. `https://web.dev/learn/pwa/service-workers`, 2024.

[25] webkit.org. JavaScriptCore engine. `https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html`, 2024.

# Biografija autora

**Lazar Čeliković** rođen je 18. septembra 1998. godine u Užicu. U ovom gradu završava osnovno i srednje obrazovanje, nakon čega 2017. godine upisuje osnovne studije na smeru Informatika na Matematičkom fakultetu u Beogradu. Osnovne studije završava 2021. godine i odmah nakon diplomiranja upisuje master studije u okviru istog studijskog programa. Pred kraj osnovnih studija zapošljava se u kompaniji FishingBooker na poziciji softverskog inženjera za mobilne aplikacije gde i danas radi.