Reaktivno programiranje korištenjem okvira ReactiveX

Umiljanović, Terezija

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni preddiplomski studij računarstva

REAKTIVNO PROGRAMIRANJE KORIŠTENJEM OKVIRA REACTIVEX

Završni rad

Terezija Umiljanović

Osijek, 2019.

SADRŽAJ

1. UVOD

Zahtjevi i očekivanja korisnika računalnih programa i mobilnih aplikacija svakim danom postaju sve veća. Kako bi držali korak s time, razvojni inženjeri primorani su neprestano razvijati i nadograđivati stečena znanja i iskustva u vezi programiranja te posezati za novijim i prikladnijim konceptima. Jedan takav koncept je i reaktivno programiranje. Iako reaktivno programiranje postoji godinama tek je sada dobilo na zamahu. Ono se uzima kao rješenje za problem sve kompleksnijih asinkronih korisničkih sučelja. Kombinacija oblikovnih obrazaca Promatrač i Iterator s funkcionalnim programiranjem postavila je temelj za danas najrašireniji okvir za implementiranje reaktivnog programiranja, ReactiveX okvir. ReactiveX okvir omogućava implementaciju reaktivnog programiranje u većini najzastupljenijih programskih jezika. Jedan od njih je i Kotlin u kojem su napisani svi primjeri unutar ovog rada.

U drugom poglavlju ovog rada dan je uvid u reaktivnu paradigmu i njezin razvoj te su opisani osnovni koncepti i principi rada Rx okvira. U trećem su poglavlju kroz primjere prikazane osnovne gradbene komponente Rx okvira: promotrivi tok, operatori i raspoređivači.

Četvrto poglavlje ovog rada prikazuje implementaciju reaktivnog programiranja u konkretnoj Android aplikaciji. Prije opisa implementacijskih detalja dan je uvid u razvojno okruženje korišteno za izradu aplikacije, Android Studio. Također je opisana struktura paketa projekta, uključene biblioteke i MVVM arhitekturni obrazac.

1.1. Zadatak završnog rada

U teorijskom dijelu rada potrebno je opisati koncept reaktivnog programiranja uključujući osnovne pojmove, jezike i okvire koji ga omogućuju. U praktičnom dijelu ostvariti vlastito programsko rješenje za Android platformu koje ugrađuje koncepte opisane u teorijskom dijelu rada.

2. REAKTIVNO PROGRAMIRANJE

Reaktivno programiranje je u zadnje vrijeme postalo vrlo popularno zahvaljujući sve detaljnijoj i opširnijoj dokumentaciji koja programerima uvelike olakšava razumijevanje reaktivne paradigme. U reaktivnom programiranju sve se vrti oko toka podataka (engl. *data stream*), odnosno emitiranja događaja koji se mogu pronaći na raznim mjestima u kodu. Takav tok mogu činiti varijable, korisnikov unos, poziv preko aplikacijskog programskog sučelja (engl. *Application programming interface*, API), lokacija uređaja, klik, itd. Način praćenja promjena i rukovanja tim tokovima je zapravo ono što reaktivno programiranje čini zanimljivim i vrlo isplativim. Pošto je reaktivno programiranje samo po sebi širok pojam u ovom poglavlju bit će opisani njegovi osnovni koncepti i kvalitete.

2.1. Paradigme programiranja

Kroz godine se razvio velik broj stilova programiranja koji su svrstani u jednu ili više programskih paradigmi. Najopćenitija podjela programskih paradigmi je na imperativnu i njoj oprečnu deklarativnu.

Imperativna paradigma je najstarija te se koristi za rješavanje većine problema u inženjerskoj praksi. Ona naglasak stavlja na naredbe koje mijenjaju stanja programa dajući točne algoritme kako nešto treba biti obavljeno. Zato se imeperativni kod često sastoji od uvjeta, petlji i nasljeđivanja klasa [1]. Način na koji procesor redom čita i izvršava jednostavne naredbe je imperativan. U većini slučajeva to predstavlja način programiranja s kojim se novi programeri prvo susretnu kako bi što lakše mogli razumjeti rad računala. Najraširenije podskupine imperativne paradigme su proceduralno i objektno-orijentirano programiranje koji se najbolje poznaju kroz programske jezike**:** C**,** C#**,** C++**,** Java, PHP i Assembly.

Slika 2.1 prikazuje primjer imperativnog koda (napisanog u Kotlinu) gdje se može primijetiti:

1. Inicijaliziranje stanja programa
2. Mijenjanje stanja programa

Snalaženje i pisanje imperativnog koda kod većih i kompleksnijih asinkronih aplikacija, koje su u današnje vrijeme neizbježne, postaje vrlo izazovno i naporno te se programeri okreću boljim i efikasnijim alternativama kao što je to, primjerice, funkcionalno programiranje.

```
val numbers : (Mutable)List<Int!>! = Arrays.asList(1, 2, 3, 4, 5)
//1
var oddSum = 0

for (x : Int! in numbers) {
    if (x % 2 != 0) {
        //2
        oddSum += x
    }
}
```

Slika 2.1. *Primjer imperativnog koda*

Deklarativna paradigma, za razliku od imperativne, naglasak stavlja na operatore i na ono što treba biti obavljeno. Zbog toga je deklarativni pristup sličniji načinu na koji čovjek razmišlja što dovodi do toga da se deklarativni kod puno jednostavnije čita i razumije [2]. Funkcionalno programiranje spada u skupinu deklarativne paradigme. Ideja funkcionalnog programiranja proizašla je iz načina na koji se rješavaju matematičke funkcije. Rezultat svake funkcije f(x) ovisi samo o ulazu x. Tako se za isti ulazni x uvijek dobije isti rezultat. Takav će program eliminirati sve tzv. popratne efekte (engl. *side effect*) i ni u kojem slučaju neće mijenjati vrijednost ulaznog parametra. Postoje programski jezici koji podržavaju čisto funkcionalno programiranje kao što su Haskell, Mercury, Hope i drugi.

Program sa slike 2.2 radi isto kao i program sa slike 2.1, zbraja sve neparne brojeve iz liste, samo na deklarativan način. Može se primijetiti kako je deklarativni kod puno kraći i jednostavniji. Mnogi programski jezici koji su imali isključivo imperativni pristup u današnje se vrijeme sve više nadopunjuju i drugim oblicima programiranja. Primjer takvog programskog jezika je Java. Ona je u svojoj verziji 8 osigurala podršku za funkcionalno programiranje kroz lambda izraze koji se koriste kao zamjena za anonimne unutarnje klase koje implementiraju sučelje sa samo jednom metodom.

```
val numbers : (Mutable)List<Int!>! = Arrays.asList(1, 2, 3, 4, 5)
val oddSum : Int = numbers.filter { it % 2 != 0 }.sum()
```

Slika 2.2. *Primjer deklarativnog koda*

Reaktivna paradigma predstavlja zaseban deklarativan pristup programiranju koji se temelji na toku podataka i propagiranju njegovih promjena [4]. Primjer koji možda najbolje predočava

reaktivan pristup podacima opisan je u [5], jest onaj s promjenama u tablicama Microsoftovog alata Excel.

Recimo da je unutar Excel tablice polju C1 dodijeljena vrijednost preko funkcije SUM tako da ona zbraja vrijednosti iz polja A1 i B1. Nakon promjene vrijednosti u polju A1 ili B1, vrijednost u polju C1 bit će automatski ažurirana. Reaktivno programiranje najviše se koristi prilikom stvaranja nisko latentnih i vrlo interaktivnih aplikacije koje zahtijevaju stalno ažuriranje i najčešće su dio nekog reaktivnog sustava. Prema Reaktivnom Manifestu [6] glavne karakteristike reaktivnog sustava trebaju biti:

- Brza reakcija (engl. *Responsive*) – održavanje korisničkog sučelja ažurnim s najnovijim podacima
- Otpornost (engl. *Resilient*) – zadržavanje stabilnosti i dostupnosti prilikom pojave pogrešaka
- Elastičnost (engl. *Elastic*) – podnošenje različitih uvjeta opterećenja
- Upravljivost porukama (engl. *Message driven*) – komuniciranje među komponentama temeljeno na porukama

U ovoj tehnološkoj eri, koja za sobom povlači enormne količine podataka, za divove poput Facebooka, Amazona, Googlea, Microsofta i drugih ovakav reaktivan pristup je neophodan.

2.2. Nastanak i razvoj reaktivnog programiranja

Koncept reaktivnog programiranja prisutan je od ranih 70-ih godina prošlog stoljeća, no svoj veliki uspon doživljava kroz posljednje desetljeće. Najveći razlog tomu je zapravo masovno korištenje Interneta iza kojeg stoji ogroman broj aplikacija sa sve kompleksnijim asinkronim korisničkim sučeljima. Godine 2005. Microsoftov tim, predvođen Erikom Meijerom, odlučio je osmisliti programsko rješenje koje bi olakšalo razvijanje takvih asinkronih aplikacija. Nakon 4 godine intenzivnog rada predstavili su Rx.NET, skraćeno od *Reaktivne Ekstenzije za .NET okvir,* biblioteku čija je ideja zasnovana na kombinaciji oblikovnih obrazaca (engl. *Design Patterns*) Promatrač (engl. *Observer Pattern*) i Iterator (engl. *Iterator Pattern*) s funkcionalnim programiranjem [2]. U poglavlju 2.4 opisano je zašto je baš ta kombinacija postavila temelje reaktivnom programiranju.

Prvi veći uzlet za rekativno programiranje dogodio se 2013. godine kada su Jafar Husain (koji je iz Microsofta prešao u Netflix, ostavši vjeran Rx-u) i Ben Christensen, voditelj Netflixovog razvojnog tima, predstavili reaktivnu ekstenziju za Javu nazvavši je RxJava. To je proizišlo kao odgovor na sve zahtjevnije i kompleksnije potrebe Netflixovog sustava koji se prebacio s usluga

iznajmljivanja DVD na usluge internetskog prijenosa te je brojao preko 30 milijuna korisnika krajem 2012. godine [2]. Netflixov API sagrađen je na Java programskom jeziku te zbog toga ne čudi što je Java prva dobila svoju reaktivnu ekstenziju.

Slika 2.3. *Popis programskih jezika sa njihovim reaktivnim ekstenzijama*

Slika 2.4. *Logo ReactiveX platforme predstavlja električnu jegulju jer je Rx projekt zvan još i Volta*

Reaktivno programiranje je zatim polako počelo skupljati simpatije diljem tehnološke scene kao što su sučelje i potpora web i mobilnih aplikacija. Zapravo sve što je upravljano događajima (engl. *event-driven)* moglo je vrlo lako zablistati na reaktivan način. Širenju reaktivnog programiranja također je doprinijela mogućnost pristupa softverskom rješenju pošto je otvoreno za sve (engl. *open source*). Zbog toga su reaktivne funkcionalnosti ubrzo bile implementirane u raznim programskim jezicima stvorivši tako jedan višeplatformski standard [3] nazvan ReactiveX. Na slici 2.3 vidi se kako je većina najzastupljenijih programskih jezika dobila svoju reaktivnu ekstenziju.

2.3. RectiveX okvir za Android Platformu

Mobilne aplikacije su zbog svoje dinamičnosti i visoke interaktivnosti postale savršena meta za implementiranje reaktivnog programiranja. Osigurati što bolju komunikaciju s korisnikom i pritom paziti da se izbjegne bilo kakvo blokiranje korisničkog sučelja bila je misao vodilja prilikom dodavanja reaktivnog programiranja u Android. Programski jezici koji se koriste za razvoj Android aplikacija su Java i u posljednje vrijeme sve više Kotlin. Kotlin je zamišljen kao interoperabilan s Javom tako da njihovo povezivanje ne predstavlja problem, odnosno Java kod se može pokretati u Kotlin datotekama i obrnuto [7].

Za Android platformu koriste se četiri glavne ReactiveX ekstenzije: RxJava, RxKotlin, RxAndroid i RxBindings. Zbog već spomenute kompatibilnosti, Java reaktivnu ekstenziju je moguće koristiti u Kotlin projektima. Bez obzira na to, svakako se preporuča i korištenje RxKotlin-a koji je omotač oko RxJave, a sadrži dodatne komponente, kao što su funkcije proširenja (engl. *extension functions*), koje još više pojednostavljuju pisanje reaktivnog koda. RxAndroid dodatak je RxJavi za Android platformu koji pruža raspoređivače (engl. *Schedulers*) za pravljenje rasporeda na glavnoj ili nekoj od sporednih niti. To je vrlo bitno kada je potrebno da se dugački poslovi obavljaju u pozadini. RxBindings sadrži veliki broj korisnih metoda koje omogućuju omotavanje elemenata pogleda (engl. View) u promotrivi tok (engl. *Observable*).

2.4. Osnovni koncepti

Kao što je već spomenuto, osnovne koncepte ReactiveX-a postavila su dva oblikovna obrasca: Promatrač i Iterator te funkcionalno programiranje. Oba oblikovna obrasca dolaze iz skupine obrazaca ponašanja i detaljno su opisana u knjizi *Design Patterns* [9].

Oblikovni obrazac Promatrač zasniva se na Subjektu koji predstavlja izvor informacija i listi promatrača (engl. *Observers*) koji se pretplaćuju na taj izvor kako bi bili obavještavani o promjenama. Ovakav način omogućuje vezu jedan prema više jer Subjekt može posluživati više promatrača. Subjekt osigurava sučelje za prijavu i odjavu promatrača, dok ga za bilo kakvu promjenu obavještava pozivajući neku njegovu metodu kojoj ima pristup. Ovaj oblikovni obrazac opisuje reaktivnost u njezinoj srži jer se prilikom promjene subjekta automatski događa nekakva promjena unutar promatrača. Tu je primjetna sličnost s promjenama u Excelovim tablicama. Ovom obrascu koji je postavio temelje reaktivnom programiranju u reaktivnom svijetu dodane su još neke karakteristike

kao što je obavještavanje promatrača u slučaju pogreške (engl. *error*) ili kada je poruka završena (engl. *complete*).

Oblikovni obrazac Iterator definira sučelje za pristup i obilazak elemenata objekta bez otkrivanja njegove unutarnje strukture. Sučelje Iteratora mora sadržavati metodu za dohvaćanje idućeg elementa, a obilaženje elemenata može se implementirati na različite načine dodatnim metodama. Iterator sa slike 2.5 sam povlači podatke iz kolekcije (engl. *pull-based*) pozivajući *next()* metodu. Takav eksplicitan zahtjev za podacima predstavlja imperativan pristup što se u rekativnom programiranju pokušava izbjeći. Erik Meijer, prilikom kreiranja Rx-a, iskoristio je ovaj obrazac ali na način da podaci budu gurani (engl. *push-based*) prema promatraču [2]. Na slici 2.6 prikazan je reaktivan kod u kojem *cities* predstavlja promotriv tok podataka (imena gradova) na koji se moguće pretplatiti. Nakon što je cities dobio promatrača gurao mu je redom imena gradova što je ovaj odlučio ispisati. Detaljnije o komponentama Rx-a nalazi se u poglavlju 3.

```
val cities : MutableList<String> =
        mutableListOf<String>("London", "Istanbul", "Paris", "Madrid")
val iterator : MutableIterator<String> = cities.iterator()
while (iterator.hasNext()) {
    println(iterator.next())
}

/* Prints
London
Istanbul
Paris
Madrid
 */
```

Slika 2.5. *Primjer iteriranja liste u Kotlinu*

```
val cities : Observable<String!>! =
        Observable.just( item1: "London", item2: "Istanbul", item3: "Paris", item4: "Madrid")

cities.subscribe { it: String!
    println(it)
}


 /* Prints
London
Istanbul
Paris
Madrid
 */
```

Slika 2.6. *Primjer iteriranja liste na reaktivan način*

Zadnji vrlo bitan koncept za Rx je funkcionalno programiranje. On se može pronaći gotovo u svim Rx operatorima koji se koriste za kreiranje, filtriranje, kombiniranje, transformiranje i ostale akcije nad tokom podataka. Funkcionalno programiranje sigurno je za rad s nitima (engl. *Thread-Safe*) jer onemogućuje mijenjanje stanja globalnih varijabli ili interakciju sa bilo kakvim vanjskim izvorima. Također funkcionalno programiranje je pogodno za ulančavanje operatora zbog čega se reaktivni kod puno brže čita i razumije.

Recimo da funkcija *getPhones()*, iz primjera sa slike 2.7, šalje upit na poslužitelj i nazad vraća promotrivu listu podataka koja sadrži sve aktualne mobitele na tržištu. Kada su ti podaci emitirani kroz ulančane Rx operatore, koji su vrlo slični Kotlinovim funkcijama za operacije nad kolekcijama, vrlo lako se postiže željeni rezultat, odnosno prvih 7 mobitela čija je cijena manja od 6000kn a da su proizvedeni u 2019. godini. Ovako ulančavanje operatora može se poistovjetiti s oblikovnim obrascem Graditelj (engl. *Builder Pattern*) koji omogućava kreiranje objekata korak po korak ali s bitnom razlikom. Razlika je ta što u graditelju nije bitan poredak funkcija kojima se gradi objekt, dok je u Rx-u poredak operatora vrlo bitan. Bitan je iz razloga što operatori u Rx-u ne mijenjaju originalni promotrivi izvor svaki zasebno, već mijenjaju promotrivi tok podataka koji je nastao uporabom prethodnog operatora [10]. Tako će prema gornjem primjeru drugi filter operator djelovati nad već filtriranim podacima, odnosno mobitelima čija je cijena manja od 6000kn.

```
val phonesObservable : PublishSubject<Phone>  = getPhones()

phonesObservable
        .filter { phone -> phone.price < 6000 }
        .filter { phone -> phone.year == Year.now().value }
        .take( count: 7)
        .map { phone -> "model: " + phone.model + " price: " + phone.price + "kn" }
        .subscribe{println(it)}
```

Slika 2.7. *Primjer ulančavanja operatora u reaktivnom programiranju*

3. GLAVNE KOMPONENTE REACTIVEX OKVIRA

U ovom poglavlju proći će se kroz osnovne gradivne komponente Rx-a: Promotrivi tok, Operatore i Raspoređivače. Uz primjere koda bit će korišteni i kliker (pikule) dijagrami (engl. *Marble diagrams*) koji služe za vizualizaciju emitiranja podataka i rada operatora i tako olakšavaju njihovo razumijevanje. ReactiveX okvir sadrži mnoštvo korisnih klasa i operatora od kojih će neki u ovom poglavlja biti tek spomenuti dok će oni učestaliji biti prikazani na primjerima. Svi primjeri su pisani u programskom jeziku Kotlin te koriste biblioteke Rxjava i RxKotlin.

3.1. Promotrivi tok

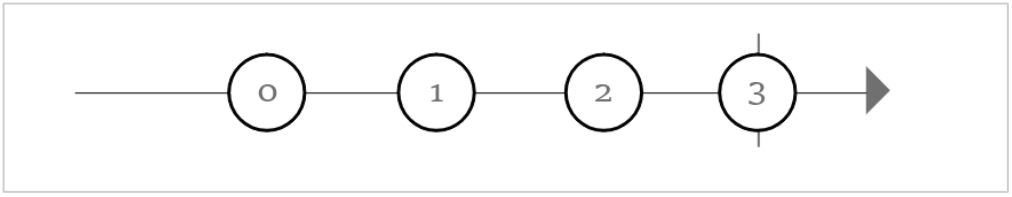
Promotrivi tok (engl. *Observable stream*) omogućuje asinkrono emitiranje događaja nakon pretplate te predstavlja osnovni element reaktivnog programiranja. Postoje tri vrste događaja koje tok može emitirati [3]:

- **Sljedeći** (engl. *a next event*) – sadrži idući vrijednosni podatak, slično kao kad se na *Iteratoru* pozove *next()*
- **Dovršen** (engl. *a complete event*) – daje do znanja da je emitiranje podataka iz toka uspješno završeno i da više neće biti emitiranja
- **Pogreška** (engl. *an error event*) – daje do znanja da je tok okončan zbog pogreške i da više neće biti emitiranja

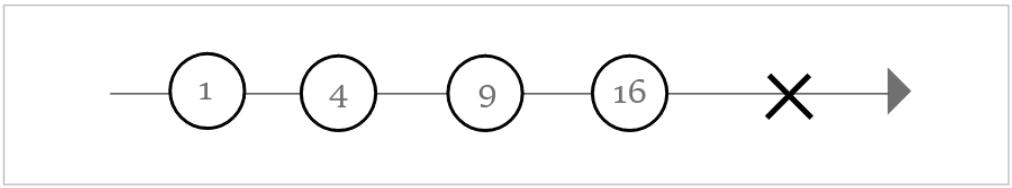
Tok može biti konačan i beskonačan. Konačan tok ima određen broj emitiranja te može završiti uspješno ili s pogreškom. Primjer gdje se koristi konačan tok je preuzimanje nekakve datoteke s Interneta [3]. U tom slučaju priželjkivan je pozitivan ishod, odnosno da će tok biti uspješno završen. Također može doći i do pogreške prilikom preuzimanja ili uspostavljanja internetske veze što bi dovelo do toga da i tok završi s pogreškom. Elementi korisničkog sučelja kao što su npr. polja za unos teksta ili pak gumbi, često predstavljaju beskonačni tok. Također i prekidač (engl. *switch button*) može biti beskonačni tok jer će emitirati samo vrijednosti uključeno \ isključeno. Kao i kolekcije tok može biti građen od samo jedne vrste podataka, no pomoću operatora može se transformirati iz jednog oblika u drugi.

Kliker dijagrame čini vremenska crta po kojoj se slijedno emitiraju podaci. Oni se čitaju s lijeva na desno. Pa je tako prvi emitirani podatak prema slici 3.2 '0', dok je zadnji '3' s kojim je emisija završena. Slika 3.3 prikazuje emitiranje kvadrata brojeva, ali i pogreške koja je okončala daljnje emitiranje. Vremenski razmak između emitiranih podataka ne mora uvijek biti jednak.

Slika 3.1. *Kliker dijagram - prazan tok*



Slika 3.2. *Kliker dijagram – uspješno završen tok*



Slika 3.3. *Kliker dijagram - tok okončan s pogreškom*

Kreiranje promotrivog toka

Za kreiranje promotrivog toka koristi se oblikovni obrazac Metoda tvornica (engl. *Factory Method*). Mogući operatori za stvaranje su: *create, defer, empty/never/throw, from, interval, just, range, repeat, start* i *timer*. U sljedećim primjerima bit će prikazani neki od njih, a na stranici ReactiveX-a postoji detaljan opis i svih ostalih [10].

Just operator kreira promotrivi tok samo s onim objektima koji su poslani kao parametri. Na primjeru sa slike 2.6 tok je kreiran pomoću just operatora i sadrži imena 4 grada. Iako na prvi pogled izgleda kao da ta imena tvore listu, po tipu toka koji je *Observable<String>* može se zaključiti da to nije tako. Imena gradova predstavljaju zasebne elemente koji se emitiraju jedan iza drugoga.

Primjer sa slike 3.4 prikazuje drugačiji slučaj u kojem se emitira samo jedan element, lista. To se postiže koristeći funkciju *listOf()*.Tu je tip toka *Observable<List<String>>*. U slučaju već gotove liste koju je potrebno razdijeliti na zasebne elemente pri kreiranju promotrivog toka koristi se operator *fromIterable* kao na primjeru sa slike 3.5. Ispisivanje emitiranih podataka događa se isključivo nakon pretplaćivanja (engl. *subscribe*) na promotrivi izvor. Pretplatom se potiče emitiranje podataka te daju upute što treba s njima biti učinjeno. Prilikom pretplate na promotrivi tok koji je kreiran *empty* operatorom očekuje se jedino emitiranje događaja o završenosti toka. Ako je potrebno kreirati tok koji ne sadrži ništa i nikada neće biti okončan onda se koristi *never* operator. S *range* operatorom kreira se tok s cjelobrojnim vrijednostima tako da se određuje vrijednost od koje počinje tok i ukupan broj elemenata koje je potrebno sekvencijalno emitirati. *Create* operator omogućava kreiranje događaja

```
val observable : Observable<List<String>!>! =
        Observable.just(listOf("London", "Istanbul", "Paris", "Madrid"))

    observable.subscribe{ it: List<String>!
        println(it)
    }

/* Prints
[London, Istanbul, Paris, Madrid]
 */
```

Slika 3.4. *Primjer just operatora*

```
val cities : List<String> = listOf("London", "Istanbul", "Paris", "Madrid")
val observable : Observable<String!>! =
        Observable.fromIterable(cities)

    observable.subscribe{ it: String!
        println(it)
    }

/* Prints
London
Istanbul
Paris
Madrid
 */
```

Slika 3.5. *Primjer fromIterable operatora*

koji će se emitirati na jedan posve drugačiji način. Pomoću njega se može kroz uvjete odrediti kada će neki događaj biti emitiran.

Slika 3.6 predstavlja jednokratni proces u kojem se može iskoristiti *create* operator. Pomoću njega kreiran je tok koji na promatraču poziva *onSuccess* u slučaju kada je prijavljivanje odobreno te *onError* u slučaju pogreške. Nakon pretplate na taj tok (slika 3.8) šalje se zahtjev za prijavom koji daje odgovor o uspješnosti u obliku zadatka (engl. *task*). Ako je zadatak uspješno izvršen promatrač

```
fun login(email: String, password: String): Single<Boolean> {

    return Single.create { emitter : SingleEmitter<Boolean!> ->
        authentication.signInWithEmailAndPassword(email, password)
                .addOnCompleteListener { task : Task<AuthResult!> ->
                    if (task.isSuccessful && task.isComplete) {
                        emitter.onSuccess( t: true)
                    } else {
                        emitter.onError(Exception(task.exception))
                    }
                }
    }
```

Slika 3.6. *Primjer autentifikacije pomoću create operatora*

```
public fun onLoginTaped(email:String, password: String) : Single<Boolean>
        = authManager.login(email, password)
```

Slika 3.7. *Pozivanje funkcije login() sa slike 3.6 unutar viewModela (posredničke klase između sučelja i baze)*

```
loginVM.onLoginTapped(email,password)
        .doOnSuccess { showHomeActivity() }
        .doOnError { showError(it) }
        .subscribe()
```

Slika 3.8. *Pozivanje funkcije onLoginTaped() unutar Aktivnosti (sučelja)*

će (u ovom slučaju to je aktivnost) o tome kroz onSuccess biti obavješten te će moći ažurirati korisničko sučelje, odnosno prebaciti ga s ekrana za prijavu na glavni ekran aplikacije.

Jedan od načina na koji se autentikacija vrši bez reaktivnog programiranja prikazan je u primjeru na slici 3.9. Na slici 3.10 vidljivo je kako se u prezenteru odlučuje kada i kako će biti ažurirano sučelje što nije slučaj kod prethodnog primjera gdje se o tome odlučuje unutar aktivnosti odnosno sučelja. Pošto na prvi pogled primjer autentikacije pomoću Rx-a izgleda kompliciranije postavlja se pitanje zašto uopće koristiti reaktivno programiranje. Odgovor leži u načinu na koji se implementiraju dodatne funkcionalnosti kao što je primjerice ponavljanje API poziva određeni broj puta ukoliko je vraćena greška. Pomoću Rx-a taj se slučaj može vrlo lako riješiti dodavanjem retry operatora u lanac operatora dok bi se bez Rx-a cijela ta logika morala pisati ručno.

```
fun login(email: String, password: String, onResult: (Boolean) -> Unit) {
    authentication.signInWithEmailAndPassword(email, password)
            .addOnCompleteListener { it: Task<AuthResult!>
                onResult(it.isComplete && it.isSuccessful)
            }
}
```

Slika 3.9. *Primjer autentifikacije bez Rx-a*

```
authManager.login(email, password){isSuccess : Boolean ->
  if(isSuccess){
    view.onLoginSuccess()
  }else{
    view.showLoginError()
  }
```

Primjer 3.10. *Pozivanje funkcije login() sa slike 3.9 unutar prezentera (posredničke klase između sučelju i baze)*

Može se primijetiti kako je u gornjem primjeru sa slike 3.6 korišten *Single*, a ne *Observable*. Single predstavlja jedan od tri specijalizirana podtipa promotrivog toka [3]:

- *Single* – emitira događaj o uspješnosti (*onSuccess*) prosljeđujući i vrijednost (što predstavlja kombinaciju događaja *next i completed*) ili događaj o pogrešci (*onError*)
- *Completable* – emitira događaj o završenosti (*onComplete*) ili o pogrešci, ne prosljeđuje vrijednost
- *Maybe* – predstavlja spoj ova dva tipa, može emitirati sve prethodno nabrojane događaje

Pretplata na promotrivi tok vraća jednokratan objekt (engl. *Disposable*) preko kojeg se može zaustaviti emitiranje podataka na način da se prekine pretplata [2]. Prekidanje pretplate je dobro i poželjno rabiti jer se tako sprječava curenje memorije do kojeg može doći ako se neki beskonačni tok ne okonča. Kako svaka pretplata ne bi morala biti ručno obustavljena osmišljen je spremnik za spremanje *disposable* objekata. *CompositeDisposable* je spremnik u koji se mogu dodavati sve pretplate bez obzira kojem toku one pripadale. Pozivanjem metode *dispose()* na spremniku osigurano je prekidanje svih pretplata. Na primjeru sa slike 3.11 za kreiranje promotrivog toka je korišten *defer* operator kako bi se odgodilo kreiranje toka do trenutka pretplate. Pošto se lista životinja popunjava tek nakon kreiranja toka, da nije korišten koristili *defer* operator ispis programa bi bio "[]".

```
val disposables = CompositeDisposable()
val farm = Farm()

val observable : Observable<String!>! = Observable.defer { Observable.fromIterable(farm.animals) }

farm.animals = mutableListOf("Cow", "Hen", "Duck", "Pig", "Sheep")

disposables.add(
        observable.subscribe { it: String!
            println(it)
        })

disposables.dispose()

/* Prints
Cow
Hen
Duck
Pig
Sheep
*/
```

Slika 3.11. *Primjer CompositeDisposable i defer operatora*

Subjekti

U prethodnim primjerima kreirani su promotrivi tokovi koji imaju postavljene sve elemente za emitiranje. Nakon emitiranja svih elemenata tok se zatvara. Što u slučaju kada je potrebno i nakon pretplate dodavati elemente za emitiranje u tok? Takve situacije je realnije očekivati prilikom razvijanja jedne aplikacije. Rješenje toga leži u subjektu (engl. *subject*) koji se u isto vrijeme ponaša kao tok i kao promatrač. Postoji četiri oblika subjekata [12]:

- PublishSubject – započinje prazan i emitira samo nove elemente
- *BehaviorSubject* – emitira i vrijednost koja je bila emitirana prije pretplate

- *ReplaySubject* – kao parametar uzima broj koji govori koliko će se zadnjih elemenata emitirati
- *AsyncSubject* – započinje prazan i emitira zadnji primljeni element prije završetka toka

Primjer sa slike 3.12 prikazuje princip rada *PublishSubject-a*. Na početku je kreiran PublishSubject koji predstavlja aukciju tako da emitira predmete reprezentirane znakovnim nizom (engl. *string*). Aukcija je odmah nakon kreiranja pokušala emitirati znakovni niz no pošto u tom trenutku nije imala niti jednog pretplatnika on se nije ispisao. Zatim su se aukciji priključila dva pretplatnika „pero“ i „ivo“ koji aktivno prate tok aukcije, što je vidljivo po bilježenju njihove zainteresiranosti (slika 3.9). Nakon toga aukciji se priključio još jedan pretplatnik „marica“ koja je propustila aukciju prethodna dva predmeta ali je nastavila pratiti daljnji tok aukcije. Nakon toga pretplatnik „pero“ je odustao od aukcije koja je dalje emitirala još samo jedan predmet te završila. Tek se tu priključio novi pretplatnik „miro“ koji je jedino primio obavijest da je aukcija završila. Iz ovog primjera vidljivo je kako pretplatnici na *PublicSubject* propuštaju sve elemente koji su emitirani

```
val auction = PublishSubject.create<String>()
auction.onNext( t: "White House")
val pero = auction.subscribeBy(
        onNext = { println("Pero is interested for $it") },
        onComplete = { println("Auction completed") }
)
val ivo= auction.subscribeBy(
        onNext = { println("Ivo is interested for $it") },
        onComplete = { println("Auction completed") }
)
auction.onNext( t: "Ferrari Dino")
auction.onNext( t: "Excalibur")
val marica = auction.subscribeBy(
        onNext = { println("Marica is interested for $it") },
        onComplete = { println("Auction completed") }
)
pero.dispose()
auction.onNext( t: "The scream")
auction.onComplete()
val miro = auction.subscribeBy(
        onNext = { println("Miro is interested for $it") },
        onComplete = { println("Auction completed") }
)
```

Slika 3.12. *Primjer - PublishSubject*

prije njihove pretplate. Također *PublishSubject* sve pretplatnike bez obzira na vrijeme pretplate obavještava o završenosti emitiranja ili pojavi greške.

Kad se na primjeru sa slike 3.12 zamjeni *PublishSubject* sa *BehaviorSubject-om* dobije se ispis koji je prikazan na slici 3.14. Po tom ispisu je vidljivo kako *BehaviorSubject* ima vrlo sličan način rada kao i *PublishSubject*. Razlika je u tome što *BehaviorSubject* svojim pretplatnicima emitira još i prethodnu vrijednost koju su propustili. *ReplySubjec*t je opet vrlo sličan *BehaviorSubject-u*. Razlika je u tome što *ReplySubject* uzima kao parametar broj koji određuje koliko će zadnjih emitiranih elemenata biti spremljeno u međuspremnik te emitirano pretplatniku prilikom trenutka pretplate. S ovim subjektom treba vrlo oprezno rukovati kako ne bi došlo do pretrpavanja memorije, pogotovo ako su pitanju elementi čije instance same po sebi zauzimaju više memorije kao naprimjer lista ili slike. *AsyncSubject* najviše odskače od ostalih subjekata iz razloga što emitiranje započinje tek kada je pozvan *onComplete* događaj te emitira samo zadnji element iz toku. Prilikom greške on ne emitira ništa.

```
Example of: PublishedSubject
Pero is interested for Ferrari Dino
Ivo is interested for Ferrari Dino
Pero is interested for Excalibur
Ivo is interested for Excalibur
Ivo is interested for The scream
Marica is interested for The scream
Auction completed
Auction completed
Auction completed
```

Slika 3.13. *Ispis koji daje pokrenuti kod sa slike 3.12*

```
Example of: BehaviorSubject
Pero is interested for White House
Ivo is interested for White House
Pero is interested for Ferrari Dino
Ivo is interested for Ferrari Dino
Pero is interested for Excalibur
Ivo is interested for Excalibur
Marica is interested for Excalibur
Ivo is interested for The scream
Marica is interested for The scream
Auction completed
Auction completed
Auction completed
```

Slika 3.14. *Ispis koji daje kod sa slike 3.12 kada je PublishSubject zamijenjen s BehaviorSubject--om*

3.2. Operatori

Rx biblioteka se pobrinula za to da kreiranje bilo kakve kompleksnije logike bude što lakše te je tako osigurala široku lepezu različitih operatora. Operatori su podijeljeni u zasebne skupine prema čijim se nazivima može lako naslutiti što rade. Tako, primjerice, postoje operatori za filtriranje, transformiranje, kombiniranje, matematičke izračune, rukovanje greškama itd. Ovdje će biti detaljnije

opisani neki važniji operatori iz prethodno nabrojanih skupina, dok se ostali mogu pronaći na stranicama ReactiveX okvira.

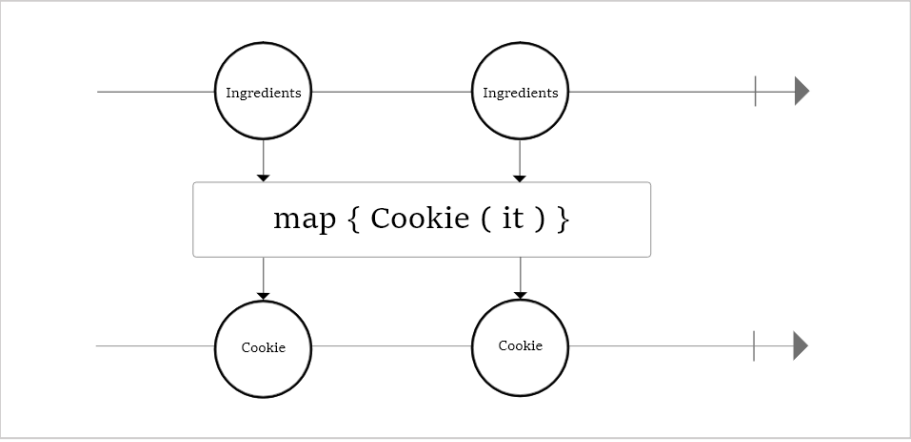
Operatori za transformiranje

Ovo skupina operatora je iznimno korisna kada je u pitanju reaktivno programiranje jer omogućuje izmjenu tipa podatka koji se emitira. Neki od operatora koji pripadaju ovoj skupini su: *buffer, map, flatmap, gropuBy, toList, switchMap, scan, window, materialize\dematerialize* itd. U ovom odjeljku opisani su operatori *map* i *flatMap.*

Map operator transformira elemente promotrivog toka u drugačiji tip podatka. Vrlo je sličan Kotlinovoj standardnoj *map* funkciji koja djeluje nad kolekcijama. Primjer u kojem ovakav operator može dobro poslužiti je kada se mrežni odgovor treba prilagoditi prikazu ekrana. Prema slici 3.15 prvi tok emitira elemente početnog tipa (*Ingredients*) dok tok ispod njega prikazuje elemente, nastale djelovanjem *map* operatora (*Cookie*) , koje će pretplatnik na kraju primiti.

```
class Ingredients(val flour: Double, val eggs: Int, val cocoa: Double, val milk: Double)
class Cookie(val ingredients: Ingredients) {
    fun makeCookie() : String  = "Mix ingredients very well"
}
Observable.just(Ingredients( flour: 250.0,  eggs: 3,  cocoa: 55.0,  milk: 0.5))
        .map { Cookie(it) }
        .subscribe { println(it.makeCookie()) }

// prints
// Mix ingredients very well
```

Slika 3.15. *Primjer – map operator*



Slika 3.16. *Primjer – Kliker dijagram – map operator*

Operator *flatMap*, za razliku od *map* operatora, emitirani element transformira u novi promotrivi tok na kojem se automatski vrši pretplata. Krajnji rezultat *flatMap* operatora je spoj svih emitiranih elemenata nastalih elementima početnog toka. Ovaj operator često pronalazi svoju svrhu kod ulančavanja mrežnih poziva jer izbjegava ugniježđene mrežne odgovore na vrlo efikasan način.

Jedan od primjera u kojem ulančavanje mrežnog poziva može biti potrebno je kada korisnik kroz neku aplikaciju za web kupovinu obriše proizvod iz košarice. U toj situaciji prvi API poziv radi se za brisanje tog proizvoda iz korisnikove košarice, a drugi API poziv radi se za dobivanje ažurirane liste proizvoda iz košarice i to samo ako je prvi poziv bio uspješan. Ulančavanje mrežnih poziva dovodi do koda koji je vrlo teško čitati i održavati [4] što je vidljivo na primjeru sa slike 3.17 . Povrh toga rukovanje pogreškama se odvija unutar svakog mrežnog odgovora zasebno što može imati za posljedicu još kompleksniji kod. Ovakav scenariji se može vrlo lako izbjeći korištenjem *PublishSubject-a* i *flatMap* operatora kako je prikazano na slici 3.18. Varijabla cartProductsChanged je instanca *PublishSubject-a* te služi kao primatelj i obavještavač o promjenama koje su se dogodile unutar košarice. Varijabla updateCart je pretplata na cartProductsChanged kojoj emitiranje praznog objekta (engl. *Unit*) predstavlja okidač za slanje API poziva. FlatMap operator u ovom slučaju je korišten kako bi se *Unit* objekt transformirao u novi promotrivi tok. Kako bi ova reaktivna implementacija bila moguća mrežni odgovori moraju biti omotani u promotrive tokove.

```
backend.deleteProduct(productId).enqueue(object : Callback<ResponseMessage> {
    override fun onResponse(call: Call<ResponseMessage>,
                            response: Response<ResponseMessage>) {
        backend.getProducts().enqueue(object : Callback<ProductsResponse> {
            override fun onResponse(call: Call<ProductsResponse>,
                                    response: Response<ProductsResponse>) {
                view.onCartDataReady(response.body().products)
            }

            override fun onFailure(call: Call<ChoreId>, t: Throwable) {}
        })
    }

    override fun onFailure(call: Call<ChoreId>, t: Throwable) {}
})
```

Slika 3.17. *Primjer ulančanog mrežnog poziva i ugniježđenog odgovora bez Rx-a*

```
val cartProductsChanged = PublishSubject.create<Unit>()

val updateCart = cartProductsChanged
        .flatMap { backend.getProducts() }
        .subscribe { view.onCartDataReady(response.products) }

backend.deleteProduct(productId)
        .subscribe { cartProductsChanged.onNext(Unit) }
```

Slika 3.18. *Primjer ulančanog mrežnog poziva korištenjem PublishSubject-a i flatMap operatora*

Operatori za filtriranje

Ova skupina obuhvaća operatore koji se koriste za filtriranje elemenata iz promotrivog toka i koji pretplatniku osiguravaju da dobije isključivo one elemente koji su mu potrebni. Neki od njih su: *debounce, distinct, elementAt, filter, first, ignoreElements, skip, take, sample* itd.

Na slici 3.19. može se vidjeti kako izgleda ulančavanje takvih operatora. S *distinct* operatorom osigurano je da se niti jedan string ne ponavlja, s filter operatorom izdvojeni su svi znakovni nizovi koji u sebi sadrže slovo c, a *takeLast* operator je od znakovnih nizova koji su ostali uzeo zadnja dva.

```
Observable.just( item1: "cappuccino", item2: "espresso", item3: "mocha",
        item4: "americano", item5: "macchiato", item6: "latte", item7: "mocha")
        .distinct()
        .filter { it.contains( other: "c") }
        .takeLast( count: 2)
        .subscribe{ println(it)}

/* prints
   americano
    macchiato
*/
```

Slika 3.19. *Primjer – filter operatori*

Operatori za kombiniranje

Ovi operatori rade s više promotrivih tokova te ih objedinjuju u jedan. To su operatori: *join, combineLatest, merge, startWith, switch, zip* itd. *Zip* operator vrlo je koristan kada je potrebno dva izvora različitih tipova spojiti u jedan. Prvi element u novom toku rezultat je spajanja prvih elemenata

iz svakog izvora. To znači da će broj ukupno emitiranih elemenata biti jednak onom izvoru koji ima manje elemenata. Ako jedan od ta dva izvora završi s emitiranjem i nastali tok će završiti sa svojim emitiranjem.

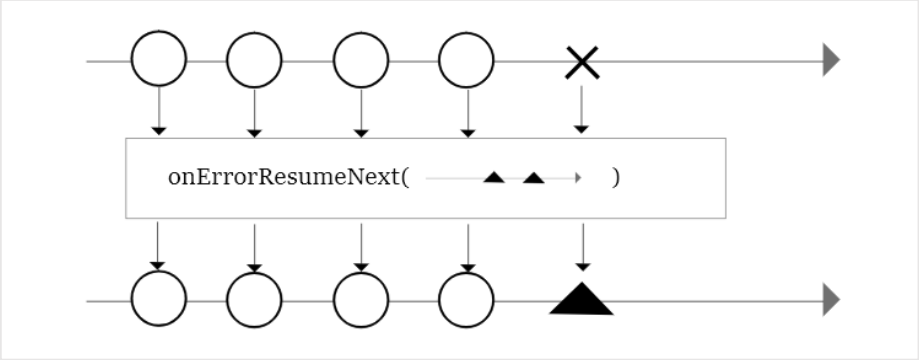
```
val person = PublishSubject.create<String>()
val food = PublishSubject.create<String>()

Observables.zip(person, food) { person, food ->
    "$person ordered $food"
}.subscribe { it: String!
    println(it)
}
person.onNext( t: "Iva")
food.onNext( t: "french fries")
person.onNext( t: "Ivo")
food.onNext( t: "pizza")
person.onNext( t: "Pero")
person.onNext( t: "Tea")
food.onNext( t: "hamburger")
food.onNext( t: "cheeseburger")
person.onNext( t: "Marko")

/* prints
  Iva ordered french fries
  Ivo ordered pizza
  Pero ordered hamburger
  Tea ordered cheeseburger
*/
```

Slika 3.20. *Primjer – zip operator*

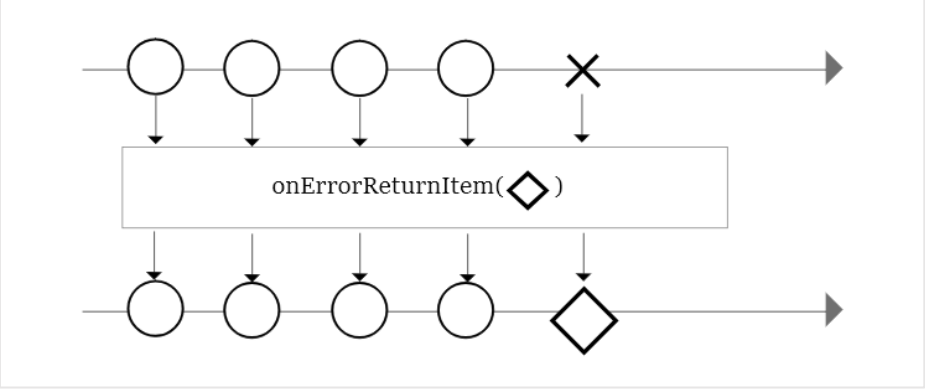
Operatori za rukovanje greškama

S većim i kompleksnijim aplikacijama pojavljivanje grešaka je neizbježno. Kako bi korisnici bili što zadovoljniji potrebno je situacije u kojima se mogu pojaviti greške predvidjeti i staviti pod kontrolu. Neke od najučestalijih grešaka u aplikaciji su gubitak internetske konekcije, pogrešan unos te greške prilikom komunikacije s mrežom, odnosno HTTP greške [3]. Rukovanje s pogreškama u Rx okviru je vrlo jednostavno jer promotrivi tok može emitirati i događaj o pogrešci. Osiguravanje nesmetanog rada aplikacije pomoću Rx okvira može teći u dva smjera. Prvi je da se prilikom pojave pogreške vrati neka zadana vrijednost (engl. *default value*) dok je drugi da se tok ponovi određeni ili neodređeni broj puta. Hvatanje pogrešaka u Rx slično je načinu pokušaj-uhvati (engl. *try-catch*). Dva glavna operatora za hvatanje pogrešaka su *onErrorResumeNext* i *onErrorReturnItem*. *OnErrorResumeNe*xt će presresti grešku i umjesto nje emitirati novi unaprijed određeni tok dok će *OnErrorReturnItem* u stari tok emitirati novi, unaprijed zadani, element.

Za ponavljanje toka u slučaju pogreške koristi se *retry* operator koji dolazi u tri verzije. Osnovni *retry* operator će forsirati ponavljanje toka sve dok on ne uspije. Ako je unutar toka poslan mrežni upit i on vrati grešku, korištenjem *retry* operatora taj upit će se iznova i iznova ponavljati. To predstavlja problem jer postoji mogućnost da se upit šalje za nečim što ne postoji što bi dovelo do stvaranja beskonačnog kruga. Zato postoji verzija *retry* operatora u kojoj se kao parametar prima broj koliko ponavljanja treba obaviti. Dobra je praksa kombinirati takav *retry* operator s nekim od *onError* operatorima, odnosno da se prilikom pojave pogreške nekoliko puta ponovni tok, a ako to ne uspije da se vrati zadana vrijednost. Postoji još i *retryWhen* operator koji se u praksi najčešće koristi. On sadrži funkciju koja vraća promotrivi tok i tek kad taj tok emitira vrijednost dolazi do ponavljanja prvotnog toka. Tok kojeg vraća funkcija zapravo glumi okidač na koji se događa ponavljanje. Često se zna miješati *retry* operator s *repeat* operatorom, *Repeat* operator ponavlja tok samo kada je emitiran događaj o uspješnoj završenosti toka za razliku od retry operatora koji to radi samo kada je emitirana pogreška.



Slika 3.21. *Kliker dijagram - onErrorResumeNext*



Slika 3.22. *Kliker dijagram - onErrorReturnItem*

3.3. Raspoređivači

Raspoređivači (engl. *Schedulers)* daju mogućnost određivanja na kojoj niti će ulančani operatori djelovati na promotrivi tok. Ako se nit točno ne specificira uzima se zadana nit, a to je ona na kojoj se vrši pretplata [4]. Dva operatora koji omogućuju izmjenu niti su *subscribeOn* i *observeOn*. *SubscribeOn* prima kao parametar raspoređivač koji ovisno o metodi kojom je stvoren raspoređuje posao na određenoj niti. *ObserveOn* također kao parametar prima raspoređivač, no za razliku od *subscribeOn* operatora, taj raspoređivač određuje nit na kojoj će promotrivi tok obavještavati svoje pretplatnike o promjenama. Unutar Rx klase *Schedulers* postoji nekoliko tipova raspoređivača koje je moguće kreirati Metodom tvornica [2]:

- *Schedulers.newThread()* – vraća raspoređivač koji za svaki dio posla nekog operatora kreira novu nit
- *Schedulers.from(Executor)* – vraća specifiranog izvršitelja (engl. Executor) kao raspoređivač
- *Schedulers.single()* – vraća raspoređivač koji podržava samo jednu nit na kojoj će se posao slijedno obavljati
- *Schedulers.io()* – vraća raspoređivač koji je vezan za ulazno / izlazne poslove kao što su mrežni pozivi ili komunikacija s bazom podataka
- *Schedulers.trampoline()* – vraća raspoređivač koji raspoređuje posao na trenutnoj niti tako da se svaka dio posla odrađuje nakon što je prethodni dovršen
- *Schedulers.computation()* – vraća raspoređivač koji je namijenjen za obavljanje računskih zadataka ili zahtjevnijih procesorskih poslova

Unutar Android aplikacija najčešće se koristi *Schedulers.io()* kako bi se rasteretila glavna nit prilikom komunikacije s mrežom, bazom ili datotečnim sustavom. Glavna nit u Androidu sustavu je *AndroidSchedulers.mainThread()* koju je moguće dobiti preko Android raspoređivača koji se nalazi unutar rxAndroid biblioteke. Kod operacija u kojima je na kraju potrebna izmjena korisničkog sučelja, *observeOn* operatoru šalje se Android raspoređivač.

```
fun <T> Observable<T>.addSchedulers(): Observable<T> {
    return this.subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread())
}
```

Slika 3.23. *Primjer funkcije proširenja za dodavanje raspoređivača na promotrivi tok*

4. IMPLEMENTACIJA REAKTIVNOG PROGRAMIRANJA UNUTRA ANDROID APLIKACIJE "NAJBOLJI JA"

U ovom dijelu završnog rada bit će prikazana primjena reaktivnog programiranja u konkretnoj Android aplikaciji. Aplikacija „Najbolji Ja" namijenjena je za mentorski program koji provodi *Centar za nestalu i zlostavljanu djecu Osijek,* a osmišljena je kao radna bilježnica sa zadacima kroz koju će dijete uz pomoć mentora na interaktivan i zanimljiv način razvijati socijalne vještine. Prema definiciji svjetske zdravstvene organizacije, socijalne vještine predstavljaju sposobnosti prilagođavanja i pozitivnog ponašanja koja omogućuju osobama da se uspješno nose sa zahtjevima i izazovima svakodnevice [13]. One nisu urođene već naučene i zato ih je bitno razvijati pogotovo kod rizičnih skupina djece i mladih. Aplikacija je trenutno u početnoj verziji koja se sastoji od tri izazova.

4.1. Specifikacija zahtjeva

Ova aplikacija treba bilježiti rad mentora i djeteta. Kako bi voditelji projekta mogli pratiti njihov napredak u izvršavanju izazova potrebno je sve podatke spremati na vanjski poslužitelj. Također je nužno omogućiti nekakav oblik prijave zbog toga što svaki mentorski par zasebno rješava zadatke. Aplikacija treba moći posluživati oko 30-ak mentorskih parova s tendencijom daljnjeg rasta. Zadaci u aplikaciji moraju biti odvojeni i jasno definirani te na zabavan način poticati interakciju između mentora i djeteta. Na kraju svakog riješenog izazova treba se pojaviti obavijest o tome u obliku pohvale. Aplikacija je namijenjena isključivo za online rad te je potrebno dati do znanja informaciju u slučaju gubitka internetske veze.

Tablica 4.1. *Prikaz zahtjeva zasebno za svaki izazov*

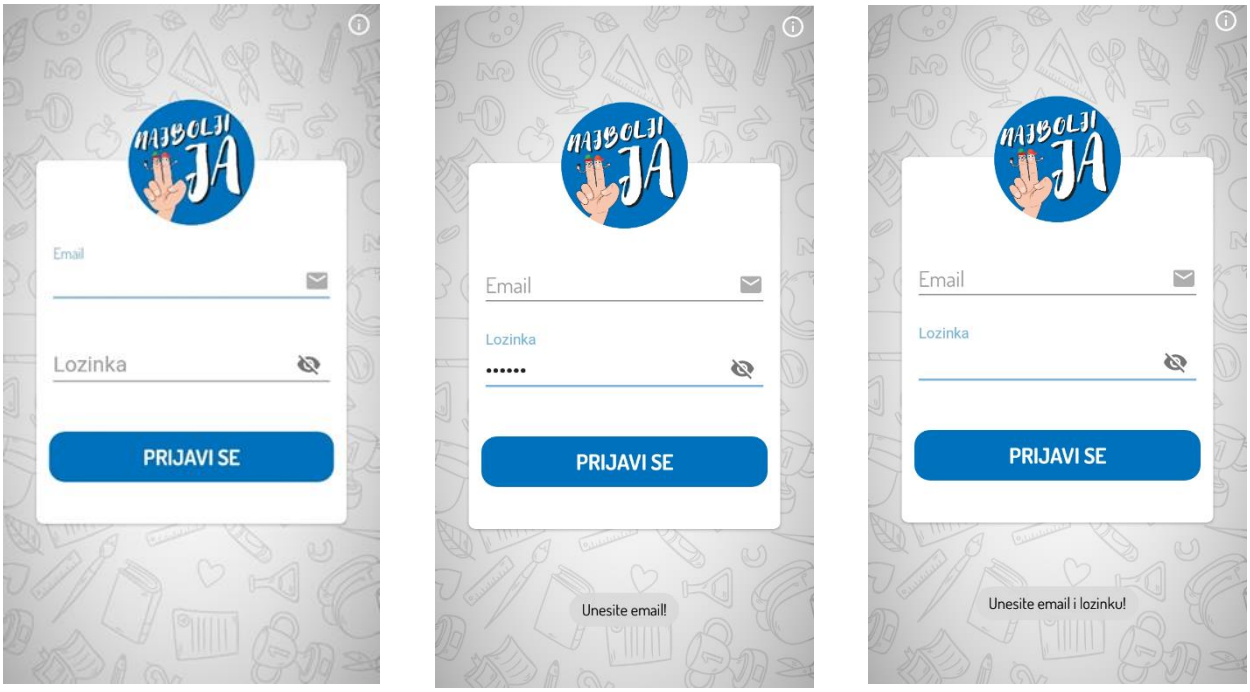
1. izazov – *Bolje se upoznajmo*	- ovaj izazov treba služiti za „probijanje leda" prilikom upoznavanja djeteta i mentora - u njemu je potrebno omogućiti zasebno upisivanje odgovora i djeteta i mentora na 14 općenitih pitanja o njim kao što su najdraža hrana, sport, nadimak itd. - odgovore je potrebno pohraniti u bazu na poslužitelju

	- kada su i dijete i mentor dali odgovore na sva pitanja, prikazuje im se obavijest o završenosti zadatka te im se treba otključati kviz preko kojeg mogu provjeriti koliko znaju jedno o drugome - u kviz je potrebno bilježiti točne i netočne odgovore te ograničiti vrijeme za davanje odgovora
2. izazov – *Foto izazov*	- ovaj izazov treba služiti za produbljivanje suradnje između djeteta i mentora, također treba poticati kreativnost na zabavan način - u njemu je potrebno omogućiti spremanje fotografija različitih situacija u kojima su se dijete i mentor morali zajedno uslikati kao na primjer sa žutim automobilom - svaki sljedeći zadatak treba se prikazati tek kada je prethodni zadatak obavljen - za spremanje fotografije u bazu trebaju postojati dvije opcije: slikanje te odabir slike iz albuma - obavijest o završenosti izazova treba se prikazati tek kada su svi zadaci obavljeni
3. izazov – *Pogled naopačke*	- ovaj izazov treba služiti kao poticaj za promišljanje i razgovor o važnosti zauzimanja svog stava ali isto tako i uvažavanja tuđeg - on se treba sastojati od tri dijela: slika, pitanja i rasprave, koji se istim redom otključavaju - kod slika potrebno je prikazati 4 optičke iluzije te postaviti pitanje što je na slici, nakon što i dijete i mentor unesu odgovore, svako onako kako vidi, treba im omogućiti vidjeti „točan“ odgovor - kod pitanja trebaju biti postavljena dva pitanja oko kojih dijete i mentor trebaju porazgovarati

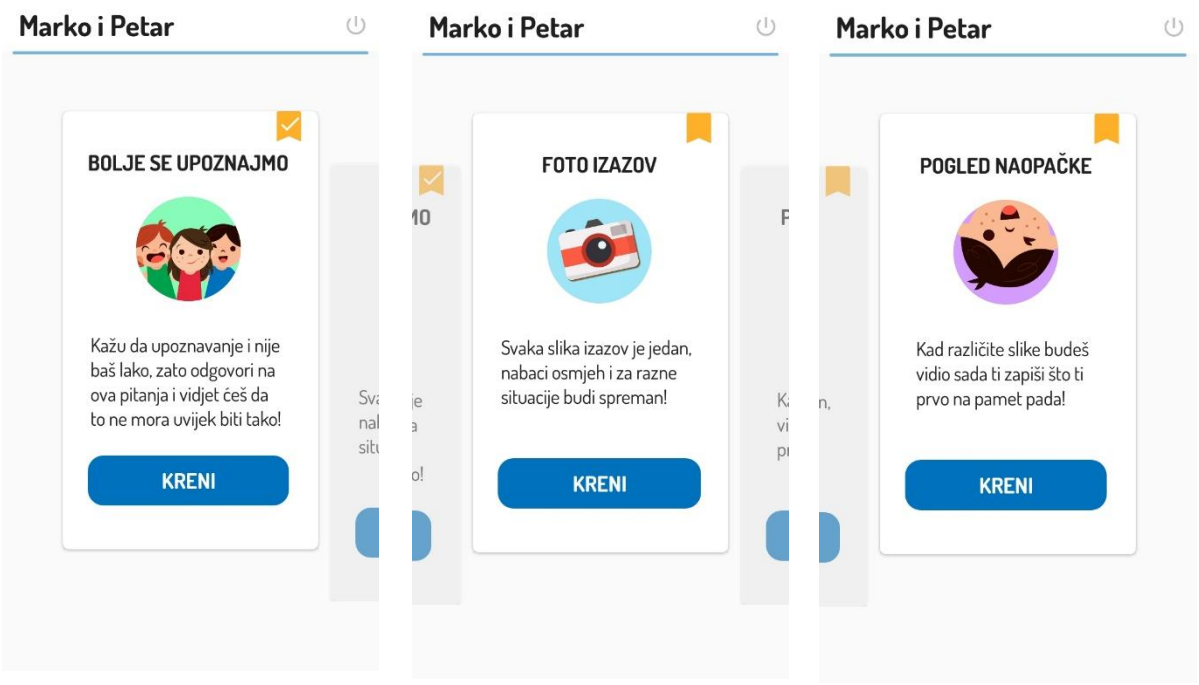
	- u raspravi treba biti omogućeno unošenje za i protiv argumenata za nekoliko različitih situacije kao što je na primjer „Trebaju li u školama učenici nositi jednake uniforme? Zašto da ili zašto ne“, mentor na pitanja odgovara sa DA, a dijete s NE pa zatim obrazlažu svoje odgovore iako se možda s time zapravo ne slažu - sve odgovore i argumente potrebno je pohraniti u bazu

4.2. Postupak korištenja aplikacije „Najbolji JA“

Aplikacija se pokreće klikom na njezinu ikonu koja otvara zaslon za prijavu. U slučaju da je mentorski par već prijavljen otvara se početni zaslon.

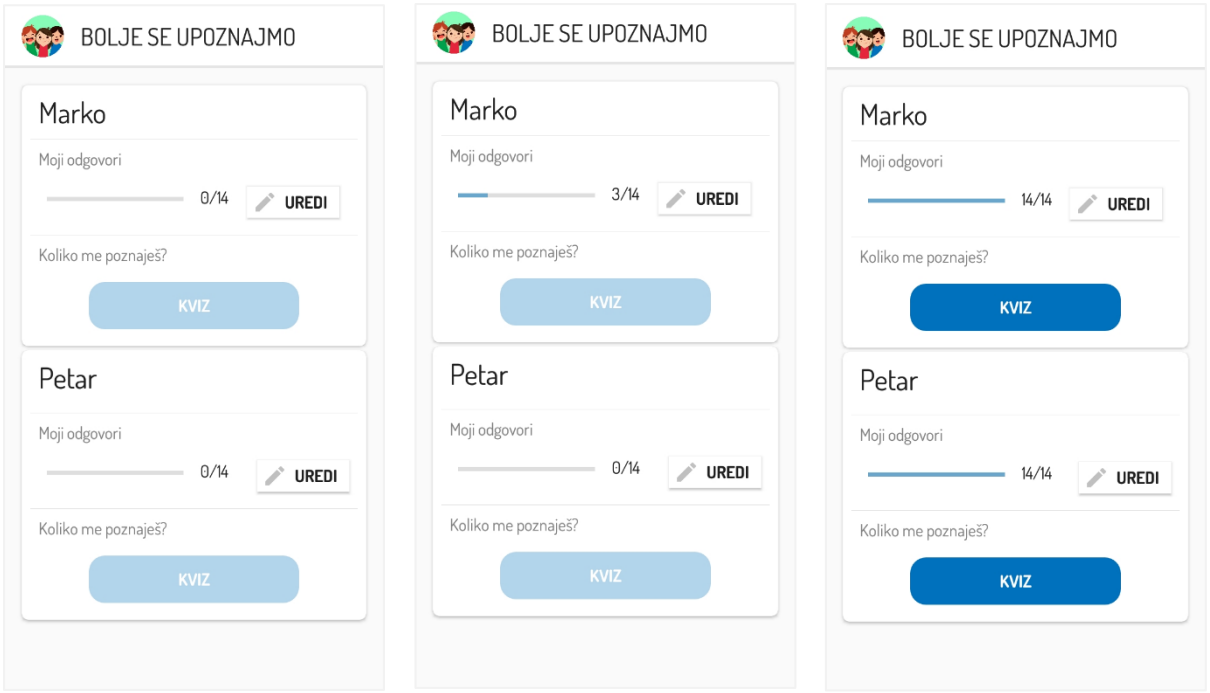


Slika 4.2. *Zaslon za prijavu mentorskog para*



Slika 4.3. *Prikaz izazova na početnom zaslonu aplikacije*

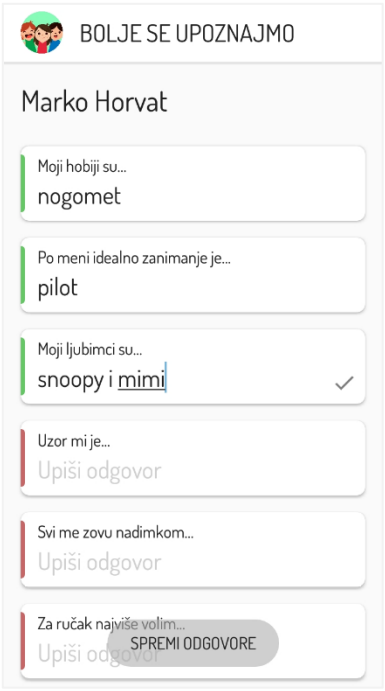
Početni zaslon prikazuje kratki opis svakog izazova, daje uvid u njihovu završenost te omogućava otvaranje izazova. Na njemu se također nalazi ime djeteta i mentora te gumb za odjavu koji vraća mentorski par nazad na zaslon za prijavu. Nakon pritiska na gumb „kreni“ otvara se izabrani izazov.



Slika 4.4. *Početni zaslon izazova „Bolje se upoznajmo“*

Na početnom zaslonu izazova „Bolje se upoznajmo“ nalaze se dvije kartice, jedna za dijete, a druga za mentora Na tim karticama se prati na koliko pitanja je dan odgovor. Klikom na gumb „uredi“ otvara se zaslon sa pitanjima.

Zaslon sa pitanjima prikazan je na slici 4.5. Preko njega je moguće upisivati, izmjenjivati i spremati odgovore na pitanja. Nakon što su i dijete i mentor dali odgovore na sva pitanja prikazuje im se zaslon sa slike 4.6, te im je otključan kviz. Klikom na gumb „kviz“ otvara se zaslon sa slike 4.7 na kojem se nalaze upute za kviz. Klikom na gumb „započni kviz“ otvara se kviz. U kvizu se prikazuju pitanja te odbrojavač preostalog vremena. Kroz gumbove „točan“ i „netočan“ moguće je bilježiti odgovore. Ako vrijeme za pitanje istekne, a nije zabilježen ni točan ni netočan odgovor, odgovor se bilježi kao netočan i prebacuje se na sljedeće pitanje. Nakon završetka kviza prikazuje se zaslon sa rezultatima (slika 4.7).



Slika 4.5. *Zaslon za upisivanje i spremanje odgovora*

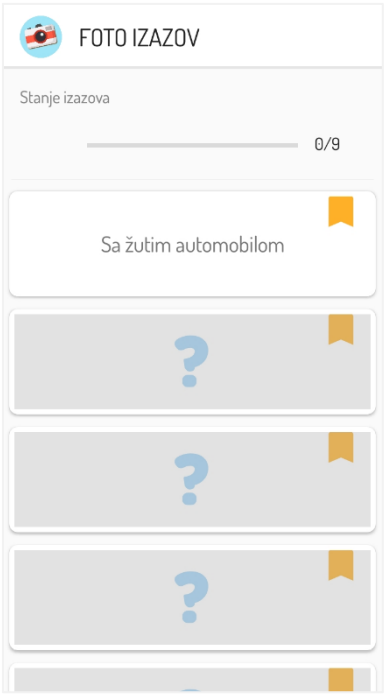
Slika 4.6. *Zaslon koji se prikaže nakon što je izazov riješen*

Na početnom zaslonu izazova „Foto izazov“ (slika 4.10) prikazane su kartice na kojima su zadaci za fotografiranje te linija za praćenje napretka izazova. Svaki idući zadatak prikazuje se tek kada je prethodni riješen. Slika se može učitati iz albuma ili uslikati. Nakon što je slika spremljena moguće ju je uvijek vidjeti klikom na zadatak. Kada su ispunjeni svi zadaci, odnosno spremljene sve slike prikazuje se zaslon sa slike 4.11.

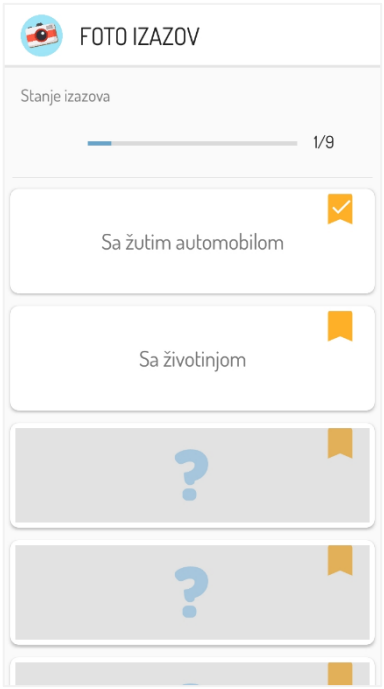
Slika 4.7. *Zaslon s uputama*

Slika 4.8. *Zaslon kviza*

Slika 4.9. *Zaslon s rezultatom kviza*



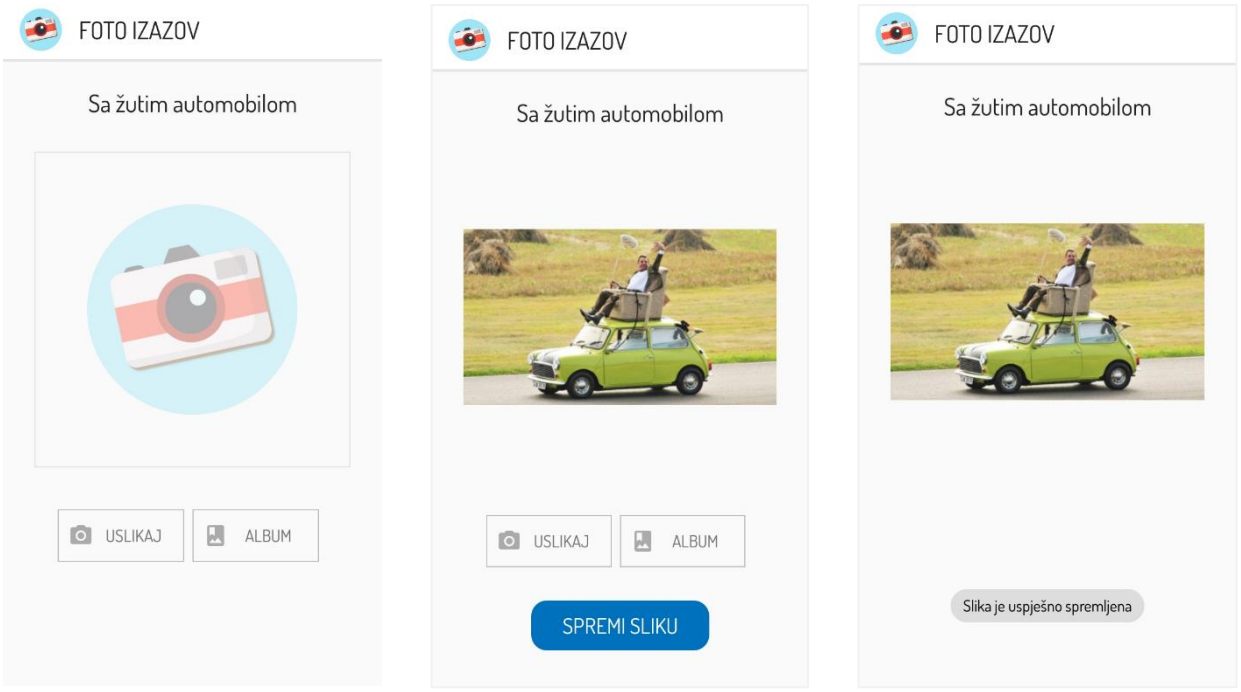
a)



b)

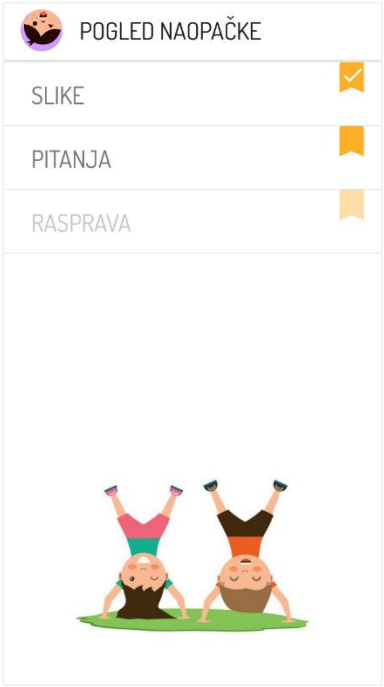
Slika 4.10. *Zaslon izazova „Foto izazov“, a) Početno stanje, b) Izvršen jedan zadatak*

Slika 4.11. *Zaslon koji se prikaže nakon što je izazov riješen*



a) *b)* *c)*

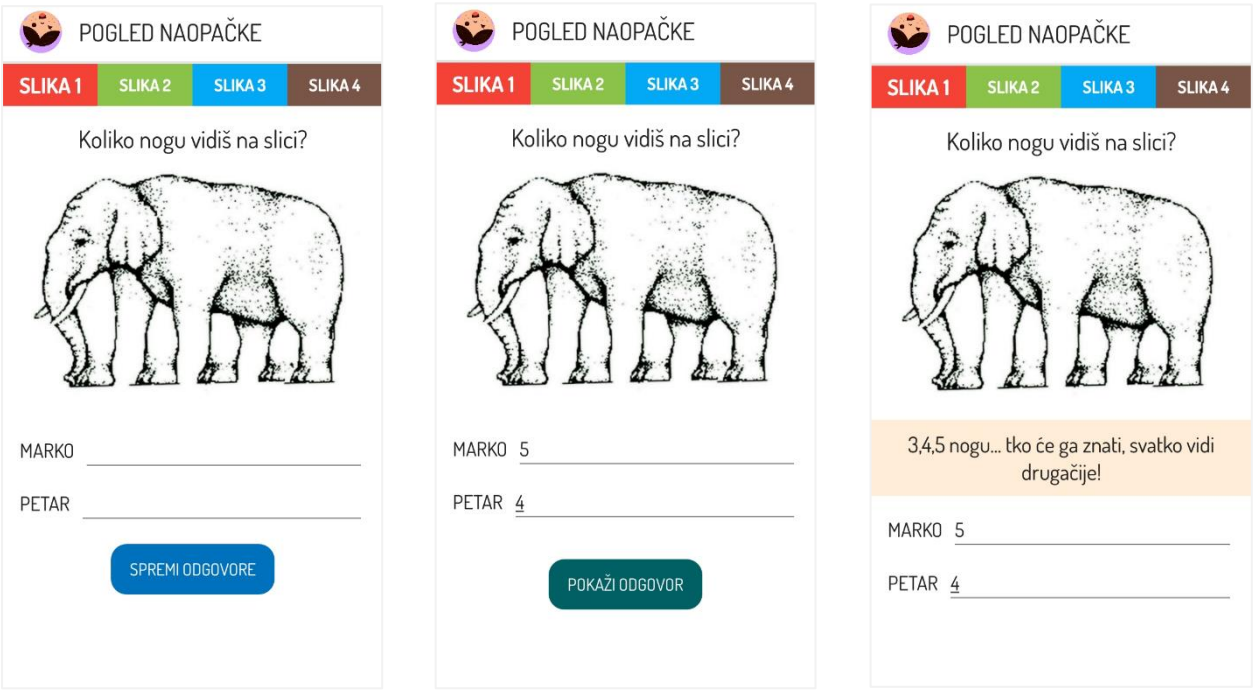
***Slika 4.12**. Zaslon za spremanje slike izazova, a) Početno stanje, b) Nakon što je slika izabrana, b) Nakon što je slika spremljena*



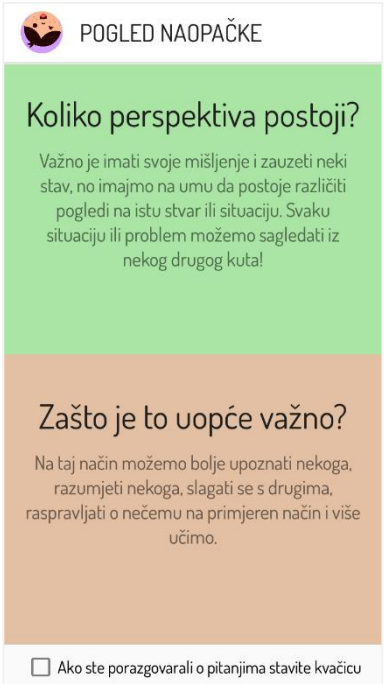
***Slika 4.13.** Početni zaslon izazova „Pogled naopačke“*

***Slika 4.14.** Zaslon koji se prikaže nakon što je izazov riješen*

Na početnom zaslonu izazova „Pogled naopačke“ nalaze se tri zadatka: slike, pitanja i rasprava, koja se tim redoslijedom i otključavaju. U zadatku „slika“ prikazane su 4 optičke iluzije za koje dijete i mentor upisuju što vide na njima. Nakon što spreme odgovore mogu prikazati „točan“ odgovor.



Slika 4.15. *Zaslon zadatka „slika“*



Slika 4.16. *Zaslon zadatka „pitanja“*

Slika 4.17. *Zaslon zadatka „rasprava“*

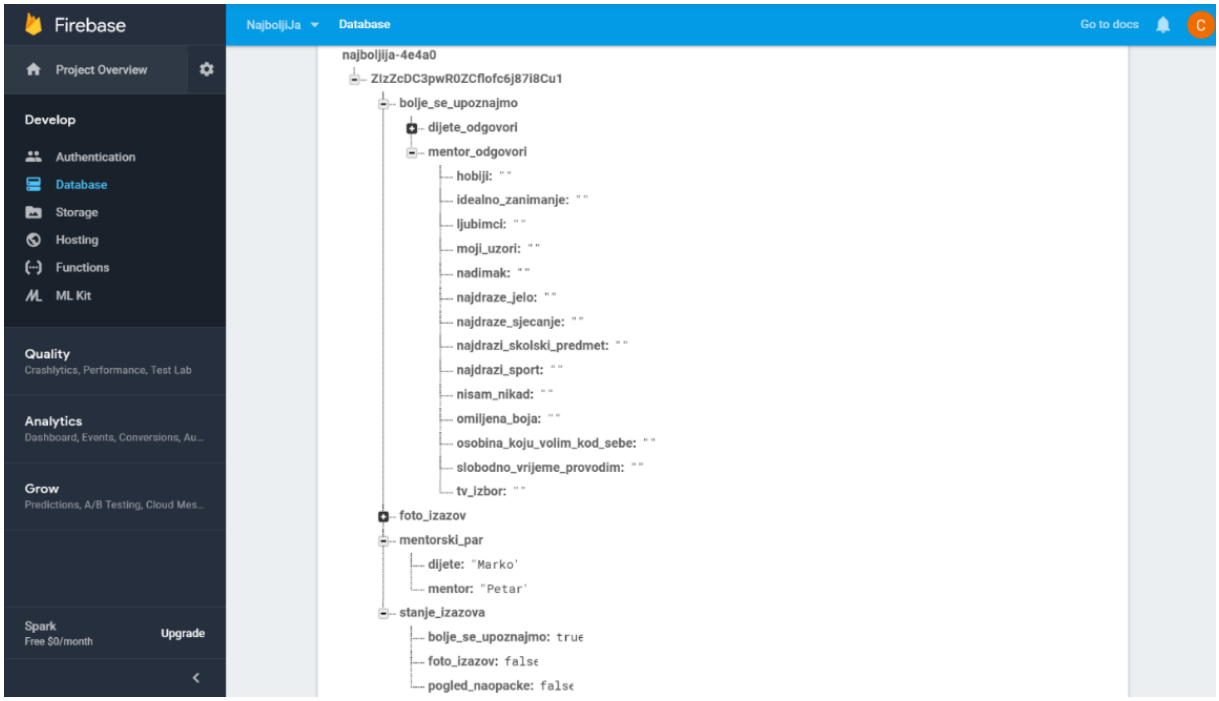
4.3. Opis platforme i tehnologije

Za razvoj aplikacije korišteno je službeno integrirano razvojno okruženje (engl. *Integrated development environment,* IDE) za izradu Android aplikacija, Android Studio. On u sebi sadrži osnovne pakete za razvoj koji se često nadopunjuju drugim paketima pomoću sustava za izgradnju (engl. *build systems*) Gradle. Biblioteke treće strane (engl. *third party libraries*) olakšavaju razvoj aplikacija i štede vrijeme programerima jer za neke probleme ne moraju „izmišljati toplu vodu“ nego se mogu osloniti na već napisana i istestirana rješenja. Biblioteke koje su uz Rx uključene u ovaj projekt su:

- Koin – za ubrizgavanje ovisnosti prilikom povezivanja sučelja i logike,
- Retrofit – za povezivanje s Firebaseom,
- Epoxy – kao sučelje za rad s elementom pogleda RecyclerView,
- Glide – za rad sa slikama,
- ViewModel – za slaganje MVVM arhitekturnog obrasca.

Cijela aplikacija pisana je u programskom jeziku Kotlin. Kotlin sadrži obilježja objektno orijentirane i funkcionalne paradigme u isto vrijeme. Vrlo je sažet i jednostavan, te optimizira broj linija koda i do 40% u odnosu na Javu [11]. Neke novine koje je Kotlin donio sa sobom su: ništavni tipovi (engl. *nullable types*), koji odrađuju odličan posao kada su u pitanju pogreške vezane uz ne-instancirane objekte (engl. *NullPointerException*), funkcije višeg reda (engl. *higher-order functions*) te funkcije proširenja (engl. *extension functions*). Zadnja objavljena verzija Kotlina je 1.3.41.

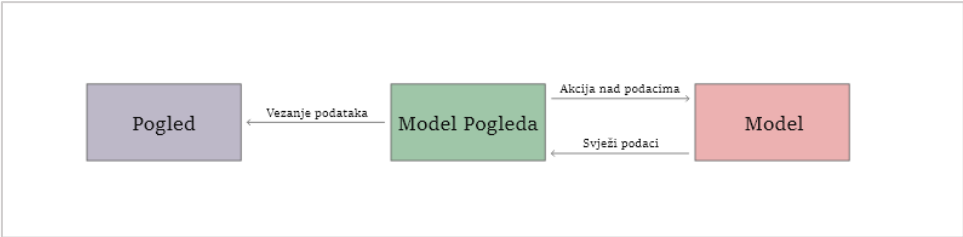
Za svrhu spremanja podataka korišten je Firebase, Googleova platforma koja uz bazu podataka u stvarnom vremenu pruža i mnoge druge korisne alate za rad s podacima kao što su provjera autentičnosti, *Crashlytics* (alat koji prati padove aplikacije te o tome šalje obavijest u stvarnom vremenu), podrška za strojno učenje, slanje obavijesti itd. Firebaseova baza podataka u stvarnom vremenu je NoSQL baza podataka koja sve podatke sprema u JSON formatu što doprinosi čitljivosti baze. Autentifikacija u aplikaciji je za sada napravljena sa email adresom i lozinkom koje će biti dane svakom mentorskom paru.



Slika 4.18. *Izgled baze podataka za jedan mentorski par*

4.4. Opis rješenja

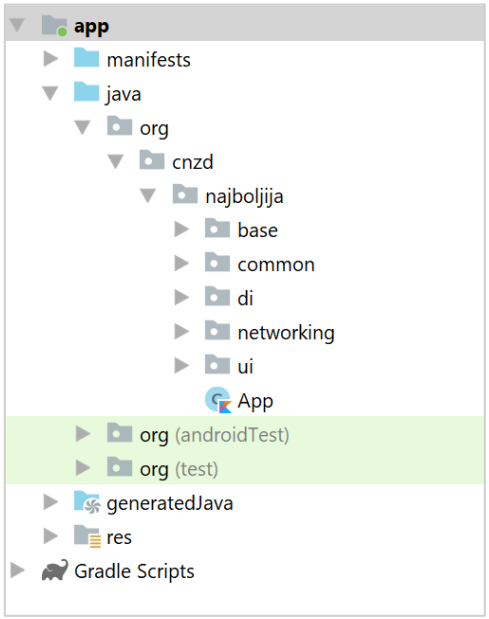
Kako bi se pratili principi oblikovnih obrazaca i čistog koda prilikom razvoja aplikacije kod vezan za poslovnu logiku odvaja se od koda vezanog za elemente na korisničkom sučelju. Dva najčešće korištena arhitekturna obrasca kojima se to postiže su: Model-Pogled-Prezenter (engl. *Model-View-Presenter*, MVP) i Model-Pogled-Model Pogleda (engl. *Model-View-ViewModel*, MVVM) . Sloj modela služi za mrežno povezivanje i komunikaciju s bazom. Sloj pogleda najbliži je korisniku jer je zadužen samo za prikaz podataka na sučelju te prihvaćanje i prosljeđivanje korisničkih interakcija. Sloj koji povezuje ta dva sloja sadrži svu poslovnu logiku. U MVP obrascu to je prezenter koji ima obostranu komunikaciju sa slojem pogleda. Kod MVVM obrasca posrednički sloj je model pogleda koji uspostavlja jednosmjernu komunikaciju sa slojem pogleda. Unutar ove aplikacije implementiran je MVVM obrazac zbog svog načina rada (u kojem sloj pogleda osluškuje promjene na sloju modela pogleda) koji predstavlja savršenu podlogu za reaktivno programiranje.



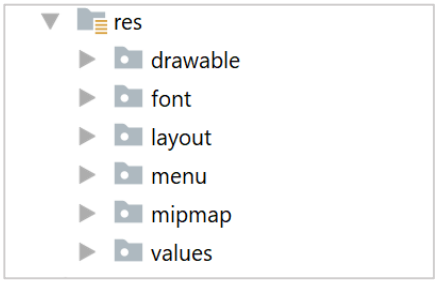
Slika 4.19. *Izgled toka podataka unutar MVVM obrasca*

Svaki Android studio projekt sastoji se od jednog ili više modula i postavkama sustava za izgradnju za taj specifični modul. Unutar modula nalaze se datoteke sa izvornim kodom. U zadnje vrijeme česta je praksa razdvajati različite funkcionalnosti projekta na više modula pošto se svaki modul može zasebno graditi, testirati i biti podvrgnut postupku otklanjanja pogrešaka. Ovaj se projekt sastoji od jednog modula, app modula, koji je Android studio sam generirao prilikom stvaranja projekata.

U app modulu (Slika 4.20) nalaze se paketi *manifest*, *java* i *res*. *Manifest* sadrži datoteku AndroidManifest.xml koja pruža esencijalne informacije o aplikaciji i njezinim komponentama kao što su aktivnosti, servisi i davatelji sadržaja. Unutar *java* paketa nalazi se izvorni kod aplikacije, dok se u *res* paketu nalazi sve ostalo što nije bazirano na kodu. *Res* paket podijeljen je na potpakete koji sadrže resurse u različitim oblicima (slika 4.21). U *layout* paketu nalaze se pogledi pisani jezikom za označavanje podataka (engl. *EXtensible Markup Language, XML*). *Drawable* paket sadrži slike, vektore, ikone te oblike i pozadine koji se koriste za neke elemente pogleda. Dimenzije, boje, stilovi i znakovni nizovi korišteni u aplikaciji smješteni su u paketu *values*.



Slika 4.20. *Izgled strukture paketa projekta*



Slika 4.21. *Potpaketi res paketa*

Izvorni kod podijeljen je na pakete *base, common, networking, di* i *ui*. U base paketu odvojene su klase zajedno sa funkcionalnostima koje su zajedničke svim klasama nekog tipa. Tako se npr. u tom paketu nalazi klasa *BaseFragment* koji sadrži logiku za prikazivanje učitavanja podataka ili pojave pogreške. Svi ostali fragmenti nasljeđuju *BaseFragmet* jer im je potrebna ta logika. Na ovaj način se izbjegava dupliciranje koda i olakšava dodavanje promjena koje trebaju vrijediti za sve fragmente. U *common* paketu izdvojene su konstante i pomoćne klase za rad s podacima koje se koriste u cijeloj aplikaciji. U di paketu su definirani koin moduli preko kojih se ubrizgava ovisnost između klasa. *Networking* paket sadrži sve potrebno za komunikaciju s mrežom. Unutar paketa ui zaslon za prijavu, početni zaslon te zasloni svakog izazova odvojeni su u zasebne pakete zajedno sa pripadajućim modelima pogleda. Aplikacija je napravljena na način da svaki paket unutar ui paketa sadrži po jednu aktivnost unutar koje se po potrebi izmjenjuju fragmenti. Na samom kraju vidljiva je još i App klasa (slika 4.22). Ona nasljeđuje Androidovu *Application* klasu te predstavlja osnovnu klasu za upravljanje globalnim stanjima aplikacije. Pošto se instancira prije nego sve ostale klase, u njoj se pokreće koin biblioteka navodeći sve potrebne module.

```
class App : Application() {

    override fun onCreate() {
        super.onCreate()

        startKoin { this: KoinApplication
            androidContext( androidContext: this@App)
            modules(
                    listOf(
                            networkinModule,
                            sharedPrefsModule,
                            viewModels,
                            serviceModule,
                            commonModule)
            )
        }
    }
}
```

Slika 4.22. *App klasa u kojoj se pokreće ubrizgavanje ovisnosti*

4.5. Važni implementacijski detalji

Reaktivno programiranje u aplikacijama najčešće primjenu pronalazi prilikom komunikacije s mrežom ili bazom. Iako Firebase pruža svoje API sučelje za rad s podacima iz baze, u ovom projektu je za potrebe autentifikacije, čitanja i pisanja podataka korišteno REST (engl. *Representational state transfer*) API sučelje s Retrofitom kao klijentom. Retrofit ima ugrađenu podršku za Rx koja omogućava omotavanje mrežnog odgovora u promotrivi tok. Ta podrška uključena je prilikom instanciranja Retrofita dodavanjem RxJava2CallAdapterFactory unutar graditelj metode *addCallAdapterFactory* (slika 4.23).

```
//RETROFIT
single<Retrofit> { this: Scope
    Retrofit.Builder()
            .baseUrl(BASE_URL)
            .client(get())
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
            .build()
}
```

Slika 4.23. *Instanciranje Retrofita unutar Koin modula*

Prilikom pokretanja aplikacije prikazuje se zaslon za prijavu mentorskog para (slika 4.2). Postupak prijave započinje klikom na gumb za prijavu čiji se osluškivač (engl. *listener*) nalazi unutar LoginFragmenta. Ako je neko polje ostalo prazno prilikom stiska gumba za prijavu automatski se izbacuje poruka da se polje treba popuniti. Također prije nego što se preko mreže pošalje zahtjev za prijavom provjeri se stanje internetske konekcije i u slučaju da je nema prikazuje se poruka o tome. Ako su svi uvjeti ispunjeni na modelu pogleda LoginVM poziva se funkcija login (slika 4.25) čiji je povratni tip promotrivi tok koji emitira LoginResponse. Unutar te metode se i na Interactoru poziva login metoda sa parametrima za web *api key* i podacima za prijavu. Interactor je posrednik između modela pogleda i REST sučelja. U njemu su kroz funkciju proširenja *addScheduler*s dodani raspoređivači na svaki promotrivi tok dobiven mrežnim odgovorom.

```
fun login(): Observable<LoginResponse> {
    loginLiveData.value = LoginRequest(email, password)
    return interactor.login(WEB_API_KEY,loginLiveData.value!!)
}
```

Slika 4.25. *Login metoda koja se nalazi u LoginVM*

Jedan promotrivi tok se provlači od REST sučelja preko Interactora i LoginVM sve do LoginFragmenta. Ako zahtjev za autentifikacijom uspješno prođe bit će emitiran LoginResponse i kod koji se nalazi u *doOnNext* bit će izvršen (slika 4.24). Login response u sebi sadrži osnovne informacije o korisniku te localId i idToken. Ti identifikatori su spremljeni u dijeljene postavke (engl. *SharedPreferences*) kako bi se mogli koristiti i kod drugih API poziva unutar aplikacije. IdToken traje 60 minuta te se nakon njegovog isteka ponovno prikazuje zaslon za prijavu.

```
if (hasInternet(context)) {
    viewModel.login()
            .doOnNext { it: LoginResponse!
                sharedPrefs.putBoolean(KEY_IS_LOGGED_IN, true)
                sharedPrefs.putString(KEY_TOKEN_ID, it.idToken)
                sharedPrefs.putString(KEY_LOCAL_ID, it.localId)
                val intent = Intent(activity!!.baseContext, HomeActivity::class.java)
                intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
                startActivity(intent) }
            .onErrorReturn { it: Throwable
                context!!.showToast(getString(org.cnzd.najboljija.R.string.error_occured))
                ^onErrorReturn LoginResponse()
            }
            .longSubscribe()
} else {
    context!!.showToast(getString(org.cnzd.najboljija.R.string.no_internet_connection))
}
```

Slika 4.24. *Dio koda unutar LoginFragmenta koji sadrži reakcije na emitiranje događaja promotrivog toka kojeg vraća login()*

```
class HomeVM(private val interactor: Interactor) : BaseViewModel() {

    val namesData = MutableLiveData<String>()
    val challengesState = MutableLiveData<Map<String, Boolean>>()

    var tokenId:String? = null
    var localId: String? = null

    init {
        namesData.value = null
        challengesState.value = null
    }

    fun getHomeData(): Observable<Any> = Observable.concat(getNamesData(), getChallengesStateData())

    fun getNamesData() : Observable<String!>! = interactor.getNames(localId!!, tokenId!!)
            .map { it.getValue( key: "dijete") + " i " + it.getValue( key: "mentor") }
            .doOnNext { namesData.value = it }

    fun getChallengesStateData() : Observable<Map<String, Boolean>!>! = interactor.getChallengesState(localId!!, tokenId!!)
            .doOnNext { challengesState.value = it }
}
```

Slika 4.26. *Prikaz HomeVM*

```
@POST( value: "https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword")
fun logIn(@Query( value: "key") webApiKey: String, @Body loginRequest: LoginRequest): Observable<LoginResponse>
```

Slika 4.27. *Prikaz REST metode za prijavu čiji je odgovor umotan u promotrivi tok koji emitira LoginResponse*

Nakon uspješne prijave prikazuje se početni zaslon (engl. home screen) (slika 4.3). Pogled odgovoran za taj zaslon je HomeActivity. On se sastoji od elementa pogleda ViewPager koji prilikom klizanja po ekranu prikazuje fragmente izazova. Za početni zaslon iz baze se povlače podaci o imenu djeteta i mentora te o trenutom stanju riješenosti svakog izazova. HomeVM podatke koji trebaju biti prikazani na zaslonu drži unutar LiveData (slika 4.26), klasi koja zajedno s klasom ViewModelom pripada komponentama Android Arhitekture (engl. *Android Architecture Components*). LiveData služi za obavještavanje pogleda o promjeni podataka koji trebaju biti prikazani na zaslonu. HomeActivity preko funkcije observe postaje promatrač promjena unutar LiveData te kada do njih dođe osvježi zaslon s novim podacima. Vrijednost koju LiveData drži mijenja se unutar *doOnNext* operatora koji se poziva prilikom uspješnog emitiranja odgovora s mreže.

Prilikom dohvaćanja podataka bilo iz baze ili sa mreže potrebno je uvijek korisniku dati do znanja da se podaci učitavaju. U ovoj aplikaciji za to se koristi kružni učitavač (engl. *loader*). Za prikaz podataka na početnom zaslonu potrebno je napraviti dva API poziva te kružni učitavač skloniti tek kada su oba uspješno završena. Taj problem je riješen pomoću operatora *concat* koji dva promotriva toka spaja u jedan, odnosno izvršava *doOnComplete* tek kada su oba toka uspješno završila. Unutar Pogleda se započinje pretplata na promotrivi tok kojeg vraća mreži odgovor. Na taj način se lako kontrolira onoga što će biti prikazano na zaslonu u slučaju učitavanja podataka, pogreške ili prekida internetske veze. Funkcija proširenja *longSubscribe* (slika 4.28), u kojoj se nalazi logika vezana za prethodno spomenute slučajev, smještena je unutar BaseFragmenta kako bi ju svi fragmenti koji nasljeđuju BaseFragment mogli koristiti.

```
protected fun <T> Observable<T>.longSubscribe() {
    val observable =
            this.doOnSubscribe { loaderLayout?.toVisible() }
                    .doOnError { loaderLayout?.toInvisible(); handleError(it) }
                    .doOnComplete { loaderLayout?.toInvisible(); noInternetLayout?.toInvisible() }
                    .doOnNext { loaderLayout?.toInvisible(); noInternetLayout?.toInvisible() }
                    .retryWhen { it: Observable<Throwable!>
                        it.flatMap { it: Throwable
                            btnRetry?.let { view -> RxView.clicks(view) }
                        }
                    }

    addDisposable(observable.subscribe({}, { it.printStackTrace() }))
}

private fun handleError(throwable: Throwable) {
    when (throwable.message) {
        UNABLE_TO_RESOLVE_HOST -> noInternetLayout?.toVisible()
        HTTP_401 -> logOut()
        else -> context!!.showToast("Dogodila se pogreška.")
    }
}
```

Slika 4.28. *Prikaz funkcije proširenja longSubscribe*

U trenutku pretplate prikazuje se kružni učitavač koji ostaje prikazan sve dok nije emitiran podatak mrežnog odgovora, događaj o završenosti ili događaj o pogrešci. U slučaju emitiranja događaja o pogrešci na *doOnError* se objekt tipa *Throwable* šalje funkciji *handleError* koja djeluje s obzirom na tip pogreške. U slučaju prekida internetske veze prikazuje se zaslon sa slike 4.29, dok se u slučaju neovlaštenog pristupa, do kojeg dolazi nakon isteka tokenId-a, prikazuje zaslon za prijavu. Za rukovanje s pogreškom zbog prekinute internetske veze koristi se *retryWhen* operator unutar kojeg *flatMap* operator tok pogreške transformira u tok pritiska gumba „pokušaj ponovno“ koji je umotan u RxView klik. Klik na taj gumb služi kao okidač da se cijeli tok ponovi pa tako i API poziv.

Na slici 4.30 prikazana je funkcija addDisposable, koja se također nalazi u BaseFragmentu, unutar koje se pretplate spremaju u compositeDisposable. Prekidanje pretplata događa se unutar životne metode (engl. *lifecycle method*) fragmenta na uništenju (engl. onDestroy) kako bi se osiguralo da su sve pretplate prekinute kada više sigurno nisu potrebne.

Slika 4.29. *Zaslon prikazan u slučaju prekida internetske veze*

```
private fun addDisposable(disposable: Disposable) {
    if (compositeDisposable.isDisposed)
        compositeDisposable = CompositeDisposable()
    compositeDisposable.add(disposable)
}

override fun onDestroy() {
    super.onDestroy()
    compositeDisposable.dispose()
}
```

Slika 4.30. *Dodavanje pretplata u compositeDisposable i njihovo prekidanje prilikom uništenja aktivnosti ili fragmenta*

Za implementaciju kviza sa slike 4.8 kreiran je promotrivi tok putem interval operatora. Interval operator emitira beskonačni niz uzlaznih brojeva sa zadanim vremenskim razmakom, u ovom slučaju to je 1 sekunda. Za njim se ulančavaju funkcija proširenja *addSchedulers* te *take* i *map* operatori. *Take* operator prima broj 9 jer je potrebno odbrojavati od 8 do 0. *Map* operator transformira

emitirane brojeve u silaznu sekvencu. Na *doOnNext* svaki emitirani broj se ispiše na zaslon. Ako odbrojavanje završi bez prekida poziva se *doOnComplete* u kojem se automatski sprema odgovor kao netočan te se provjerava je li prikazano zadnje pitanje i ako je završava se kviz. Ulančani su još doOnDispose i *repeat* operatori. *DoOnDispose* se poziva prilikom pritiska na gumbove točno ili netočno i u njemu se povećava redni broj pitanja. Unutar *repeat* operatora određeno je koliko puta se tok treba iznova ponoviti što odgovara sveukupnom broju pitanja.

```
private fun getObservable(): Observable<Long> {
    return Observable.interval( period: 1, TimeUnit.SECONDS)
            .addSchedulers()
            .take( count: 9)
            .map { 7 - it }
            .doOnNext { tvTimer.text = it.toString() }
            .doOnComplete {
                wrong++
                if (questionIndex == 13) {
                    onQuizFinished()
                } else {
                    questionIndex++
                    setupQuestionAndTimer()
                    tvWrong.text = wrong.toString()
                }
            }
            .doOnDispose { questionIndex++ }
            .repeat(repeatTimes)
}
```

Slika 4.31. *Prikaz promotrivog toka koji predstavlja kviz*

5. ZAKLJUČAK

U ovom završnom radu opisani su osnovni koncepti i principi rada reaktivnog programiranja korištenjem ReactiveX okvira. Na primjeru konkretne Android aplikacije prikazana je implementacija Rx komponenti zajedno sa arhitekturnim obrascem MVVM i komponentama Android Arhitekture: LiveData i ViewModel. Najzahtjevniji dio prilikom pisanja ovog rada bilo je razabrati i prikazati najbitnije komponente Rx okvira budući da on obuhvaća velik broj klasa s različitim funkcionalnostima.

Prilikom razvijanja aplikacije bilo je potrebno voditi brigu o mogućnosti proširivanja i dodavanja novih izazova u budućnosti. U procesu stvaranja aplikacije najviše vremena utrošeno je na osmišljavanje izgleda dizajna korisničko sučelja i načina na koji će se zasloni povezivati. Aplikacije bi se mogla poboljšati unaprjeđivanjem zaslona korisničkog sučelja da više prate dizajn materiala (engl. *Material Design*) koji sadrži smjernice za gradnju nativne Android Aplikacije. Rad aplikacije testirat će se kroz mentorski program prilikom njezinog doticaja sa stvarnim korisnicima, tj. mentorskim parovima. Također, nakon određenog vremena od početka korištenja aplikacije bit će moguće, uz pomoć anketa, analizirati korisnost aplikacije i zadovoljstvo korisnika te na taj način proširiti i upotpuniti ovaj rad.

Ostvarenje implementacije Rx okvira unutar aplikacije može se postići na mnogo različitih i korisnih načina. U ovom radu opisani su samo neki što otvara prostor za daljnje istraživanje i učenje o primjeni Rx komponenti u stvaranju programske podrške.

LITERATURA

[1] M. Novak, Imperative versus declarative code… what's the difference?, Medium, 2018., dostupno na: https://medium.com/front-end-weekly/imperative-versus-declarative-code-whats-the-difference-adc7dd6c8380, pristupljeno: 22. lipanj, 2019.

[2] C. Arriola, A. Huang, Reactive Programming on Android with RxJava, Leanpub, 2017.

[3] A.Sulivan, Reactive Programing with Kotlin, Razeware LLC., 2019.

[4] A. Maglie, Reactive Java Programming, Apress, Italy, 2016.

[5] K. Jablonski, Reactive Programming with RxAndroid in Kotlin: An Introduction, RayWenderlich, 2019., dostupno na: https://www.raywenderlich.com/2071847-reactive-programming-with-rxandroid-in-kotlin-an-introduction, pristupljeno: 23.lipanj, 2019.

[6] J. Boner, D. Farley, R. Kuhn, M. Thomposn, The Reactive Manifesto, We Are Reactive, 2014., dostupno na: https://www.reactivemanifesto.org/, pristupljeno: 24.lipanj, 2019.

[7] Calling Java code from Kotlin, Kotlin, dostupno na: https://kotlinlang.org/docs/reference/java-interop.html, pristupljeno: 25.lipanj, 2019.

[8] RxKotlin, GitHub, 2018., dostupno na: https://github.com/ReactiveX/RxKotlin, pristupljeno: 25.lipanj, 2019.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides (Gang of Four), Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, Massachusetts, Sjedinjene Američke Države, 1994.

[10] Introduction: Chaining Operators, ReactiveX, dostupno na: http://reactivex.io/documentation/operators.html, pristupljeno: 26.lipanj, 2019.

[11] FAQ, Kotlin, dostupno na: https://kotlinlang.org/docs/reference/faq.html

[12] Subject, ReactiveX, dostupno na: http://reactivex.io/documentation/subject.html, pristupljeno: 1.rujan, 2019.

[13] Socijalne vještine, Crte-osobnosti, 2015., dostupno na: http://crte-osobnosti.com/socijalne-vjestine/, pristupljeno: 16.lipanj, 2019

SAŽETAK

Cilj ovo završnog rada je dati uvid u reaktivnu paradigmu razvoja programske podrške, objasniti kako i zašto je nastala, opisati njezine gradbene komponente i principe rada te prikazati njezinu konkretnu primjenu u Android aplikaciji kroz okvir RectiveX. U ovom radu reaktivno programiranje stavljeno je u kontekst paradigmi programiranja te su detaljno opisani koncepti koji su postavili njegove temelje. Osim toga prikazana je i struktura Android aplikacije, biblioteke koje su u nju uključene te platforma Firebase koja je korištena za pohranu podataka. Za razvoj projekta korišteno je razvojno okruženje Android studio, a programski jezik u kojem je pisan projekt je Kotlin.

Ključne riječi: Android, MVVM, Oblikovni obrasci, Reaktivne ekstenzije

Reaktivno programiranje

ABSTRACT

The aim of this Bachelor's Thesis is to give an insight into the reactive software development paradigm, to explain how and why it was crated, to describe its building components and working principles as well as to demonstrate its application in the Android application through the RectiveX framework. In this Thesis, reactive programming is put in the context of programming paradigms and the concepts that laid its foundations are described in detail. The structure of the Android application, libraries included in it and Firebase platform used to store the data are presented. For the project development, Android studio development environment was used and the programming language in which the project was written is Kotlin.

Key words: Android, MVVM, Design patterns, Reactive extensions, Reactive programming

ŽIVOTOPIS

Terezija Umiljanović rođena je 26.6.1997. u Našicama, gdje je pohađala Osnovnu Školu Dore Pejačević i Srednju Školu Isidora Kršnjavog. Nakon završene opće gimnazije upisuje preddiplomski studij na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek, smjer računarstvo. Radila je 7 mjeseci kao Anotator podataka u Osječkom startupu Gideon Brothers, a trenutno studira i radi kao Android developer u informatičkoj tvrtci Plava tvornica.

PRILOZI

1. Reaktivno programiranje korištenjem okvira ReactiveX u .docx formatu
2. Reaktivno programiranje korištenjem okvira ReactiveX u .pdf formatu
3. Izvorni kod