Razvoj mobilne aplikacije za ocjenu kvalitete života u novoj sredini

Hudiček, Matej

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:296025

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-20**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 549

RAZVOJ MOBILNE APLIKACIJE ZA OCJENU KVALITETE ŽIVOTA U NOVOJ SREDINI

Matej Hudiček

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 549

RAZVOJ MOBILNE APLIKACIJE ZA OCJENU KVALITETE ŽIVOTA U NOVOJ SREDINI

Matej Hudiček

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 549

Pristupnik: **Matej Hudiček (0036524708)**
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi
Mentor: prof. dr. sc. Krešimir Trontl

Zadatak: **Razvoj mobilne aplikacije za ocjenu kvalitete života u novoj sredini**

Opis zadatka:

Klimatske promjene i urbanizacija rezultiraju promjenama kvalitete života u gradovima. Dinamika promjene poslova u modernom društvu dovodi pak do promjene radnog i životnog okruženja, pri čemu je jedan od bitnih faktora u donošenju odluke o preseljenju u novi grad i ocjena kvalitete života u novoj sredini. Stoga je planiran razvoj CroQoL mrežne aplikacije koja će korisniku omogućiti procjenu kvalitete života u hrvatskim gradovima. Cilj rada je razvoj mobilne aplikacije koja upotrebom dostupnih API funkcija omogućava procjenu kvalitete života u hrvatskim gradovima. Također je uz odgovarajuću autentifikaciju potrebno omogućiti ažuriranje korištenih baza podataka aktualnim informacijama s lokacije mobilnog uređaja.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

Uvod 2

1. Mobilne aplikacije 3

2. Razvoj Android mobilne aplikacije 5

   2.1. Android Studio 5

   2.2. Kotlin 5

   2.3. Jetpack Compose 7

      2.3.1. Aktivnosti i njihov životni ciklus 9

      2.3.2. Material Design 11

   2.4. Arhitektura mobilne aplikacije 11

      2.4.1. Sloj korisničkog sučelja 12

      2.4.2. Sloj pristupa podacima 14

      2.4.3. Prosljeđivanje ovisnosti 17

3. CroQoL aplikacija 21

   3.1. Autentifikacija i autorizacija 21

   3.2. API za procjenu kvalitete života 24

   3.3. Google Maps 27

   3.4. Početni zaslon 28

      3.4.1. Podaci i funkcionalnost početnog zaslona 34

   3.5. Anketa 38

      3.5.1. Podaci i funkcionalnost ankete 40

Zaključak 44

Literatura 45

Sažetak 47

Summary 48

Uvod

Klimatske promjene i urbanizacija sve više utječu na kvalitetu života u gradovima širom svijeta. Kako se gradovi razvijaju i mijenjaju, tako se mijenjaju i uvjeti života unutar njih, što može imati značajan utjecaj na svakodnevni život stanovnika. S druge strane, ubrzana dinamika promjene poslova u modernom društvu često dovodi do toga da ljudi mijenjaju radno i životno okruženje, preseljavajući se u nove gradove. U tom kontekstu, ocjena kvalitete života postaje ključni faktor pri donošenju odluke o preseljenju.

Razvoj mobilnih tehnologija i sve veća upotreba pametnih telefona omogućili su stvaranje alata koji mogu pomoći korisnicima u procjeni kvalitete života u različitim urbanim sredinama. Jedan od takvih alata je i planirana mobilna aplikacija CroQoL.

Cilj ovog rada je razvoj mobilne aplikacije koja će koristiti dostupne API funkcije za procjenu kvalitete života u hrvatskim gradovima. Aplikacija će omogućiti korisnicima da, putem svojih mobilnih uređaja, odaberu željeni grad i za njega dobiju procijenjene uvjete kvalitete života.

Kroz ovaj rad, istražit će se proces razvoja Android mobilne aplikacije koristeći najnovije alate i tehnologije, uključujući Android Studio, Kotlin, te Jetpack Compose. Posebna pažnja bit će posvećena dizajnu korisničkog sučelja i funkcionalnosti aplikacije, autentifikaciji korisnika, te integraciji s relevantnim API-jevima, kao što je Google Maps. Uz to, rad će obuhvatiti i analizu arhitekture mobilne aplikacije, uključujući slojeve korisničkog sučelja, pristupa podacima, te prosljeđivanja ovisnosti.

Na kraju, bit će prikazana konkretna implementacija CroQoL aplikacije, s fokusom na funkcionalnosti poput početnog zaslona, prikupljanja i obrade podataka, te pružanja korisnicima mogućnost ispunjavanja anketa koje će dodatno doprinijeti ocjeni kvalitete života u novoj sredini.

1. Mobilne aplikacije

U današnje doba interneta i pametnih telefona takve su i najpopularnije aplikacije, web i mobilne. Prednost aplikacija weba nad stolnim i mobilnim aplikacijama je što ih korisnik ne mora instalirati na uređaj već im može lako pristupiti s pomoću web preglednika, troše malo resursa uređaja, ne treba ih korisnik sam ažurirati i najvažnije je što su više platformske. Nebitno o softveru i hardveru uređaja, ni je stolno ili prijenosno računalo ili mobilni uređaj, ako ima web preglednik može otvoriti aplikaciju weba. S druge strane mobilne aplikacije rade brže i elegantnije zato što se izvode izravno na uređaju, moguća je puno veća personalizacija sadržaja i mogućnosti aplikacije mogu biti veće zbog pristupa lokaciji i drugih funkcionalnosti koje pruža mobilni uređaj. Mobilne aplikacije također ne trebaju internet da bi radile i mogu slati „push" obavijesti korisnicima čak i dok oni ne koriste aplikaciju što omogućuje slanje novih ponuda, popusta ili događaja i promovirati aplikaciju što je velika prednost iz poslovne strane.

U sklopu diplomskog rada se razvija mobilna aplikacija koja će iskoristiti njihove navedene mogućnosti. Prvi odabir kod razvoja mobilne aplikacije je na kojem će operacijskom sustavu raditi, iOS ili Android. Programski jezik preporučen za razvoj aplikacija na iOS-u je Swift dok je na Androidu to Kotlin. Oba jezika su moderna, jako dobro dokumentirana, imaju čistu sintaksu i podržavaju različite programske paradigme. Razvojna okolina za razvoj aplikacija na Androidu je Android Studio, a za iOS je XCode. Oba programa nude skoro identične mogućnosti za razvoj na prikladnim platformama. Razlika je također u principima dizajna korisničkog sučelja. iOS dizajn promovira ideje minimalizma i jednostavnosti s intuitivnom raspodjelom elemenata po ekranu, jednostavnim oblicima i bojama. Android aplikacije su jako fleksibilne i prilagodljive svojim dizajnom što omogućuje i kompleksnije rasporede elemenata po ekranu. Takva fleksibilnost je omogućila korištenje Androida na raznim uređajima kao što su pametni satovi, televizori, tableti i mobilni uređaji raznih dimenzija i gustoća ekrana zbog čega se razvoj može zakomplicirati. Na iOS-u takvog prolema nema zato što je Appleov ekosistem zatvoren i pod njihovom kontrolom.

Aplikacija je prvenstveno namijenjena hrvatskim korisnicima, a velika većina korisnika ima mobilne uređaje koji koriste Android operacijski sustav[1]. Iz tog razloga će aplikacija biti razvijena za Android operacijski sustav.

2. Razvoj Android mobilne aplikacije

Android je mobilni operacijski sustav temeljen na modificiranoj verziji jezgre Linuxa i drugog softvera otvorenog kôda, dizajniran prvenstveno za uređaje s ekranom na dodir. Najpopularniju verziju Androida razvija i održava Google te se ona isporučuje s unaprijed instaliranim dodatnim vlasničkim softverom Google Mobile Services (GMS) koji uključuje aplikacije kao što su Google Chrome, YouTube, Google Maps i tako dalje.

2.1. Android Studio

Android Studio je službena razvojna okolina za razvoj Android aplikacija. Temeljena je na moćnom uređivaču programskog kôda i razvojnim alatima IntelliJ IDEA i pruža još dodatne mogućnosti za što veću produktivnost[2]. Neke od tih mogućnosti su pokretanje razvijane aplikacije na mobilnom uređaju povezanome na Android Studio preko kabla na računalo ili preko WiFi mreže za direktno korištenje aplikacije na uređaju. Također je moguće koristiti emulatore kako bi isprobali aplikaciju na raznim vrstama Android uređaja bez potrebe fizičkog uređaja. Moguće je emulirati mobitele raznih dimenzija, tablete, pametne satove i televizore te simulirati njihove mogućnosti kao primanje poziva i SMS poruka, pristup lokaciji uređaja i izmjena svijetlog i tamnog načina rada[3]. Android Studio pruža i grafičko sučelje za razvoj korisničkog sučelja. Korisnik može mišem kontrolirati položaj komponenti na ekranu te im mijenjati atribute i dizajn. Pregled izgleda ekrana se mijenja uživo kako korisnik mijenja komponente[4].

IntelliJov uređivač kôda indeksira sve preuzete biblioteke te pruža automatsko dovršavanje kôda, refaktoriranje datoteka i kôda, uočava greške, ima mogućnost spajanja na bazu podataka i ima integriran sistem za upravljanjem verzija datoteka Git.

2.2. Kotlin

Kotlin je programski jezik koji Google preporučuje za razvoj aplikacija za Android. Iz tog razloga Android Studio podržava Kotlin bez potrebe dodatnih koraka. Moderan je jezik prvi put izdan 2011. godine. Razvijen je da bude potpuno kompatibilan s Javom. Kotlin programski kôd može pozivati Java kôd i obrnuto. Pokreće ga Java Virtual Machine

(JVM) i podržava Javine biblioteke i alate. Jedna prednost Kotlina je što je `null` siguran. Većina pogrešaka do kojih dolazi u Java kôdu nastanu kod pristupa metodama ili poljima objekta koji je `null`, odnosno ne referencira konkretnu instancu tog objekta[5]. Kotlin ima `null` sigurnost, odnosno do greške zato što je neki objekt `null` ne može doći implicitno. Kotlinov jezični prevoditelj ne dopušta varijable da imaju `null` vrijednost kod vremena prevođenja.

Kotlin također ima posebnu vrstu klasa, `data` klase. To je vrsta klase namijenjena isključivo čuvanju podataka. U Javi za implementacije takvih klasa potrebno je uvijek pratiti isti predložak metoda koje treba definirati dok Kotlin sam to radi. Potrebno je jedino definirati koje varijable će klasa imati.

```
data class User(val name: String, val age: Int)
```

Za primjer `data` klase `User` Kotlin će sam stvoriti metode za dohvaćanje i postavljanje varijabli `name` i `age`, te će stvoriti metode za usporedbu objekta po njima i za stvaranje kopije objekta[6]. Važna razlika je i u samoj sintaksi jezika. Kotlin ima sažetiji kôd od Jave kojim je moguće napisati iste funkcionalnosti s manje kôda.

```
//Java
public class User {
    private String name;
    public User(String name) {
        this.name = name;
    }
    public int getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
//Kotlin
class User(private var name: String) {
    fun getName() = name
    fun setName(value: String) { name = value}
}
```

Kôd 2.1 Sintaksa Jave i Kotlina

Na isječku kôda 2.1 se vidi da je Kotlin kôd puno sažetiji i čitljiviji. To omogućava definiranje atributa klase u konstruktoru i definiranje povratnu vrijednost funkcije kao atribut klase. Kotlin također podržava sve koncepte funkcijsko programiranja dok Java podržava samo neke. U Kotlinu se funkcije mogu spremati u varijable i strukture podataka, predati kao argumenti funkcije i biti povratna vrijednost drugih funkcija[7].

```
//Funkcija kao varijabla
val operation = fun(one: String, two: String): String {
    return one.plus(two)
}

//Funkcija višeg reda
fun operationOnTwoStrings(one: String, two: String,  operation: (String,
String) -> String): String {
    return operation(one, two)
}
operationOnTwoStrings("One", "Two", operation)
```

Kôd 2.2 Primjeri funkcijskog programiranja u Kotlinu

U primjeru kôda 2.2 se prvo vidi spremanje definicije funkcije u varijablu `operation`. Zatim funkciju višeg reda `operationOnTwoStrings`. Funkcije višeg reda su one koje ili primaju kao argumente druge funkcije ili im je povratna vrijednost nova funkcija. U primjeru funkcija `operationOnTwoStrings` prima kao argument funkciju koja prima dva tipa `String` i vraća tip `String` zatim u definiciji vraća rezultat te funkcije.

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]
    this[index1] = this[index2]
    this[index2] = tmp
}
mutableListOf(2, 4, 5, 6).swap(0, 3) // 6, 4, 5, 2
```

Kôd 2.3 Primjer funkcije proširenja

Funkcije proširenja u Kotlinu pružaju mogućnost dodavanja novih mogućnosti na klase i sučelja bez potrebe nasljeđivanja tih klasa ili upotrebe oblikovnih obrazaca. Posebno su korisne kad se žele dodati funkcionalnost za klase ili sučelja kojima netko nije vlasnik[8]. U primjeru kôda 2.3 definirana je funkcija proširenja `swap` nad listom brojeva koja prima dva indeksa te u listi zamijeni vrijednosti na tim indeksima. Ključna riječ `this` u definiciji funkcije proširenja odnosi se na sam objekt koji poziva funkciju

2.3. Jetpack Compose

Jetpack Compose je Androidov preporučeni moderan skup alata i biblioteka za gradnju korisničkih sučelja koji pojednostavljuje i ubrzava njihov razvoj[9]. S Composeom korisničko sučelje se gradi s pomoću funkcija, nazvanih sastavljenih funkcija, koje primaju podatke i opisuju elemente korisničkog sučelja, a ne definiraju procese same konstrukcije sučelja kao što su inicijalizacija elemenata i vezanje na roditelje. Sastavljene funkcije

stvaraju se označavanjem s `@Composable` anotacijom. Anotacije omogućavaju dodavanje mogućnosti i informacija klasama, funkcijama i atributima u kôdu.

```
@Preview
@Composable
fun Greeting(name: String) {
    Text(
        text = „Hello $name!“,
    modifier = Modifier
        .padding(8.dp)
        .clickable(onClick = onClick)
    )
}
```

Kôd 2.4 Primjer sastavljene funkcije

U kôdu 2.4 se vidi `@Compose` anotacija na funkciji Greeting koja označava da je ta funkcija sastavljena. Anotacija daje do znanja prevoditelju da je toj funkciji namjena da pretvara podatke u elemente korisničkog sučelja. U definiciji funkcije vidimo poziv druge sastavljene funkcije Text koja služi za prikaz teksta na ekranu. To je jedna od komponenti koje sadrži Jetpack Compose biblioteka za gradnju korisničkog sučelja uz druge komponente za prikaz sadržaja, definiranje raspored elemenata na ekranu i unos podataka. U kôdu je također funkcija označena `@Preview` anotacijom koja omogućava pred pregled sastavljene funkcije unutar Android Studija bez potrebe gradnje aplikacije i instalacije na uređaj ili emulator.

Elementi korisničkog sučelja su u hijerarhijskom odnosu s elementima koji sadrže druge elemente. Kao u kôdu 2.4 definirana sastavljena funkcija zove drugu. Također u pozivu funkcije se joj predaje modifikator `modifier` kojim je moguće ukrašavanje i konfiguracija ponašanja elementa. Modifikatorom je moguće mijenjati veličinu, raspored, izgled ili dodavati interakcije naprimjer prilikom pritiska na element. Mogućnosti se dodaju ulančanim pozivima[10].

Sastav je opis korisničkog sučelja koji gradi Compose prilikom izvršavanja sastavnih funkcija. Sastavne funkcije pretvaraju podatke u elemente korisničkog sučelja. Podaci koji su potrebni sastavnoj funkciji skupno se nazivaju stanje te funkcije. Ako se stanje promijeni Compose ponovno izvrši sastavljene funkcije koje sadrže to stanje s novim stanjem što stvara novo korisničko sučelje. Taj proces se zove ponovno sastavljanje i Compose se sam brine o tome kad ga je potrebno zvati[11].

Varijable se definiraju kao stanja tipovima `State` i `MutableState`. Za te tipove Compose automatski prati promjenu. `State` je nepromjenjiv pa se može samo čitati dok

je `MutableState` i promjenjiv[12]. Stanja se stvaraju prikladnim funkcijama `stateOf` i `mutableStateOf`.

```
var input = mutableStateOf(0)
TextField(
    value = input.value,
    onValueChange = { input.value = it },
    modifier = Modifier
}
```

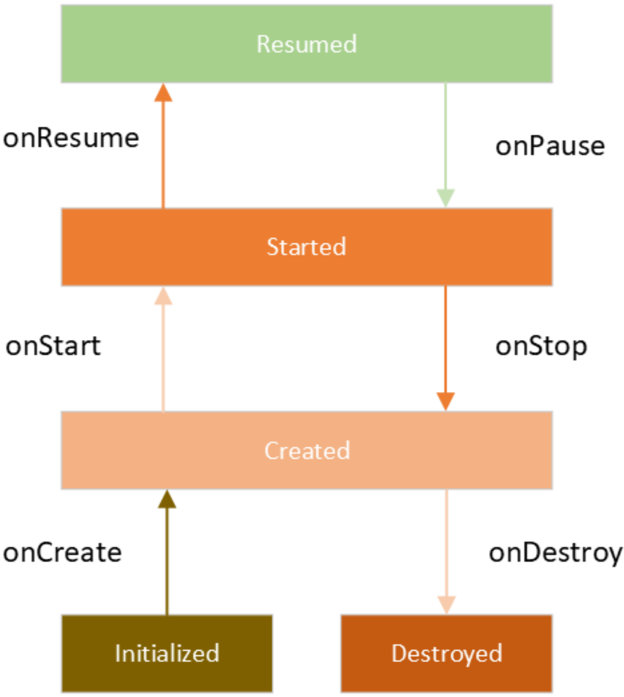
Kôd 2.5 Primjer promjenjivog stanja

U isječku kôda 2.5 definira se promjenjivo stanje `input` i komponenta `TextField` koja predstavlja polje za unos teksta i postavlja se da je naše stanje vezano za njezin unos. Stanje sastavne funkcije se gubi i ponovno postavi prilikom ponovnog sastavljanja ako nije spremljeno. U primjeru kôda stanje se mijenja kod svakog novog unosa što će izazvati ponovno sastavljanje koje će stanje staviti na inicijalnu vrijednost. Stanje se mora spremiti da ostane isto prilikom ponovnog sastavljanja. Spremanje se postiže `remember` funkcijom. Vrijednost koju dobije `remember` sprema prilikom početnog stavljanja i vraća je tijekom ponovnog sastavljanja.

```
var input by remeber { mutableStateOf("0") }
```

2.3.1. Aktivnosti i njihov životni ciklus

Početna točka Android aplikacija je neka klasa koja nasljeđuje `Activity` klasu iz Android paketa. Aktivnosti imaju na sebe vezan životni ciklus koji se sastoji od metoda koje se zovu u određenim trenucima „života“ aktivnosti.



Slika 2.1 Životni ciklus aktivnosti

Početak rada Android aplikacija će uvijek biti u `onCreate` metodi životnog ciklusa glavne aktivnosti. Kako korisnik koristi aplikaciju, zatvara je i ponovno otvara, pozivaju se druge metode iz životnog ciklusa. Na slici 2.1 vidi se da se aktivnost može vratiti u stanje ciklusa iz kojeg je došla i da postoji jedno početno stanje prilikom inicijalizacije i jedno završno stanje prilikom završetka rada aplikacije.

Metoda `onCreate` se zove samo jednom, odmah nakon inicijalizacije aktivnosti. Funkcija `onStart` se poziva kada je aplikacija vidljiva na ekranu uređaja. S njezine druge strane metoda `onStop` se poziva kad aplikacije nije više vidljiva na ekranu. Metoda `onResume` zove se kad je aplikacija dovedena u prvi plan i kad je moguća interakcija, a `onPause` kada se gubi fokus aplikacije[13].

Jedan od događaja kada je bitan životni ciklus aktivnosti je prilikom konfiguracijskih promjena. Do njih dolazi kada se stanje uređaja toliko korjenito promjeni da je najbolje sustavu ugasiti aplikaciju i ponovno je pokrenuti. Jedna od takvih promjena je promjena jezika uređaja, cijeli raspored će se morati promijeniti zbog smjera teksta i njegove dužine. Takvom promjenom se smatra i promjena orijentacije uređaja iz portreta u vodoravni i

obrnuto. Problem kojeg treba biti svjestan prilikom tih promjena je mogući gubitak stanja. Sustav će zvati `onDestroy` metodu da zatvori aktivnosti i nakon toga će je ponovno pokrenuti sa zadanim vrijednostima. Spremanje stanja kroz konfiguracijske promjene postiže se funkcijom `rememberSaveable`.

```
var input by rememberSaveable { mutableStateOf("0") }
```

2.3.2. Material Design

Material Design je dizajn sustav koji su osmislili i održavaju dizajneri i programeri iz Googla. Sadrži implementacije komponenti korisničkog sučelja i upute za bolje korisničko iskustvo za Android, Flutter i web. Najnovija verzija, Material 3, omogućuje prilagodljive boje, bolju pristupačnost i prilagodbu na veličinu ekrana[14]. Material omogućuje automatsko stvaranje palete boja i fonta koje će koristiti aplikacija s odabirom glavne izvorne boje[15]. Stvorenu temu moguće je preuzeti za Android Compose u obliku Kotlin klasa koje sadrže definicije boja za podloge i tekst, oblike i font, te definicije koriste se kao parametri `MaterialTheme` klase kao u kôdu 2.6. Ta klasa je sastavljena klasa koja se koristi kao omotač oko cijele aplikacije da sve ostale komponente koriste zadanu temu.

```
MaterialTheme(
    colorScheme = colorScheme,
    typography = AppTypography,
    shapes = shapes,
    content = content
)
```

Kôd 2.6 Klasa Material teme

Sve Compose komponente sadržane u Material biblioteci pridržavaju se njihovih standarda za bolje korisničko iskustvo. Primjeri Material komponenti su gumbovi, tekstna polja i klizači za korisnikov unos, karte i karuseli za prikaz sadržaja i kartice i trake za navigaciju.

2.4. Arhitektura mobilne aplikacije

Arhitektura aplikacije upućuje kako raspodijeliti odgovornosti između klasa aplikacije. Dobro dizajnirana arhitektura olakšava skaliranje i dodavanje novih funkcionalnosti aplikaciji te može pojednostaviti suradnju u timovima. Dva glavna principa arhitekture u mobilnim aplikacijama su razdvajanje odgovornosti i pokretanje korisničkog sučelja iz modela. Razdvajanje odgovornosti navodi da svaka klasa ima svoju

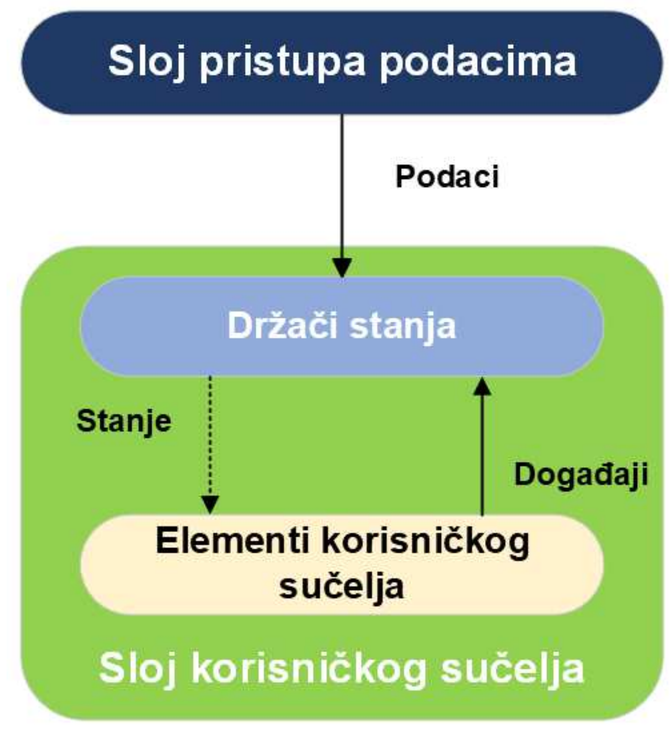
odgovornost i samo tu. U mobilnim aplikacijama primjer kršenja tog principa bi bilo pisanje cijelog kôda u komponente korisničkog sučelja koje bi trebale služiti samo za prikaz sučelja i interakciju s korisnikom. Pokretanje korisničkog sučelja iz modela navodi da bi stanja i njihove promjene, iz kojih se pokreće korisničko sučelje, trebale dolaziti iz modela. Modeli su komponente namijenjene rukovanjem podacima aplikacije. Odvojeni su od elemenata korisničkog sučelja i komponenti aplikacije te na njih ne utječe životni ciklus aplikacije.

S pomoću ta dva principa svaka arhitektura aplikacije morala bi imati barem prva dva sloja i neobavezan treći sloj:

1. Sloj korisničkog sučelja
2. Sloj za pristup podacima
3. Sloj domene

2.4.1. Sloj korisničkog sučelja

Uloga sloja korisničkog sučelja (ili prezentacijskog sloja) je prikaz podataka na ekranu. Prilikom promjene podataka zbog interakcije korisnika ili vanjskog unosa (naprimjer internet zahtjeva) korisničko sučelje se osvježi da prikaže promjenu. Sloj se sastoji od elemenata korisničkog sučelja koji prikazuju podatke i od držača stanja koji čuvaju podatke, predaju ih sučelju i upravljaju logikom[16]. Držači stanja predaju elementima samo podatke stanja koji su im potrebni za prikaz te događaje s pomoću kojih elementi mogu mijenjati stanje. Na slici 2.2 prikazan je tok podataka u sloju korisničkog sučelja.



Slika 2.2 Sloj korisničkog sučelja

U konkretnoj implementaciji aplikacije elementi korisničkoj sučelja će biti sastavne funkcije Jetpack Compose biblioteke. Držači stanja će biti klase koje nasljeđuju `ViewModel` klasu iz Android paketa. `ViewModel` je napravljen da čuva i predaje stanja potrebna korisničkom sučelju. Sva stanja definirana u `ViewModel` čuvaju se kroz ponovno sastavljanje, konfiguracijske promjene i uništenje glavne instance aktivnosti. Sva stanja najbolje je definirati na jednom mjestu u Kotlinove `data` klase.

```
data class MyUiState(val name: String, val age: Integer)
```

Zatim se u `ViewModel` klasi definira stanje koje je tipa `data` klase i potrebne funkcije za promjenu tih stanja.

```
class MyViewModel : ViewModel() {
    val _uiState = mutableStateOf(MyUiState())
    fun updateName(input: String) {
        _uiState.value = _uiState.copy(name = input)
    }
    fun updateAge(input: Integer) {
        _uiState.value = _uiState.copy(age = input)
    }
}
```

Kôd 2.7 Primjer držača stanja s `ViewModel` klasom

Sada kad neki element korisničkog sučelja mijenja stanja predaje mu se samo to stanje koje mijenja i ekvivalentna funkcija za postavljanje stanja koja će se zvati kod događaja kada je potrebno postavljanje.

```
val viewModel = MyViewModel()
TextField(
    value = viewModel._uiState.name,
    onValueChange = { viewMode.updateName(it) },
    modifier = Modifier
)
```

Time se razdvaja odgovornost čuvanja stanja na držač stanja, dok se za prikaz i interakciju brinu elementi korisničkog sučelja. Takvim pristupom uvodimo i princip jedinstvenog izvora istine odnosno podataka. Jedinstveni izvor istine je vlasnik podataka i jedino ga on može mijenjati. Prednosti tog principa su centralizacija svih promjena tog podatka, zaštita podataka od petljanja i omogućuje pratnju promjene podataka za lakšu detekciju greški.

2.4.2. Sloj pristupa podacima

Sloj pristupa podacima aplikacije sadrži poslovna pravila. Poslovna pravila daju vrijednost aplikaciji, to su pravila koja definiraju kako aplikacija stvara, sprema i mijenja podatke[17]. Sloj pristupa podacima sadrži se od spremišta (eng. *repository*) koja sadrže nula ili jedan izvor podataka. Za svaki tip podataka koji će aplikacija koristiti potrebno je jedno spremište. Spremišta su zadužena za izlaganje podataka ostatku aplikacije, centralizaciju promjena podataka, rješavanje sukoba između više izvora podataka, apstrakciju izvora podataka od ostatka aplikacije i provjeru poslovnih pravila. Svaka klasa izvora podataka treba imati odgovornost raditi s jednim izvorom. Najčešći izvori su datoteke, mrežni izvor ili lokalna baza podataka. Ostali slojevi nikad ne smiju imati izravan pristup izvoru podataka. Sva interakcija s podatkovnim slojem mora prolaziti kroz spremišta. Takav pristup omogućuje strogo praćenje poslovnih pravila i fleksibilnost korištenog izvora podatka u spremištu. Podaci koje će podatkovni sloj izložiti moraju biti neizmjenjivi da se spriječi njihovo petljanje od drugih klasa što bi moglo dovesti do nekonzistentnog stanja u spremištu.

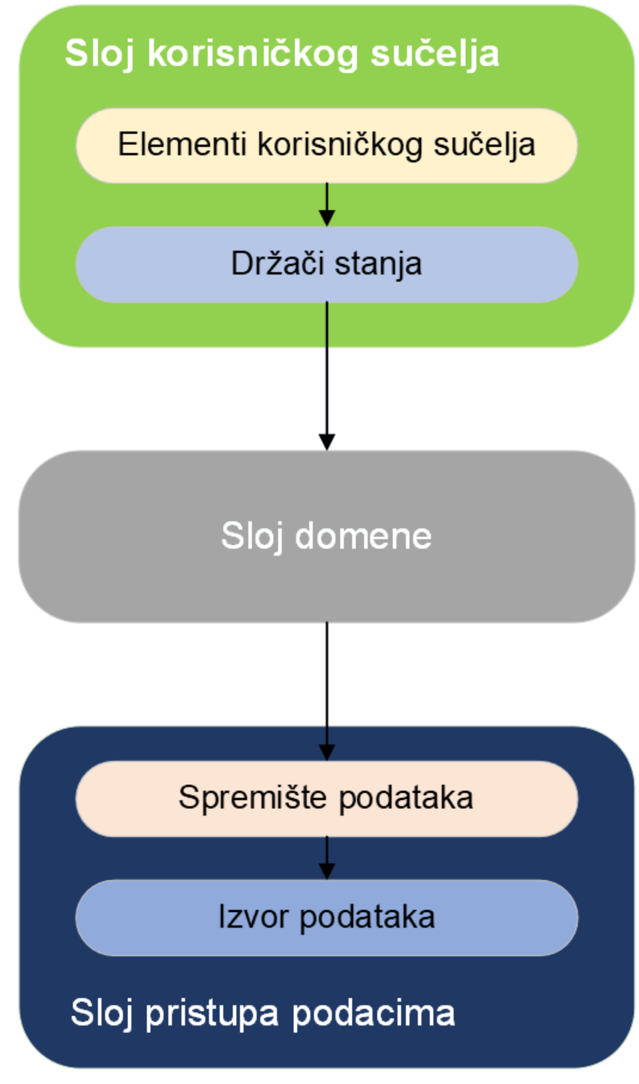
```
interface UserRepository { ... }

class UserFileRepository (
    fileDataSource: DataSource
) : UserRepository { ... }

class UserNetworkRepository (
    apiDataSource: ApiService
) : UserRepository { ... }
```

Kôd 2.8 Spremišta ovisno o izvoru podataka

Sva spremišta podataka za isti tip moraju pružati jednake mogućnosti pa se te mogućnosti definiraju u zajedničkom sučelju. Svako konkretno spremište će prilagoditi implementaciju tog sučelja svom izvoru podataka. U primjeru kôda 2.7 definirano je sučelje za spremište korisnika i dva konkretna spremišta za datoteku i za mrežni izvor. Spremištima će izvor podataka proslijediti aplikaciji u njihovom konstruktoru.



Slika 2.3 Slojevi arhitekture aplikacije

Za potrebe aplikacije u sloju pristupa podacima trebat će komunikacija s web serverom koji prati REST (Representational State Transfer) arhitekturu. U toj arhitekturi klijent komunicira sa serverom s HTTP (Hypertext Transfer Protocol) porukama koje sadrže URI (Uniform Resource Locator) kojim je identificiran željeni resurs, prikladnu HTTP metodu ovisno o operaciji nad resursom, zaglavlje i tijelo poruke. Glavne HTTP metode koje se koriste su GET za dohvaćanje resursa, POST za stvaranje novog resursa, PUT za ažuriranje postojećeg resursa i DELETE za brisanje resursa. Kao odgovor od servera klijent će dobiti status kôd koji naznačuje uspjeh ili neuspjeh operacije, zaglavlje i tijelo poruke koje je najčešće u JSON (JavaScript Object Notation) formatu.

Za komunikaciju s REST serverom koristit će se biblioteka Retrofit[18]. Biblioteka stvori kôd potreban za komunikaciju iz putanja za resurs, definirane HTTP metode i dodatnih parametara. Retrofit sam formatira zahtjev i pretvara odgovore u objekte definirane kao povratni tip zahtjeva.

```
Retrofit.Builder()
    .baseUrl("...")
    .client(okHttpClient)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Kôd 2.9 Stvaranje Retrofit objekta

U kôdu 2.8 stvara se Retrofit objekt za stvaranje zahtjeva kojemu je potrebno definirati početi URI na adresu servera i pretvarač za obradu zahtjeva i odgovora.

```
interface ApiService {
    @GET("users/{user}")
    fun getUser(
        @Path("user") user: Integer
    ): Call<UserData>
    @POST("users")
    fun postUser(
        @Body userData: UserData
    ): Call<Integer>
}
```

Kôd 2.10 Sučelje Retrofit zahtjeva

HTTP metode i zahtjevi definiraju se u posebnom sučelju kao u kôdu 2.9. Definiraju se funkcije anotirane s prikladnom HTTP metodom i putanjom na resurs. Kao parametre funkcijama moguće su varijable iz putanje ili podaci koji će biti u tijelu zahtjeva. Za odgovor se dobije objekt čiji tip je definiram povratnim tipom funkcije. `Call` tip dolazi iz Retrofit biblioteke i omogućuje obradu odgovora ovisno o tome je li zahtjev bio uspješan.

2.4.3. Prosljeđivanje ovisnosti

Prosljeđivanje ovisnosti je princip koji se često koristi u programiranju i prikladan je za razvoj na Androidu. Implementacijom prosljeđivanja ovisnosti postiže se mogućnost ponovnog korištenja programskog kôda, olakšano refaktoriranje i testiranje[19]. Klasama su često potrebne reference na druge klase, te reference se nazivaju ovisnosti te klase. Postoji tri načina kako klasa dobije potrebnu ovisnost.

1. Klasa sama izgradi potrebnu ovisnost.
2. Klasa uzme ovisnost izvana. Naprimjer iz konteksta aplikacije ili sistema.
3. Klasa dobije ovisnost kao parametar. Aplikacija može proslijediti ovisnost prilikom izgradnje klase ili funkcijama kojima je potrebna.

Treći način je prosljeđivanje ovisnosti.

```
class Car {
    private val engine = Engine()

    fun start() {
        engine.start()
    }
}

fun main() {
    val car = Car()
    car.start()
}
```

Kôd 2.11 Klasa sama izgradi ovisnost

U kôdu 2.10 klasa `Car` je ovisna o klasi `Engine` i sama si ju izgradi. Problem kod takvog pristupa je što su sada te klase usko vezane, jedna instanca klase `Car` koristi jednu instancu klase `Engine` i nije moguće koristiti podrazrede ili alternativne implementacije. Usko vezana ovisnost o klasi `Engine` također otežava testiranje. Zato što `Car` sam izgradi ovisnost nije ju moguće zamijeniti instancom za testiranje.

```
class Car (
    private val engine: Engine
) {
    fun start() {
        engine.start()
    }
}

fun main() {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}
```

Kôd 2.12 Prosljeđivanje ovisnosti

U kôdu 2.11 u `main` funkciji potrebno je prvo izgraditi instancu `Engine` klase koju će aplikacija proslijediti `Car` klasi zato što ona o njoj ovisi. Prednost takvog pristupa je mogućnost ponovnog korištenja klase `Car`. Ako definiramo novi tip motora koji će biti podrazred klase `Engine` samo ga proslijedimo instanci automobila i nije potrebna promjena `Car` klase. Na isti način moguće je proslijediti instancu klase za testiranje.

Postoje dva glavna načina prosljeđivanja ovisnosti, prosljeđivanje kroz konstruktor i prosljeđivanje kroz polja. Prosljeđivanje kroz konstruktor korišteno je u primjeru kôda 2.9. Neke klase u Android sustavu, kao što su aktivnosti i fragmenti, stvara sistem pa nije moguće proslijediti ovisnost kroz konstruktor. Takvim klasama potrebno je proslijediti instancu nakon što je klasa već stvorena. Primjer takvog prosljeđivanja prikazan je u kôdu 2.12.

```
class Car  {
    lateinit var engine: Engine
    fun start() {
        engine.start()
    }
}

fun main() {
    val engine = Engine()
    val car = Car()
    car.engine = engine
    car.start()
}
```

Kôd 2.13 Prosljeđivanje kroz polje

U primjerima kôda napravljeno je ručno prosljeđivanje ovisnosti tako da se ručno stvori instanca i preda bez upotrebe biblioteke. U primjeru je bila sam jedna ovisnost pa je takvo rješenje moguće, no u pravoj aplikaciji će postojati mnogo klasa kojima će biti potrebno proslijediti ovisnosti i ručno prosljeđivanje neće više biti najbolje rješenje. U većim

aplikacijama za spajanje svih ovisnosti potrebno je mnogo šablonskog kôda. U višeslojnim aplikacijama za stvaranje objekta na višim razinama treba pružiti sve ovisnosti potrebne slojevima ispod njega. Na primjeru automobila potreban je motor, mjenjač i ostali dijelovi dok su tim dijelovima opet potrebni drugi. Također kada nije moguće izgraditi ovisnosti prije nego se proslijede, naprimjer prilikom lijene inicijalizacije, potrebno je stvoriti i održavati spremnik oko cijele aplikacije koji se brine o životnim ciklusima ovisnosti u memoriji. Postoje biblioteke koje rješavaju te probleme automatizacijom stvaranja i prosljeđivanja ovisnosti. Biblioteka za prosljeđivanje ovisnosti koju preporučuje Android je Hilt[20].

Hilt omogućuje prosljeđivanje ovisnosti stvaranjem spremnika za svaku klasu u aplikaciji i zbrinjavanjem o njihovim životnim ciklusima. Hilt je izgrađen uz pomoć druge biblioteke za prosljeđivanje ovisnosti Dagger[21]. Svaka aplikacija koja koristi Hilt mora sadržati `Application` klasu s anotacijom `@HiltAndroidApp`. Ta anotacija okida Hiltovo stvaranje kôda uključujući klasu koja će biti spremnik ovisnosti za cijelu aplikaciju[22].

```
@HiltAndroidApp
class MyApplication : Application() { ... }
```

Stvorena Hilt komponenta spojena je na životni ciklus `Application` objekta i roditeljska je komponenta aplikacije pa mogu druge komponente pristupati ovisnostima koje će pružati. Ovisnost u nekoj klasi definirana je `@Inject` anotacijom najčešće u konstruktoru klase kao u kôdu 2.13.

```
class MyRepository @Inject constructor(
    private val myService: MyService
) { ... }
```

Kôd 2.14 Definiranje ovisnosti u konstruktoru

Neke tipove koje će Hilt prosljeđivati kao ovisnosti nije moguće proslijediti kroz konstruktor. Do toga može doći kad je ovisnost definirana kao sučelje koje nije moguće stvoriti ili kada je ovisnost neka klasa kojoj nije vlasnik iz vanjskih biblioteka. Kod takvih slučaja potrebne su klase koje će Hiltu pružati instance tog tipa – Hilt moduli.

```
@Module
@InstallIn(Singleton::class)
interface DataModuleBinds {
    @Binds
    fun bindMyRepository(repository: NetworkMyRepository): MyRepository
}
```

Kôd 2.15 Hilt modul koji pruža sučelje

U kôdu 2.14 primjer je definicije Hilt modula s `@Module` anotacijom i `@InstallIn` anotacijom koja definira kako će i gdje modul biti stvoren. U modulu definirana je funkcija `provideMyRepository` anotirana s `@Binds` anotacijom koja definira koje će sučelje povratni tip instance implementirati (`MyRepository`) i u parametru definira koju konkretnu implementaciju pruži (`NetworkMyRepository`).

Kada je potrebno pružiti ovisnost o tipu iz vanjske biblioteke potrebna je funkcija anotirana s `@Provides` unutar modula. Ta funkcija Hiltu daje informaciju o tipu instance s povratnim tipom, parametri funkcije definiraju ovisnosti potrebne za gradnju instance i u definiciji funkcije se gradi i vraća instanca. U kôdu 2.15 funkcija `provideNetworkService` vraća tip instance `NetworkService` kojemu je potrebna ovisnost o `HttpClient` tipu i stvara instancu s pomoću `Builder` metode klase instance.

```
@Provides
fun provideNetworkService (
    httpClient: HttpClient
) : NetworkService {
    return NetworkService.Builder()...
}
```

Kôd 2.16 Funkcija za pružanje objekta tipa vanjske biblioteke

3. CroQoL aplikacija

CroQoL će biti aplikacija na Android operacijskom sustavu s mogućnosti dohvaćanja procijenjenog indeksa kvalitete života za određeni grad uz mogući odabir važnosti pojedinih faktora. Također će uz odgovarajuću autentifikaciju biti moguće ispuniti anketu informacija koje će doprinijeti izračunu indeksa. Za potrebe autentifikacije i stvaranja poziva na API za procjenu kvalitete života potreban je pristup internetu. Prilikom ispunjavanja ankete aplikacija će zatražiti korisnika za dozvolu pristupa lokaciji. Dozvole koje su potrebne aplikaciji za njezin rad se definiraju uz informacije o aplikaciji u AndroidManifest.xml datoteci[23]. U Manifest datoteci se deklariraju komponente aplikacije, ikone, oznake, potrebne mogućnosti hardvera uređaja za rad aplikacije te dozvole potrebne za njezin rad. Dozvole su potrebne za pristup osjetljivim podacima kao što su SMS poruke ili kontakti ili za pristup mogućnostima uređaja kao što su pristup lokaciji, internetu i kameri.

```
<manifest ...>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
</manifest>
```

Kôd 3.1 Definiranje dozvola

Na prikazu kôda 3.1 definirane su dvije dozvole potrebne za rad aplikacije. Prvo je mogućnost pristupa internetu bez koje aplikacija neće imati nikakve funkcionalnosti. Druga je mogućnost približnog pristupa lokaciji koja je potrebna prilikom ispunjavanja ankete. Za lokaciju postoje više mogućih dozvola na temelju je li potreban pristup samo kada je aplikaciju u prvom planu ili i u pozadini te je li potreban precizan pristup ili samo približan. Za potrebu aplikacije dovoljan je približan pristup lokaciji kada je aplikacija u prvom planu, a takva dozvola definirana je na prikazu kôda 3.1 s `ACCESS_COARSE_LOCATION` dozvolom[24].

3.1. Autentifikacija i autorizacija

Za autentifikaciju i autorizaciju korisnika koristi će se OAuth2 standard. OAuth2 je standard koji omogućuje web stranici ili aplikaciji pristup dozvoljenim podacima tog

korisnika ili drugim podacima u ime prijavljenog korisnika bez da klijentska aplikacija ikad ima saznanje o načinu i podacima kojima je korisnik ovjeren[25]. OAuth2 je prvenstveno autorizacijski standard koji nakon provjere autentičnosti aplikaciji predaje pristupni token koji aplikacija može koristiti. Za svrhu aplikacije token će se koristiti kao autentifikacija korisnika i sam token se neće koristiti. Prvi korak korištenja standarda je registriranje aplikacije na Auth0 web stranici. Potrebno je dati ime aplikaciji te odabrati kao tip mobilnu aplikaciju. Servis nam tada stvori domenu i identifikator klijenta. Domena je putanja s pomoću koje će aplikacija komunicirati sa servisom. Identifikator klijenta se koristi da servis zna s kojom aplikacijom komunicira. Servisu je potrebno definirati putanju na koju će preusmjeriti prilikom uspješne prijave korisnika te prilikom odjave. Mobilne aplikacije ne koriste putanje pa će biti iste i servis će aplikaciju samo obavijestiti, a ne preusmjeriti. Putanja se definira prema formatu:

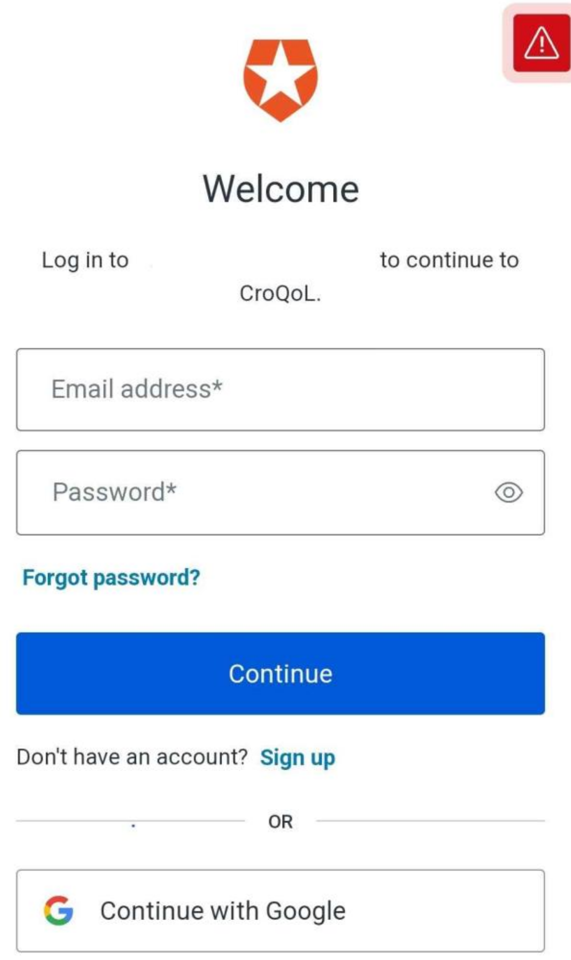
```
{SCHEME}://{DOMAIN}/android/{APP_PACKAGE_NAME}/callback
```

`SCHEME` je protokol korišten za komunikaciju. Za web aplikacije to bi bio HTTP ili HTTPS, za mobilne aplikacije može biti bilo što jer se ne koristi protokol. `DOMAIN` je domena koju stvori servis, a `APP_PACKAGE_NAME` je paket u aplikaciji na kojoj se nalazi njezina početna točka. Domenu i identifikator klijenta potrebno je spremiti u posebnu auth0.xml datoteku koja će biti resurs aplikaciji. Zatim je potrebno instalirati Auth0 paket koji sadrži implementacije za korištenje servisa. Prijavljivanje i odjavljivanje korisnika utječe na cijelu aplikaciju pa će se te funkcionalnosti nalaziti u glavnoj `ViewModel` klasi `MainViewModel`.

```
var userIsAuthenticated by mutableStateOf(false)
private lateinit var account: Auth0
fun setAccount(context: Context) {
    account = Auth0( ... )
}
fun login(context: Context) {
    WebAuthProvider
        .login(account)
        .withScheme(context.getString(R.string.com_auth0_scheme))
        .start(context, object: ...)
}
fun logout(context: Context) {
    WebAuthProvider
        .logout(account)
        .withScheme(context.getString(R.string.com_auth0_scheme))
        .start(context, object: ...)
}
```

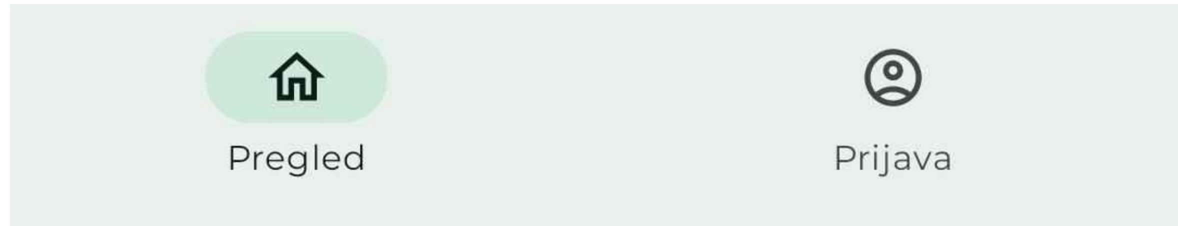
Kôd 3.2 Funkcije za prijavu i odjavu korisnika

Na prikazu kôda 3.2 prikazane su sve funkcionalnost potrebne za prijavu i odjavu korisnika. Stanje `userIsAuthenticated` čuva je li korisnik prijavljen. Varijabla `account` sadrži podatke o Auth0 računu, bitno domenu i identifikator klijenta. Funkcija `setAccount` se poziva prilikom pokretanja aplikacije da se stvori objekt `account`. Funkcije `login` i `logout` služe za prijavu i odjavu korisnika. One se pozivaju na prikladnim mjestima u aplikaciji prilikom pritiska na gumbove.



Slika 3.1 Ekran za prijavu korisnika

Na slici 3.1 prikazan je ekran koji se otvori prilikom poziva funkcije login. Aplikacija je u fazi razvoja pa je u prikazanom ekranu izbrisana domena, u završnoj verziji ona neće biti prikazana. Korisnik osim prijave ima mogućnost i registriranja te nastavka s Googleovim računom. Jednom prijavljeni korisnik ostane prijavljen i kroz ponovno pokretanje aplikacije, ako se želi odjaviti mora pritisnuti gumb za odjavu.



Slika 3.2 Aplikacijska traka ne prijavljenog korisnika

Slika 3.3 Aplikacijska traka prijavljenog korisnika

Na slikama 3.2 i 3.3 prikazana je aplikacijska traka ovisno o tome je li korisnik prijavljen. Kod pritiska na gumb Prijava se otvara ekran na slici 3.1 i korisnik može nakon prijave pristupiti ekranu Anketa i odjaviti se.

3.2. API za procjenu kvalitete života

Za izračun indeksa kvalitete života za neki grad koristit će se CroQoL API. API sadrži dvije putanje koje će aplikacija koristiti. Prva je za dohvaćanje svih gradova za koje je moguće izračunati indeks, a druga je za dohvaćanje indeksa kvalitete života i ostalih dostupnih informacija o odabranom gradu. API računa indeks kvalitete života kao zbroj šest drugih indeksa pomnoženih s odabranim težinama. Pojedini indeksi se računaju normalizacijom određenih vrijednosti na temelju njihovog raspona idealnih i stvarnih vrijednosti, samom normalizacijom drugih vrijednosti, množenje vrijednosti njihovim težinskim faktorima i konačno zbrajanje svih vrijednosti. Tih šest indeksa su:

- Klima
  - Klima se računa normalizacijom ljetne temperature, zimske temperature i vlažnosti zraka na temelju njihovih idealnih i stvarnih vrijednosti, te ugodnosti klime
- Okolina
  - Okolina se računa na temelju kvalitete zraka, kvalitete zelenih površina te dostupnosti i kvaliteti pitke vode

- Financije
  - Indeks financije se računa s pomoću prosječnog neto prihoda, cijeni stanova po metru kvadratnom, prosječnoj cijeni u restoranu za dvije osobe, prosječnoj cijeni kave, tjednoj potrošnji na namirnice i prosječnoj mjesečnoj cijeni komunalija
- Zdravstvo
  - Zdravstvo se računa na temelju broja bolnica i specijalista u gradu, prosječnom vremenu čekanja na termin i odgovora na hitni slučaj, opremljenost bolnica, broju kreveta i doktora u bolnicama, prosječnoj duljini trajanja liječenja i postotku zauzetih kreveta u bolnicama
- Sigurnost
  - Indeks sigurnosti se računa s pomoću broja policijskih postaja, prosječnom godišnjem broju zločina, nasilnih zločina i automobilskih nesreća, broju stanovnika te percepciji sigurnosti
- Promet
  - Indeks prometa se računa na temelju razine zagušenja prometa, dostupnosti javnog prijevoza, broju vozila na tisuću stanovnika i kvaliteti cesti

Za potrebe komunikacije potrebno je definirati modele koji će predstavljati podatke za gradove, težine indeksa, pojedinačne indekse te sveukupne odgovore koje će aplikacija primati kao rezultat poziva. Modeli se definiraju `data` klasama.

```
data class City(
    val cityId: Int,
    val name: String,
    val latitude: Double?,
    val longitude: Double?
)
```

Kôd 3.3 Model za podatke grada

Kao rezultat poziva na putanju za dohvat podržanih gradova aplikacija će dobiti listu gradova. Grad je potrebno modelirati s `cityId`, `name`, `latitude` i `longitude` poljima. `cityId` je identifikator grada s pomoću kojeg je moguće stvoriti poziv za indeks kvalitete života tog grada. Samo ime grada nije dovoljno za stvaranje poziva. Retrofit poziv na tu putanje će izgledati kao u primjeru kôda 3.4.

```
@GET("api/City/all")
fun getAllCities() : Call<List<City>>
```

Kôd 3.4 Poziv za dohvat gradova

Za odabir težina za pojedine indekse treba model koji pamti odabrane težine kao na primjeru kôda 3.5. `@SerializedName` anotacija naznačuje da će se taj element prilikom pretvaranja u zahtjev preimenovati u taj naziv. To je potrebno zato što API očekuje te nazive elemenata dok se oni u aplikaciji zovu drugačije.

```
data class IndexWeight(
    @SerializedName("climateIndex")
    val climateIndexWeight: Double,
    @SerializedName("environmentIndex")
    val environmentIndexWeight: Double,
    @SerializedName("trafficIndex")
    val trafficIndexWeight: Double,
    @SerializedName("financialIndex")
    val financialIndexWeight: Double,
    @SerializedName("safetyIndex")
    val safetyIndexWeight: Double,
    @SerializedName("healthIndex")
    val healthcareIndexWeight: Double
)
```

Kôd 3.5 Težine indeksa

Za poziv dohvata indeksa kvalitete života `IndexWeight` klasa će se proslijediti u tijelu zahtjeva i `cityId` će se staviti u samu putanju zahtjeva kao argument. Definicija poziva se nalazi u primjeru kôda 3.6.

```
@POST("api/QoLIndex/city/{city}")
fun postCityIndex(
    @Path("city") city: Int,
    @Body indexWeight: IndexWeight
): Call<CityIndexExtended>
```

Kôd 3.6 Poziv za dohvat indeksa kvalitete života

Kao rezultat se dobije struktura koja sadrži vrijednost konačnog indeksa, ali i vrijednosti za sve pojedinačne indekse. Potrebno je napraviti modele za svaki pojedini indeks i ujediniti ih u konačnu struktura koja će biti rezultat poziva.

```
data class Indexes(
    val climate: IndexValue<ClimateIndex>,
    val environment: IndexValue<EnvironmentIndex>,
    val financial: IndexValue<FinancialIndex>,
    val health: IndexValue<HealthIndex>,
    val safety: IndexValue<SafetyIndex>,
    val traffic: IndexValue<TrafficIndex>
)
```

Kôd 3.7 Model svih indeksa

Na prikazu kôda 3.7 definirana je klasa koja će držati sve pojedinačne indekse koji su definirani u drugim modelima. S tom klasom je sada moguće napraviti konačni model koji će biti rezultat poziva.

```
data class CityIndexExtended (
    val city: City,
    val index: Double,
    val indexes: Indexes
)
```

Kôd 3.8 Model rezultata poziva

3.3. Google Maps

U aplikaciji će se koristiti karta za prikaz lokacije korisnika te grada za koji je dobiveni indeks. Za korištenje Google Maps potreban je Google račun s kojim je moguće stvoriti projekt za koji se dobije API ključ za pristup kartama. API ključ mora ostati tajan pa se sprema u `secret.properties` datoteku i poziva se u Manifest datoteci.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="${MAPS_API_KEY}" />
```

Kôd 3.9 Definiranje ključa u Manifest datoteci

Korištenje karte omogućuje biblioteka koja sadrži kartu kao sastavljenu funkciju. Karti je moguće proslijediti postavke za geste i mogućnosti koje će biti podržane na karti. Biblioteka sadrži i sastavljene funkcije koje se mogu dodati na kartu. U aplikaciji će se koristiti Marker funkcija za prikaz odabrane lokacije korisnika.

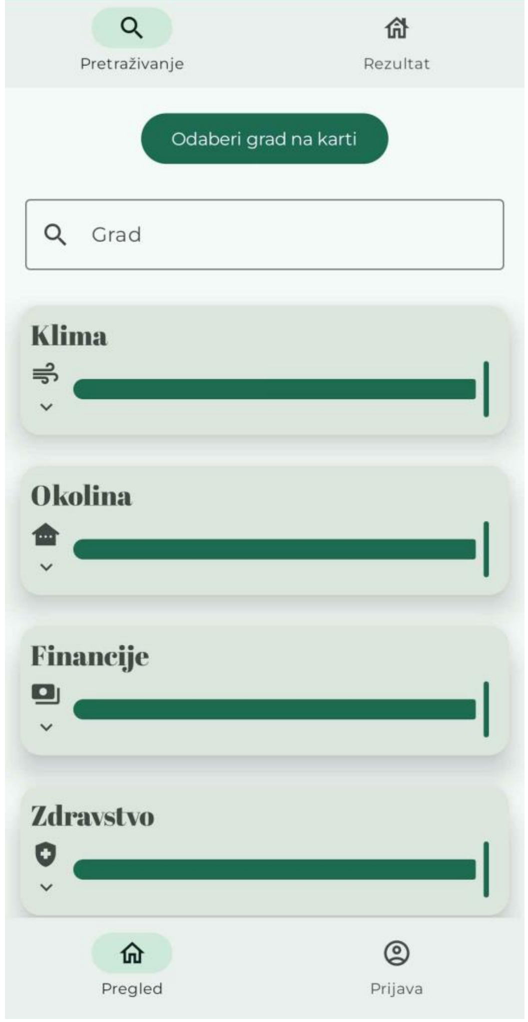
```
val cameraPositionState = rememberCameraPositionState {
    position = CameraPosition.fromLatLngZoom( ... )
}
GoogleMap(
    modifier = Modifier
        .height(312.dp)
        .padding(vertical = 16.dp),
    cameraPositionState = cameraPositionState,
    uiSettings = MapUiSettings(zoomControlsEnabled = false)
) {
    Marker(
        state = MarkerState(
            position = cameraPositionState.position.target
        )
    )
}
```

Kôd 3.10 Sastavljena funkcija za kartu

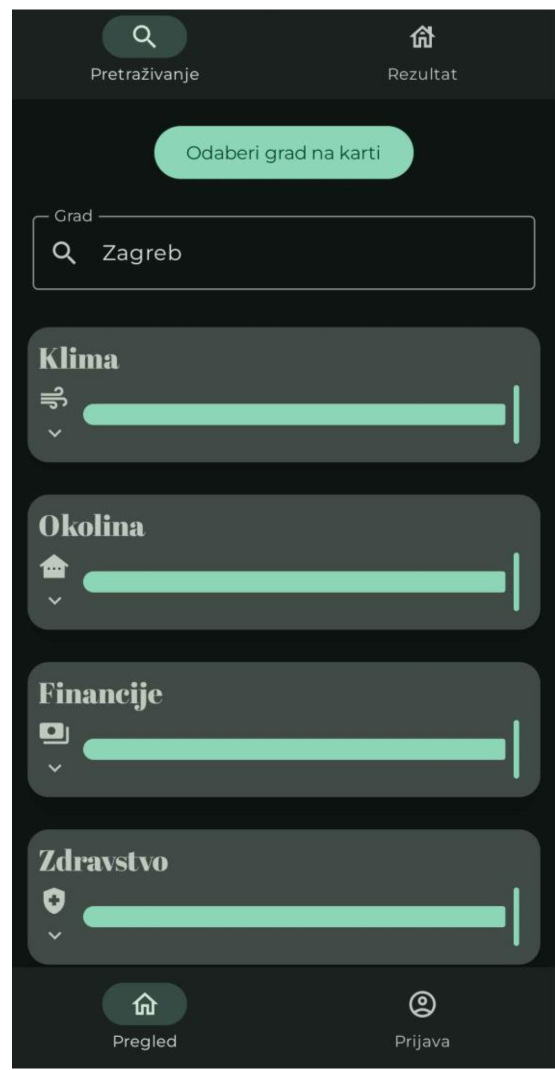
Na prikazu kôda 3.10 prvo je definirano stanje koje prati središnju poziciju karte i kojem je definirana početna pozicija. Stanje se predaje `GoogleMap` sastavljenoj funkciji uz modifikator za izgled i postavke za uklanjanje kontrola zumiranja. Na kartu je također stavljeni znak `Marker` koji prati središte karte da korisnik bolje vidi gdje se nalazi središte.

3.4. Početni zaslon

Početni zaslon aplikacije će omogućiti pretraživanje indeksa kvalitete života za odabrani grad uz moguć odabir težina pojedinih indeksa u računu.

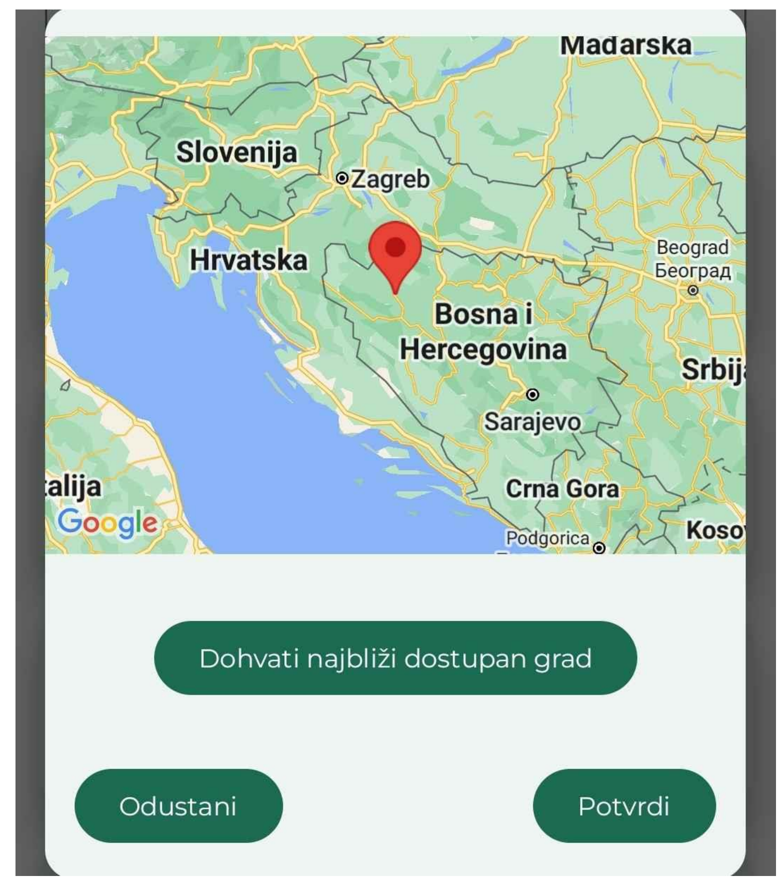


Slika 3.4 Početni ekran aplikacije u svijetlom načinu



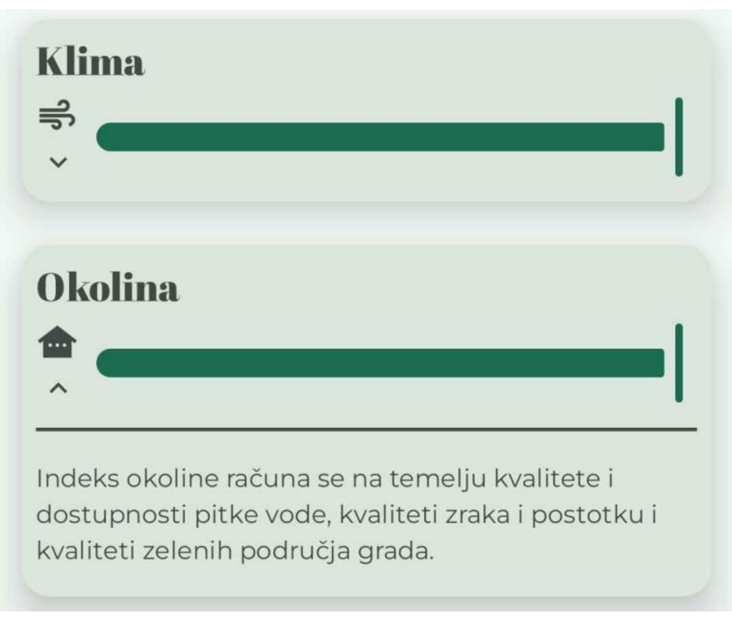
Slika 3.5 Početni ekran aplikacije u tamnom načinu

Početni ekran na vrhu ima traku za odabir pretraživanja i rezultata indeksa za grad. Na pretraživanju se prvo nalaze unosi za naziv željenog grada. Za pretraživanje potrebno je pritisnuti gumb u tekstnom polju za grad ili na tipkovnici gumb za sljedeću akciju nakon unosa. Grad je moguće i odabrati na karti klikom na gumb iznad tekstnog polja. Karta se zatim otvara u posebnom dijalogu nad početnim ekranom kao na slici 3.6. Korisnik se pozicionira gestama po karti markerom na lokaciju željenog grada i pritiskom na gumb aplikacija dohvati najbliži dostupan grad toj lokaciji. Ako je korisnik zadovoljan gradom pritiskom na gumb Potvrdi taj grad se stavlja u tekstno polje za grad i dijalog se zatvara. Ako korisnik nije zadovoljan može odustati od odabira na karti i vratiti se na početni ekran. Krajnji grad za koji će se pretraživati indeks se uvijek nalazi u tekstnom polju za naziv grada.



Slika 3.6 Odabir grada na karti

Ispod odabira grada se nalaze klizači za odabir težinskih faktora pojedinačnih indeksa. Klizači predstavljaju kontinuirane vrijednosti težina od nula do sto. Uz klizače dostupan je opis načina računanja tog indeksa pritiskom na strelicu.



Slika 3.7 Klizači za odabir težina

Klizači su programski ostvareni kao sastavljena funkcija kojoj se predaju potrebna svojstva. Takva implementacija omogućava posebno definiranje komponente, a posebna definicija samih podataka klizača. Potrebni podaci za stvaranje klizača definiraju se kao klase koje zatim možemo staviti u listu iz koje se klizači stvaraju.

```
@Composable
fun CroQoLSlider(
    label: String,
    value: Float,
    onValueChange: (Float) -> Unit,
    icon: ImageVector,
    expanded: Boolean = false,
    flipExpanded: () -> Unit = {},
    expandedContent: @Composable () -> Unit = {}
) { ... }
```

Kôd 3.11 Sastavna funkcija klizača

Svojstva potrebna za stvaranje sastavne funkcije klizača se vide na odsječku kôda 3.11. `label` predstavlja naziv indeksa, `value` je trenutna vrijednost klizača od nula do sto, `onValueChange` je funkcija koja se poziva za promjenu vrijednosti klizača, `icon` je ikona za ukras, `expanded` je varijabla koja naznačuje je li otvoren opis indeksa, `flipExpanded` je funkcija za promjenu `expanded` varijable i `expandedContent` je sastavljana funkcija koja će ići na mjesto sadržaja koji se prikazuje za opis.

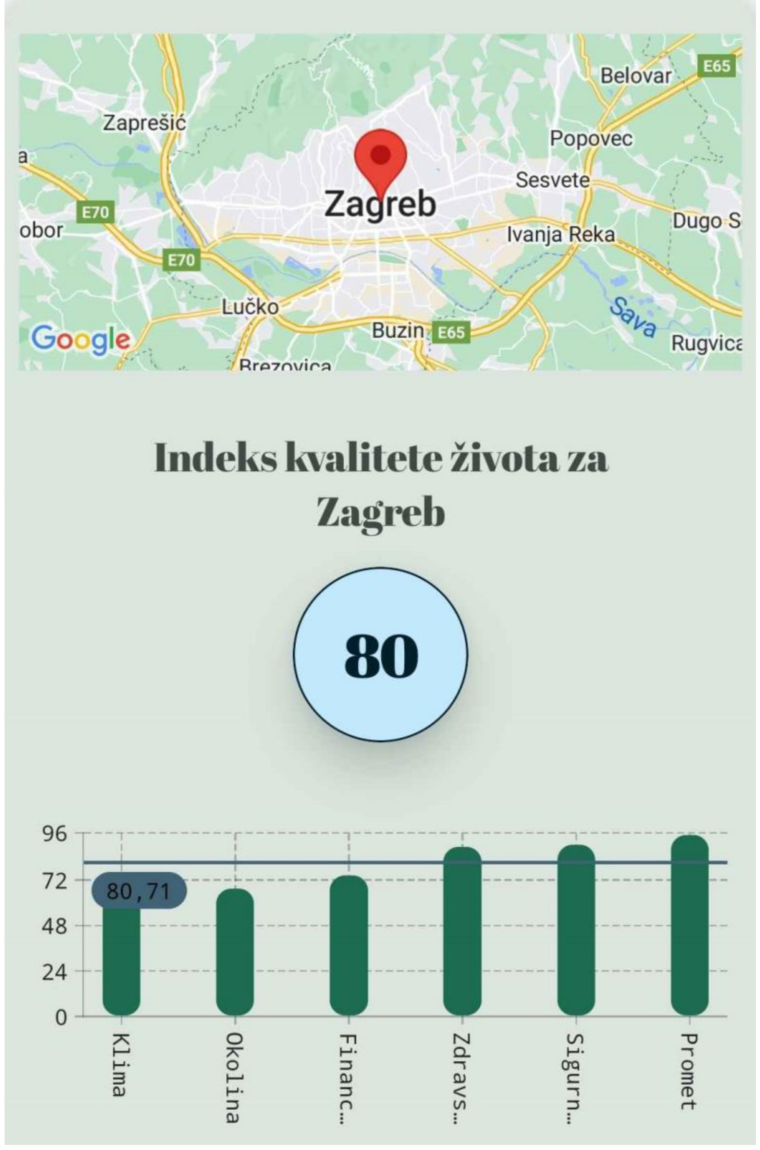
```
sealed interface CroQoLSliders {
    data object Climate : CroQoLSliders
    data object Environment : CroQoLSliders
    ...
}
data class CroQoLSlider(
    val label: String,
    val croQoLSlider: CroQoLSliders,
    val icon: ImageVector,
    val info: String
)
val allSliders = mutableListOf(
    CroQoLSlider(
        label = "Klima",
        croQoLSlider = CroQoLSliders.Climate,
        icon = Icons.Default.Air,
        info = "..."
    ), ...
)
```

Kôd 3.12 Definicije za klizače

Na isječku kôda 3.12 definirano je zatvoreno sučelje `CroQoLSliders` koje sadrži objekte za svaki indeks kojemu je potreban klizač, klasa `CroQoLSlider` koja sadrži

podatke potrebne za stvaranje klizača i lista `allSliders` koja sadrži stvorene klase s podacima za klizače.

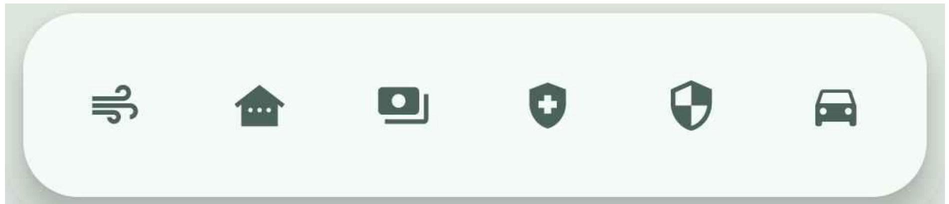
Pritiskom na gumb za pretraživanje, ako je valjani naziv grada, postavke težina koje je korisnik odabrao na klizačima se šalju u zahtjevu na API. Aplikacija se pomiče na ekran za rezultate koji se još nalazi na početnom zaslonu. Za vrijeme čekanja odgovora prikazuje se privremeni rotirajući krug, a nakon uspješnog odgovora se prikazuje ekran na slici 3.8.



Slika 3.8 Ekran nakon uspješnog dohvata indeksa

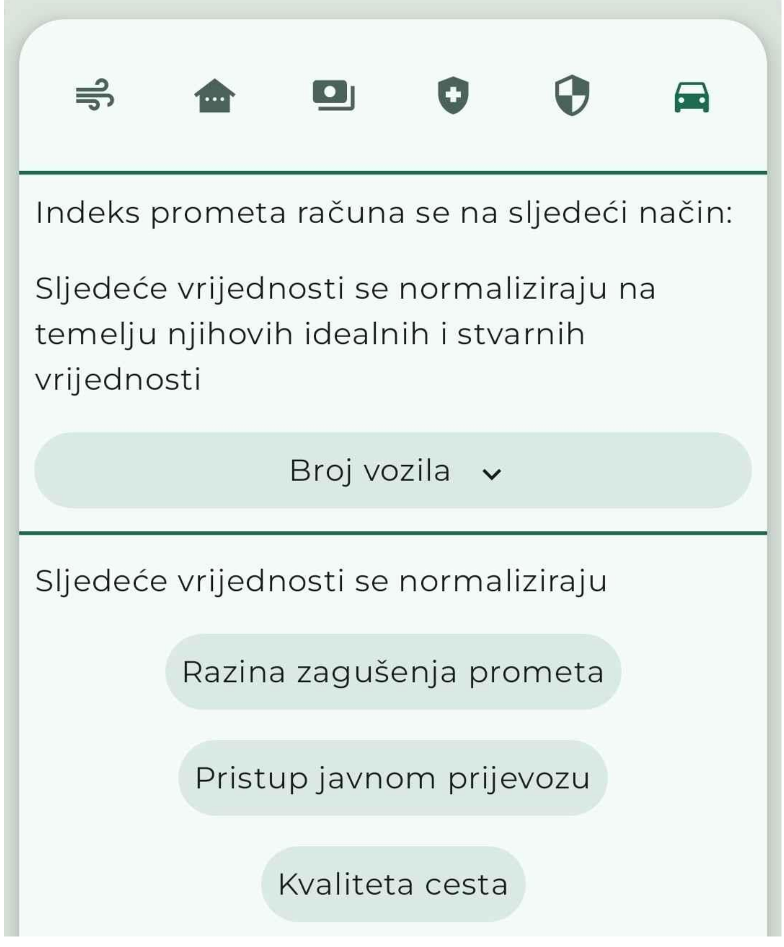
Ne ekranu se prikaže na karti grad za koji je pretražen indeks. Koordinate grada dobiju se iz odgovora na zahtjev. Ispod karte prikaže se dobiveni indeks kvalitete života, a ispod toga se prikaže stupčasti dijagram koji prikazuje pojedine vrijednosti indeksa i horizontalnu crtu koja predstavlja ukupni indeks. Za prikaz grafa korištena je vanjska

biblioteka Vico[26] koja sadrži sastavne funkcije i pomoćna stanja za stvaranje grafova. Ispod dijagrama se nalazi traka za detaljan odabir pojedinačnih indeksa sa slike 3.9.



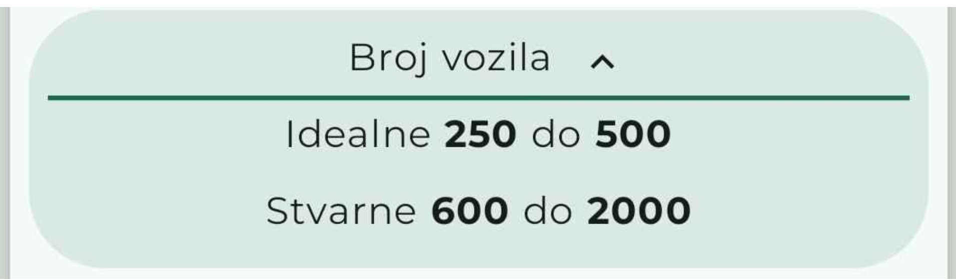
Slika 3.9 Traka za odabir informacija o pojedinom indeksu

Odabirom za pojedini indeks prikažu se informacije o detaljnom računanju tog indeksa. Na slici 3.10 vidi se primjer za indeks prometa.



Slika 3.10 Informacije za indeks prometa

Informacije se sadrže od vrijednosti koje se računaju normalizacijom idealnih i stvarnih vrijednosti te prikazom tih vrijednosti kao na slici 3.11. Također su nabrojane vrijednosti koje se samo normaliziraju i na kraju su prikazane težine za pojedine vrijednosti prema kojima se računa konačni indeks. Na slici 3.12 prikazani su težinski faktori za indeks sigurnosti.



Slika 3.11 Prikaz idealnih i stvarnih vrijednosti za normalizaciju

Slika 3.12 Težinski faktori za vrijednosti indeksa sigurnosti

3.4.1. Podaci i funkcionalnost početnog zaslona

Podaci i funkcionalnosti početnog zaslona sadržane su u `OverviewViewModel` klasi koja čuva stanje zaslona kroz konfiguracijske promjene i ponovno sastavljanje. Sva stanja potrebna početnom zaslonu definirana su u klasi `OverviewUiState`.

```
@Immutable
data class OverviewUiState(
    val searchCity: String = "",
    val indexDataUiState: IndexDataUiState = IndexDataUiState.Nothing,
    val safetySlider: Float = 100f,
    val financialSlider: Float = 100f,
    val trafficSlider: Float = 100f,
    val climateSlider: Float = 100f,
    val environmentSlider: Float = 100f,
    val healthcareSlider: Float = 100f,
    val isExpandedMap: MutableMap<Int, Boolean> = ...,
    val currentIndexedCity: String = "",
    val cities: List<City> = listOf()
)
```

Kôd 3.13 Klasa svih vrijednosti stanja početnog zaslona

U isječku kôda 3.13 prikazana je klasa koja predstavlja stanje početnog zaslona. Varijabla `searchCity` sprema tekst koji korisnik unosi u tekstno polje za naziv grada, definirane su varijable za svaki klizač težine indeksa, `isExpandedMap` je mapa koja pamti je li otvoren dodatni opis za indeks, `currentIndexedCity` pamti grad za koji je trenutno pretražen indeks. Varijabla `cities` sadrži sve dostupne gradove za pretraživanje. `indexDataUiState` je trenutno stanje podataka za indeks ovisno o fazi zahtjeva. Sve faze zahtjeva definirane su u zaključanom sučelju kao u primjeru kôda 3.14. Moguća stanja su `Nothing` kada se još nije dogodio niti jedan zahtjev, `Loading` kada je zahtjev napravljen, `Error` prilikom greške u zahtjevu što može biti ne podržani grad ili greška na poslužitelju i `Success` kada je u uspješno dohvaćen indeks. Sadržaj koji će se prikazati ovisi o tim stanjima.

```
sealed interface IndexDataUiState {
    data class Success(val indexData: CityIndexExtended):IndexDataUiState
    data object Nothing : IndexDataUiState
    data object Loading : IndexDataUiState
    data class Error(val error: String):IndexDataUiState
}
```

Kôd 3.14 Stanja podataka za indeks

`OverviewViewModel` klasa pamti tok klase stanja kao privatno stanje i izlaže ju kao preslika stanja u tom trenutku. Klasa je ovisna o `indexRepository` koji je zadužen za dohvaćanje podataka. Ovisnost će Hilt sam proslijediti klasi prilikom njezinog stvaranja.

```kotlin
@HiltViewModel
class OverviewViewModel @Inject constructor(
    private val indexRepository: IndexRepository
) : ViewModel() {

    private val _overviewUiState = MutableStateFlow(OverviewUiState())
    val overviewUiState: StateFlow<OverviewUiState> =
_overviewUiState.asStateFlow()
```

Kôd 3.15 Konstruktor i stanje koje čuva klasa za podatke početnog zaslona

Klasa, uz funkcije za postavljanje pojedinih stanja, sadrži i funkcije za dohvat dostupnih gradova, dohvat indeksa i pronalazak najbližeg grada.

```kotlin
init {
    indexRepository.getAllCities().enqueue( ... {
        override fun onResponse( ... ) {
            if (response.isSuccessful) {
                response.body()
                    ?.let { _overviewUiState.update {
                        cities = it
                    } }
            }
            if (!response.isSuccessful) {
                response.errorBody()
                    ?.let { _overviewUiState.update(
                        ... // Stanje na Error
                    ) }
            }
        }
        override fun onFailure( ... ) {
            t.message?.let {_overviewUiState.update(
                ... // Stanje na Error
            ))
            }
        }
    })
}
```

Kôd 3.16 Dohvaćanje svih dostupnih gradova

Dohvaćanje dostupnih gradova je potrebno odmah prilikom pokretanja aplikacije te se izvodi u `init` bloku klase koji se zove prilikom njezine inicijalizacije. U isječku kôda 3.16 se nalazi dohvaćanje dostupnih gradova s pomoću repozitorija i postavljanje `cities` varijable ako su gradovi uspješno dohvaćeni te postavljanje stanja na `Error` ako nisu.

```
fun getCityIndices() {
    _overviewUiState.update(indexDataUiState = IndexDataUiState.Loading)
    if (overviewUiState.value.cities.none { ... }) {
        _overviewUiState.update( ... )
        return
    }
    overviewUiState.value.cities.find { ... }?.let { city ->
        indexRepository.postCityIndex(
            cityId = city.cityId,
            indexWeight = overviewUiState.value.getIndexWeight()
        ).enqueue(object: Callback<CityIndexExtended> {
            override fun onResponse( ... ) {
                if (response.isSuccessful) {
                    response.body()?.let { ... }
                }
                if (!response.isSuccessful) {
                    response.errorBody()
                        ?.let { ...) }
                }
            }
            override fun onFailure( ... ) {
                t.message?.let {_overviewUiState.update( ... ) }
            }
        })
    }
}
```

Kôd 3.17 Funkcija za dohvaćanje indeksa za grad

Na isječku kôda 3.17 prikazana je funkcija za dohvaćanje indeksa za grad. Prvo funkcija postavlja stanje sučelja na `Loading`, zatim provjerava postoji li traženi grad u dostupnim gradovima, ako ne postavlja stanje sučelja na `Error` i vraća se iz funkcije. Ako grad postoji pronađe ga u listi dostupnih i s pomoću njegovog `cityId` i težinskih faktora iz klizača stvara zahtjev na API iz repozitorija. U slučaju uspješnog odgovora postavlja stanje u `Success` koje prima odgovor s podacima kao parametar, inače stavlja stanje u `Error`.

```
fun findClosestCity(latLng: LatLng): City? {
    return overviewUiState.value.cities.filter { it.longitude != null &&
it.latitude != null }
        .minByOrNull {
            val results = FloatArray(3)
            distanceBetween(
                latLng.latitude,
                latLng.longitude,
                it.latitude ?: 0.0,
                it.longitude ?: 0.0,
                results
            )
            results[0]
        }
}
```

Kôd 3.18 Funkcija za pronalazak najbližeg dostupnog grada

Klasa još sadrži funkciju `findClosestCity` prikazanu na kôdu 3.18. Ta funkcija se poziva prilikom korisnikovog odabira grada na karti za pronalazak najbližeg dostupnog grada. Funkcija prima kao parametar objekt koji sadrži zemljopisnu širinu i dužinu lokacije koju je odabrao korisnik. Na temelju lokacije računa udaljenosti svih dostupnih gradova od te lokacije uz pomoć funkcije `distanceBetween` iz Androidovog paketa za lokacije, sortira ih po udaljenosti i uzima onaj s najmanjom.

3.5. Anketa

Pristup zaslonu za anketu omogućen je korisniku kada se prijavi. Dolaskom na anketu aplikacija korisnika traži za dozvolu pristupa lokaciji kao na slici 3.13. Pritiskom na gumb za dopuštenje pristupa lokaciji otvara se dijalog za dopuštenja kao na slici 3.14.

Slika 3.13 Zaslon za anketu bez dopuštenja pristupa lokaciji

Slika 3.14 Dijalog za dopuštanje pristupa lokaciji

Jednom kad se dopusti aplikaciji pristup lokaciji na zaslonu je dostupna anketa. Na vrhu ankete se nalazi karta na kojoj je prikazana trenutna lokacija korisnika i tekstno polje za

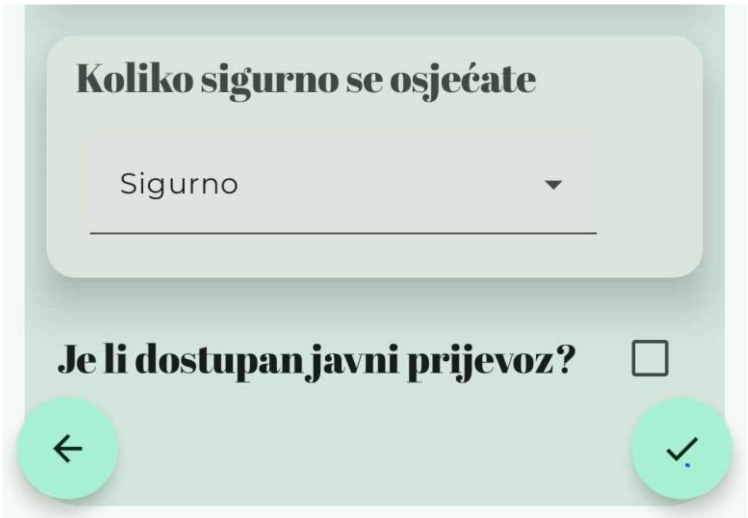
unos grada na koji se odnosi anketa. Ako korisnik unese naziv grada u tekstno polje taj će se grad koristiti umjesto lokacije. Slika tog dijela se nalazi na slici 3.15.

Slika 3.15 Karta lokacije korisnika i tekstnog polja za unos grada

Ispod tog dijela počinje anketa koja se sastoji od više tekstnih polja, potvrdnih okvira, klizača i polja s ograničenim odabirom. Na slici 3.16 se nalazi prvi dio ankete.

Slika 3.16 Prvi dio ankete

U dolje desnom kutu zaslona se nalazi gumb za prelazak na sljedeći dio ankete. Sljedeći dio ankete, koji je samo donjim dijelom vidljiv na slici 3.17, sastoji se od još unosa ankete i na dnu gumbovima za povratak na prvi dio u lijevom kutu i za predaju ankete u desnom kutu.



Slika 3.17 Donji dio ankete

3.5.1. Podaci i funkcionalnost ankete

Anketa sadrži svoju `ReviewViewModel` klasu koja čuva potrebne podatke u stanjima i sadrži funkciji potrebne za pravilan rad sučelja. Klasi je za rad potreban repozitorij koji će dohvatiti dostupne gradove.

```
@HiltViewModel
class ReviewViewModel @Inject constructor(
    private val indexRepository: IndexRepository
) : ViewModel() {
    private val _reviewUiState = mutableStateOf(ReviewUiState())
    val reviewUiState = _reviewUiState
    private val _reviewFormState = MutableStateFlow(ReviewForm())
    val reviewFormState: StateFlow<ReviewForm> =
_reviewFormState.asStateFlow()
```

Kôd 3.19 Klasa za čuvanje stanja i funkcionalnost ankete

Ova klasa će čuvati dva stanja, jedno za općenito stanje ankete, a drugo za unose ankete. Na isječku kôda 3.20 se nalazi definicija klase za čuvanje stanja korisničkog sučelja. Klasa se sastoji od varijable za grad na lokaciji korisnika `city`, `textFieldCity` koji predstavlja naziv grada napisan u tekstnom polju, `cities` što je lista svih dostupnih gradova, `marketLatLang` koji predstavlja zemljopisnu širinu i dužinu lokacije

korisnika, `isMarkerSet` naznačuje je li lokacija uspješno pronađena i `locationState` koji predstavlja stanje pronalaska lokacije.

```
data class ReviewUiState(
    val city: String = "",
    val textFieldCity: String = "",
    val cities: List<City> = listOf(),
    val markerLatLang: LatLng = LatLng(44.927, 16.383),
    val isMarkerSet: Boolean = false,
    val locationState: LocationState = LocationState.Nothing
)
```

Kôd 3.20 Klasa za podatke korisničkog sučelja ankete

Stanje pronalaska lokacije je tip `LocationState` i sličnog je oblika stanju podataka za indeks iz početnog zaslona. Sadrži se od četiri stanja definirana u zatvorenom sučelje kao na kôdu 3.21.

```
sealed interface LocationState {
    data object Success : LocationState
    data object Error : LocationState
    data object Loading : LocationState
    data object Nothing : LocationState
}
```

Kôd 3.21 Zatvoreno sučelje za stanje pronalaska lokacije

Dio klase za unose ankete prikazana je na isječku kôda 3.22. Klasa se sastoji od varijable koje čuvaju vrijednosti unosa korisnika. Većina tih vrijednosti su brojčane, ali postoje i tekstne iz ograničenog odabira mogućnosti kao za razinu buke i razinu sigurnosti. Takvi odabiri su spremljeni u liste i korisnik može kao vrijednost odabrati samo jednu iz padajuće liste.

```
data class ReviewForm(
    val climateComfortLevel: Float = 50f,
    val drinkingWaterQuality: Float = 50f,
    ...
)
val noiseLevels = listOf("Jako mirno", "Mirno",
"Normalno", "Bučno", "Jako bučno")
val safetyLevels = listOf("Jako nesigurno",
"Nesigurno", "Sigurno", "Jako sigurno")
```

Kôd 3.22 Dio klase za spremanje unosa ankete

Klasa `ReviewViewModel` prilikom inicijalizacije također dohvaća sve dostupne gradove kao na isječku kôda 3.16 i zatim sprema gradove u stanje. Dodatne funkcionalnosti koje klasa mora sadržavati su pronalazak lokacije korisnika i predaja ankete.

```
fun getCityFromLocation(context: Context) {
    updateLocationState(LocationState.Loading)
    val locationClient =
LocationServices.getFusedLocationProviderClient(context)
    val geocoder = Geocoder(context)
    viewModelScope.launch {
        locationClient.getCurrentLocation(...
        ).addOnSuccessListener { fetchedLocation ->
            if (fetchedLocation != null) {
                geocoder.getFromLocation(lat, long, 1) {
                    val city = it[0].locality
                    geocoder.getFromLocationName(city, 1) {location ->
                        val locationInfo = location[0]
                        if ( ... ) {
                            _reviewUiState.value = ...
                        }
                    }
                }
                updateLocationState(LocationState.Success)
            } else {
                updateLocationState(LocationState.Error)
            }
        }
            .addOnFailureListener { exception ->
                updateLocationState(LocationState.Error)
            }
    }
}
```

Kôd 3.23 Funkcija za dohvaćanje lokacije korisnika

Na isječku kôda 3.23 prikazana je dio funkcije `getCityFromLocation` za dohvat lokacije i zatim naziva grada na toj lokaciji. Funkcija prima objekt tipa `Context` koji predstavlja kontekst Android aplikacije iz kojeg je moguće dohvatiti lokaciju. Za dohvat lokacije i naziva grada postoje Android biblioteke koje sadrže klase `Geocoder` i `LocationServices`. `Geocoder` služi za dohvaćanje informacija o konkretnoj lokaciji kao što su država, grad i ulica. `LocationServices` služi za pronalazak točnih zemljopisnih koordinata lokacije korisnika. Pozivom funkcije `getCurrentLocation` pokreće se traženje lokacije. Poziv može biti uspješan ili neuspješan. Prilikom uspješnog dohvata lokacije poziva se još jedna funkcija `getFromLocation` iz `Geocoder` klase koja prima koordinate lokacije i vraća informacije o njoj.

```
fun submitForm( onSuccess: () -> Unit ) {
    val cityName = if ( ... ) textFieldCity else city
    reviewUiState.value.cities.find { ...}
        ?.let {
        indexRepository.postCityForm(...)
            .enqueue( ...  {
            override fun onResponse(...) {
                if (response.isSuccessful) {
                    _reviewFormState.value = ReviewForm()
                    onSuccess()
                }
                if (!response.isSuccessful) { ... }
            }
            override fun onFailure(...) {
                ...
            }
        })
    }
}
```

Kôd 3.24 Funkcija za predaju ankete

Na isječku kôda 3.24 prikazan je funkcija za predaju ankete. Funkcija prima kao parametar drugu funkciju koju zove nakon uspješne predaje ankete. U aplikaciji ta funkcija će korisnika vratiti na početni zaslon nakon predaje ankete te prikazati kratku obavijest o uspjehu predaje. Funkcija prvo određuje koji grad će koristiti ovisno je li korisnik napisao što u polje za naziv grada. Zatim pronalazi postoji li grad u dostupnim gradovima i ako postoji s pomoću repozitorija šalje zahtjev na API na čijoj putanji se nalazi identifikator pronađenog grada i u čijem tijelu se nalaze podaci iz ispunjene ankete. Prilikom uspješne predaje funkcija ponovno postavi sve vrijednosti ankete i zove funkciju `onSuccess`.

Zaključak

CroQoL mobilna aplikacije omogućuje korisniku brzu pretragu procijenjenog indeksa kvalitete života u hrvatskim gradovima. Indeks se računa pomoću šest pod indeksa koji detaljnije razmatraju uvjete u pojedinom gradu i kojima je moguće namjestiti težinu ovisno o prioritetima korisnika. Također uz autentifikaciju korisnik može ispuniti anketu iz koje će se podaci koristiti za buduće izračune.

Aplikacija je razvijena za operacijski sustav Android koji je najpopularniji operacijski sustav za pametne uređaje u Hrvatskoj. Android aplikacije se razvijaju u Android Studio razvojnoj okolini u programskom jeziku Kotlin koji je moderan i ima sažetu sintaksu. Compose je biblioteka koju je razvio Google za razvoj mobilnih aplikacija koja iskorištava prednosti Kotlin programskog jezika.

Arhitektura aplikacije je strukturirana na temelju oblikovnih obrazaca i programerskih principa koji osiguravaju učinkovitost i najbolje programerske prakse pri razvoju. Glavni principi koje je pratio razvoj aplikacije su prosljeđivanje ovisnosti, jedinstveni izvor istine i jednosmjerni tok podataka.

Programski kôd aplikacije dijeli se na kôd korisničkog sučelja koji preslikava podatke bitne aplikaciji u elemente korisničkog sučelja i na kôd za upravljanje podacima. Aplikacija za podatke gradova i računanje indeksa kvalitete života koristi CroQoL API.

Literatura

[1] Mobile Operating System Market Share in Croatia – April 2024. Poveznica: https://gs.statcounter.com/os-market-share/mobile/croatia; pristupljeno 28. Svibnja 2024.

[2] Meet Android Studio, Poveznica: https://developer.android.com/studio/intro; pristupljeno 29. Svibnja 2024.

[3] Run Apps on the Android Emulator, Poveznica: https://developer.android.com/studio/run/emulator; pristupljeno 29. Svibnja 2024.

[4] Develop a UI with Views, poveznica: https://developer.android.com/studio/write/layout-editor; pristupljeno 29. Svibnja 2024.

[5] Techie's Spot, Java Null Pointer Exception: Causes, Solutions, Best Practices, and Key Point, Medium, (2023. listopad). Poveznica: https://medium.com/@TechiesSpot/java-null-pointer-exception-causes-solutions-best-practices-and-key-points-80f4bd91a302; pristupljeno 30. Svibnja 2024.

[6] Data Classes, (2023. siječanj). Poveznica: https://kotlinlang.org/docs/data-classes.html; pristupljeno 30. Svibnja 2024.

[7] Higher-order functions and lambdas, (2023. rujan), Poveznica: https://kotlinlang.org/docs/lambdas.html; pristupljeno 31. Svibnja 2024.

[8] Extensions, (2023. rujan), Poveznica: https://kotlinlang.org/docs/extensions.html; pristupljeno 31. Svibnja 2024.

[9] Build better apps faster with Jetpack Compose, Poveznica: https://developer.android.com/develop/ui/compose; pristupljeno 1.Lipnja 2024.

[10] Compose modifiers, Poveznica: https://developer.android.com/develop/ui/compose/modifiers; pristupljeno 1. Lipnja 2024.

[11] Lifecycle of composables, Poveznica: https://developer.android.com/develop/ui/compose/lifecycle; pristupljeno 5. Lipnja 2024.

[12] State and Jetpack Compose, Poveznica: https://developer.android.com/develop/ui/compose/state; pristupljeno 5. Lipnja 2024.

[13] The activity lifecycle, Poveznica: https://developer.android.com/guide/components/activities/activity-lifecycle; pristupljeno 5. Lipnja 2024.

[14] What's Material?, Poveznica: https://m3.material.io/get-started; pristupljeno 10. Lipnja 2024.

[15] Material Theme Builder, Poveznica: https://material-foundation.github.io/material-theme-builder/; pristupljeno 11. Lipnja 2024.

[16] UI layer, Poveznica: https://developer.android.com/topic/architecture/ui-layer; pristupljeno 8. Lipnja 2024.

[17] Data layer, Poveznica: https://developer.android.com/topic/architecture/data-layer; pristupljeno 9. Lipnja 2024.

[18] Retrofit, Poveznica: https://square.github.io/retrofit/; pristupljeno 10. Lipnja 2024.

[19] Dependency injection in Android: https://developer.android.com/training/dependency-injection; pristupljeno 10. Lipnja 2024.

[20] Hilt, Poveznica: https://dagger.dev/hilt/; pristupljeno 10. Lipnja 2024.

[21] Dagger, Poveznica: https://dagger.dev/; pristupljeno 10. Lipnja 2024.

[22] Dependency injection with Hilt, Poveznica: https://developer.android.com/training/dependency-injection/hilt-android; pristupljeno 10. Lipnja 2024.

[23] App manifest overview, Poveznica: https://developer.android.com/guide/topics/manifest/manifest-intro; pristupljeno 18. Lipnja 2024

[24] Request location permissions, Povaznica: https://developer.android.com/develop/sensors-and-location/location/permissions; pristupljeno 18. Lipnja 2024

[25] What is OAuth2.0?, Poveznica: https://auth0.com/intro-to-iam/what-is-oauth-2; pristupljeno 18. Lipnja 2024.

[26] Vico, Poveznica: https://github.com/patrykandpatrick/vico; pristupljeno 20. Lipnja 2024.

Sažetak

Razvoj mobilne aplikacije za ocjenu kvalitete života u novoj sredini

Zadatak ovog rada je razvoj Android mobilne aplikacije CroQoL koja omogućava procjenu kvalitete života u hrvatskim gradovima. Klimatske promjene i urbanizacija mijenjaju uvjete života, a preseljenja zbog posla čine ocjenu kvalitete života ključnom. Aplikacija će koristiti CroQoL API za procjenu kvalitete života pojedinih gradova. API računa sveukupni indeks na temelju pod indeksa definiranih karakteristikama životne okoline bitnima korisnicima. Razvoj aplikacije uključuje korištenje Android Studia, Kotlin jezika i Jetpack Compose alata, s posebnim naglaskom na dizajn korisničkog sučelja, autentifikaciju korisnika i arhitekturu aplikacije.

Ključne riječi: Mobilna aplikacija, Android, Kotlin, Compose, Kvaliteta života

Summary

Development of a mobile application for assessment of the quality of life in a new environment

The task of this paper is the development of the Android mobile application CroQoL, which enables the assessment of the quality of life in Croatian cities. Climate change and urbanization are changing living conditions, and relocations for work make quality of life assessment crucial. The application will use the CroQoL API to assess the quality of life of individual cities. The API calculates the overall index based on the subindexes of the defined characteristics of the city essential to the user. Application development includes the use of Android Studio, the Kotlin programming language, and the Jetpack Compose tools, with special emphasis on user interface design, user authentication, and application architecture.

Keywords: Mobile application, Android, Kotlin, Compose, Quality of life