**Univerzitet u Beogradu**

**Matematički fakultet**

**Komparativna analiza modernih tehnologija u razvoju Android aplikacija**

Master rad

**Kandidat: Mentor:**

Stefan Bačević Prof. dr Vladimir Filipović

**Uvod**

Android je operativni sistem zasnovan na Linux jezgru, razvijen od strane kompanije Google primarno za mobilne uređaje. Od inicijalnog predstavljanja 2008. godine stekao je ogromnu popularnost i danas je najzastupljeniji mobilni operativni sistem na tržištu. Sa sve većom upotrebom različitih mobilnih uređaja kao i nastojanjem da Android bude operativni sistem koji će pokretati najrazličitije uređaje u okviru interneta stvari (eng. Internet of things) možemo očekivati da se trend rasta udela Androida na tržištu nastavi.

Uprkos svemu tome razvijanje aplikacija za Android je i dalje mlada oblast računarstva koja se razvija veoma brzo. Programeri su “u hodu” otkrivali kako najbolje da primene postojeće šablone u programiranju na novi kontekst i uporedo sa tim su se razvijali alati i biblioteke koji pojednostavljuju i ubrzavaju programiranje Android aplikacija. Vremenom je pređen put od aplikacija čija je sva logika sadržana u aktivnostima aplikacije (eng. Activity) preko korišćenja Model-pogled-kontroler šablona (eng. Model-View-Controller, u daljem tekstu MVC) sve do danas aktuelnih šablona Model-kontroler-prezenter (eng. Model-View-Presenter, u daljem tekstu MVP) i Model-Pogled-Pogled-Model (eng. Model-View-View-Model, u daljem tekstu MVVM). Zbog sve složenijih projekata a prateći SOLID[[1]](#footnote-1) princip inverzije kontrole započeto je korišćenje bilblioteka koje omogućavaju umetanje zavisnosti kao što su RoboGuice i Dagger.

U okviru rada biće kreirana i aplikacija otvorenog koda “Movie Master“ koja će pored pomenutih arhitekturalnih šablona i biblioteka predstaviti još i reaktivni način programiranja i njegovu podršku u Androidu kroz RxAndroid kao i podršku za Javu 8. Takođe će biti predstavljene i biblioteke “Butter Knife” za povezivanje promenljivih u Java kodu sa komponentama u xml-u preko anotacija, “Stetho“ koja olakšava pronalaženje i otklanjanje grešaka i „Fabric“ platforma za praćenje i analitiku. Aplikacija će služiti za pregled informacija o filmovima, korisniku će se predstaviti lista filmova koju može da pretražuje a klikom na željeni film prikazaće mu se dodatne informacije i imaće mogućnost da ga sačuva u personalizovanu listu, podeli i oceni. Na taj način će, pored teorijske analize i komparacije različitih tehnologija, biti prikazana i njihova implementacija u realnom okruženju i praktična primena u rešavanju najčešće viđenih problema u razvoju Android aplikacija. Aplikacija će težiti da bude razvijena u skladu sa dobrim praksama i da samim tim bude testabilna, modularna i jednostavna za buduće održavanje i eventualne izmene.

Cilj rada je da svaka od ovih tehnologija će biti sagledana i analizirana iz različitih perspektiva pri čemu će biti istaknute pozitivne i negativne strane i napravljeno kritičko poređenje sa osvrtom na neka ranija rešenja. Shodno tome rad će predstavljati svojevrsno istraživanje najkorišćenijih modernih tehnologija u razvoju Android aplikacija.

**Model-Pogled-Prezenter**

Model-Pogled-Prezenter je arhitekturalni šablon, nastao iz Model-Pogled-Kontroler šablona, koji se najviše koristi za kreiranje korisničkog interfejsa. MVP šablon se pojavljuje prvi put početkom 90-ih godina u kompaniji Taligent kao osnova za razvoj aplikacija u njihovom objektno orijentisanom CommonPoint razvojnom okruženju. Kasnije je ovaj šablon upotrebljen za kreiranje osnove za Dolphin Smalltak korisnički interfejs dok je 2006. Microsoft uveo MVP u dokumentaciju i primere za programiranje korisničkih interfejsa u .NET razvojnom okviru.

Glavne motivacije za uvođenje MVP šablona je podela odgovornosti u prezentacionoj logici, viši stepen modularnosti koda i mogućnost automatskog testiranja koda. Sa početkom razvoja Android operativnog sistema većina programera je gotovo celu logiku aplikacije pisala u aktivnostima aplikacije. Pokazalo se da su takve aplikacije jako teške za održavanje jer je prezentaciona logika usko povezana sa biznis logikom i modelom, pored toga automatsko testiranje aplikacija je gotovo nemoguće. Nakon toga je popularnost stekao Model-Pogled-Kontroler (MVC) šablon. Kod MVC šablona imamo tri komponente:

* Statičan pogled koji u sebi nema nikakvu logiku i koji ne čuva stanje. Pogled je u Androidu uglavnom implementiran u okviru statičnih resursnih xml datoteka.
* Model koji je zadužen za dohvatanje i čuvanje podataka. U konkretnoj implementaciji u modelu može biti komunikacija sa bazom ili pozivanje eksternog servisa, takođe u modelu je implementirana i biznis logika aplikacije.
* Kontroler implementira neophodne transformacije podataka za prikaz, pored toga kontroler može biti deljen između više pogleda i kontroliše koji će pogled biti prikazan. U Androidu kontroler je uglavnom implementiran u okviru aktivnosti.

Iako je ovaj šablon doneo izvesnu modularnost i podelu odgovornosti i dalje je u određenoj meri postojala spregnutost između prikaza i modela i biznis logike što se odražavalo na otežano pisanje automatskih testova.

Kod Model-Pogled-Prezenter šablona slično kao kod MVC šablona imamo tri komponente:

* Model – model ima potpuno istu ulogu i implementaciju kao kod MVC šablona. On služi za dohvatanje podataka i biznis logiku aplikacije, za razliku od MVC šablona, model je potpuno nezavisan od prikaza.
* Pogled – pogled je odgovoran za predstavljanje podataka i za prihvatanje interakcije korisnika. Poželjno je da pogled služi samo za jednostavan prikaz podataka koje dobije od prezentera kao i prosleđivanju korisničkih interakcija prezenteru i da u sebi sadrži što manje logike. Pogled najčešće implementiraju aktivnosti i fragmenti aplikacije.
* Prezenter – prezenter zahteva podatke od modela, vrši transformacije nad podacima tako da budu pogodni za prikaz, kontroliše prikaz i reaguje na korisničku interakciju. Za svaki pogled postoji poseban prezenter.

Pošto su pogled i prezenter usko povezani oni moraju da imaju referencu jedan ka drugom a sa druge strane je potrebno da budu modularni radi automatskog testiranja ili potpune zamene jednog od ta dva sloja. Iz tog razloga oni su abstrahovani i implementiraju interfejse koji služe kao svojevrsni “dogovor” između ta dva sloja i čine kod čitljivijim.



Model-Pogled-Prezenter shema

**Implementacija**

Postoji više načina za implentaciju MVP šablona, iako je ideja u osnovi identična, za određene slučajeve je bolje koristiti implementaciju ovog šablona sa izvesnim manjim izmenama. Implementacija koja će biti predstavljena je zasnovana na preporukama kompanije Google i prilagođena potrebama aplikacije.

Uzmimo sledeći scenario za primer na kome će biti pokazana implementacija MVP šablona. U aplikaciji koju razvijamo potrebno je dohvatiti podatke preko mreže sa REST servisa i prikazati podatke u listi na android uređaju. Na osnovu te liste korisnik će kasnije moći da pregleda detalje o filmovima, da favorizuje određene filmove, pretraži listu itd. Međutim za početak želimo samo da predstavimo listu gde će za svaki film biti prikazan naslov, godina premijere, ocena i poster. U skladu sa realističnim produkcionim scenariom potrebno je i da se omogući relativno jednostavno i brzo menjanje servisa za podatke kao i celokupnog prikaza liste. Takođe je potrebno da arhitektura aplikacije omogućava automatsko testiranje.

Za potrebe aplikacije razvijen je REST server u NodeJS-u radi nezavisnosti od spoljnih servisa kao i zbog veće kontrole nad resursima i boljeg predstavljanja bitnih koncepta. NodeJS je izabran kao jezik servera prevashodno zbog brzine i lakoće razvijanja jednostavnih servisa. Kada klijent uputi GET zahtev servisu na putanji /all servis će vratiti listu JSON objekata u sledećem formatu:

**{**

"id"**:** "1"**,**

"name"**:** "The Shawshank Redemption"**,**

"year"**:** 1994**,**

"rating"**:** 9.2**,**

"poster"**:**"/posters/1.jpg"

**}**

Gde je polje "poster" zapravo statička relativna putanja do resursa u vidu slike. Na klijentu će se koristiti biblioteke Dagger 2 za umetanje zavisnosti kao i OkHttp, Retrofit i Jackson za dovlačenje i parsiranje podataka i ReactiveX za reaktivnu obradu podataka. U daljem radu će biti objašnjena funkcija svake od ovih biblioteka i obrazloženo njihovo korišćenje, međutim za razumevanje MVP šablona nije neophodno detaljno poznavanje ovih biblioteka tako da će se preći na implementaciju.

Klasa MainMovieListModel će poslužiti kao model za listu filmova koju prikazujemo odnosno služiće za dohvatanje liste filmova sa servisa. Ova klasa je potpuno nezavisna od prezentera i pogleda. Jedina metoda koju će pozivati prezenter iz ove klase ima potpis Observable**<**List**<**Movie**>>** loadMovies**()**. Ova metoda je mogla vraćati i listu Movie objekta ali zbog kasnijeg demonstriranja reaktivnog načina programiranja metoda će vraćati Observable objekat nad listom filmova. Ukratko ovde je u okviru ReactiveX biblioteke implementiran posmatrač šablon (eng. Observer pattern), gde imamo objekat koji se posmatra (eng. Observable), koji je u ovom slučaju lista filmova, dok će metoda u prezenteru sa potpisom void loadMovies**()** pozivati istoimenu metodu iz modela i “osluškivati” i reagovati na sve promene koje se dešavaju nad listom filmova. Pored toga dobra je praksa da ni prezenter ni pogled ne čuvaju stanja i da se na taj način životni ciklus aktivnosti pojednostavi, iz tog razloga će u okviru modela biti implementirano i keširanje podataka. Kada prezenter zahteva podatke on će ih uvek dobijati u istom obliku ali će se u okviru modela kontrolisati da li su to već učitani podaci ili je potrebno dovući podatke sa servera.

Pogled će biti implementiran u okviru aktivnosti MainMovieListActivity. Ono što je bitno za pogled je da je on samo pasivna komponenta čija je glavna uloga prikaz liste i ostalih komponenata, pored toga u okviru pogleda će biti kontrolisan životni ciklus aktivnosti. Bilo kakva logika će biti smeštena u okviru prezentera koji će biti umetnut u pogled. Pogled će u onResume metodi postaviti referencu prezenteru ka sebi pozivanjem metode prezenteravoid setView**(**MainMovieListContract**.**View view**)**, zatim prikazati animaciju za učitavanje liste filmova i pozivati metodu void loadMovies**()** umetnutog prezentera. Nasuprot tome u metodi onPause**()** životnog ciklusa aplikacije poziva se metoda void resetView**()** prezentera koja uklanja pogled iz prezentera da bi se izbeglo “curenje” memorije (eng. memory leak) odnosno objekat koji ne bi uklonio sakupljač smeća (eng. garbage collector).

Prezenter će u metodi loadMovies pozivati metodu umetnutog modela sa potpisom Observable**<**List**<**Movie**>>** loadMovies**()** i kada se završi učitavanje liste filmova prezenter će reagovati na to tako što će ukloniti animaciju za učitavanje pozivanjem metode void hideProgressBar**()** pogleda a zatim pozvati i metodu za prikaz liste učitanih filmova void showMovies**(**ArrayList**<**Movie**>** movies**)** koja se takođe nalazi u pogledu. U nastavku je dat kod klasa MainMovieListActivity kao i MainMovieListPresenter.

public class MainMovieListActivity **extends** Activity **implements** MainMovieListContract**.**View **{**

@Inject

MainMovieListPresenter presenter**;**

MainMovieListAdapter adapter**;**

@BindView**(**R**.**id**.**movie\_list**)**

RecyclerView movieList**;**

@BindView**(**R**.**id**.**progress\_bar**)**

ProgressBar progressBar**;**

@Override

protected void onCreate**(**@Nullable Bundle savedInstanceState**)** **{**

**super.**onCreate**(**savedInstanceState**);**

**((**MovieMaster**)** getApplication**()).**getMovieComponent**().**inject**(this);**

setContentView**(**R**.**layout**.**activity\_main\_movie\_list**);**

ButterKnife**.**bind**(this);**

LinearLayoutManager layoutManager **=** **new** LinearLayoutManager**(this);**

movieList**.**setLayoutManager**(**layoutManager**);**

**}**

@Override

protected void onResume**()** **{**

**super.**onResume**();**

presenter**.**setView**(this);**

progressBar**.**setVisibility**(**View**.**VISIBLE**);**

presenter**.**loadMovies**();**

**}**

@Override

public void hideProgressBar**()** **{**

progressBar**.**setVisibility**(**View**.**GONE**);**

**}**

@Override

protected void onPause**()** **{**

**super.**onPause**();**

presenter**.**resetView**();**

**}**

@Override

public void showMovies**(**ArrayList**<**Movie**>** movies**)** **{**

**if** **(**adapter **==** **null)** **{**

adapter **=** **new** MainMovieListAdapter**(**movies**);**

movieList**.**setAdapter**(**adapter**);**

**}** **else** **{**

adapter**.**notifyDataSetChanged**();**

**}**

**}**

**}**

public class MainMovieListPresenter **implements** MainMovieListContract**.**Presenter **{**

private MainMovieListModel mainMovieListModel**;**

private MainMovieListContract**.**View view**;**

Subscription movieSubscription**;**

public MainMovieListPresenter**(**MainMovieListModel mainMovieListModel**)** **{**

**this.**mainMovieListModel **=** mainMovieListModel**;**

**}**

@Override

public void setView**(**MainMovieListContract**.**View view**)** **{**

**this.**view **=** view**;**

**}**

@Override

public void loadMovies**()** **{**

movieSubscription **=** mainMovieListModel**.**loadMovies**()**

**.**subscribe**(new** Subscriber**<**List**<**Movie**>>()** **{**

@Override

public void onCompleted**()** **{**

Log**.**i**(**"INFO"**,** "Completed"**);**

**}**

@Override

public void onError**(**Throwable e**)** **{**

e**.**printStackTrace**();**

**}**

@Override

public void onNext**(**List**<**Movie**>** movies**)** **{**

view**.**hideProgressBar**();**

view**.**showMovies**((**ArrayList**<**Movie**>)** movies**);**

**}**

**});**

**}**

@Override

public void resetView**()** **{**

view **=** **null;**

**}**

**}**

Pogled i prezenter će implementirati interfejse MainMovieListContract**.**View i MainMovieListContract**.**Presenter gde je MainMovieListContract interfejs koji sadrži “ugovor“ između prezentera i pogleda i koji na jednostavan način opisuje njihov odnos.

public interface MainMovieListContract **{**

interface Presenter **{**

void setView**(**MainMovieListContract**.**View view**);**

void loadMovies**();**

void resetView**();**

**}**

interface View **{**

void showMovies**(**ArrayList**<**Movie**>** movies**);**

void hideProgressBar**();**

MainMovieListContract**.**Presenter**;**

**}**

**}**

Iako ovaj interfejs nije neophodan ono što je njegova uloga je da precizno pokaže odnos između pogleda i prezentera i da omogući laku zamenu jedne od te dve komponente. Iz priloženog interfejsa možemo i bez analize implementacija zaključiti da pogled može zatražiti od prezentera da učita filmove dok prezenter sa druge strane može očekivati od pogleda da te filmove prikaže i ukloni animaciju za učitavanje. Takođe bitno je napomenuti i da svaki pogled ima odgovarajući prezenter ali moguće je koristiti isti model za više prezentera i pogleda jer više srodnih aktivnosti može biti vezano za isti set podataka.

**Zaključak**

MVP je šablon koji nam omogućava da efikasno razdvojimo prezentacionu i biznis logiku android aplikacije. Na taj način aplikacija postaje modularnija a to za posledicu ima lakše održavanje, izmene celog prezentacione sloja ili biznis logike kao i lakše automatsko testiranje. Implementacija i prakse prikazane ovde su opšte smernice za pravilnu implementaciju MVP šablona ali naravno u zavisnosti od konkretnog problema implementacija može i drugačije izgledati. MVP je najpogodniji za veće projekte, u slučaju manjih projekata, naročito onih koji nisu podložni krupnijim izmenama, vreme koje će biti potrebno za implementaciju i problemi na koje se nailazi mogu nadmašiti benefite ovakve arhitekture.

**Reaktivno programiranje u Androidu**

Reaktivno programiranje je paradigma programiranja orijentisana oko tokova podataka i propagiranja promena. U reaktivnim programskim jezicima moguće je sa lakoćom manipulisati tokovima podataka i propagirati promene nad tim podacima kroz programski tok. Na primer, u imperativnim programskim jezicima izraz a **=** b **+** c bi jednostavno označavao da je promenljivoj a dodeljena vrednost izraza b **+** c ako bi se potom vrednost promenljive b ili c promenila to ne bi uticalo na vrednost promenljive a. Nasuprot tome u reaktivnoj paradigmi vrednost promenljive a bi se automatski ažurirala pri promeni neke od promenljivih b ili c.

Postavlja se pitanje kakvu bi primenu reaktivno programiranje moglo imati u razvoju Android aplikacija? Problem koji se često sreće u Android aplikacijama su promenljivi podaci koje aplikacija dobija sa servisa i koje zatim mora u realnom vremenu da prikaže. Tu se kao prirodno rešenje uz razmatrane arhitekturalne šablone nameće i reaktivno programiranje koje pojednostavljuje asinhrone pozive ka servisima kao i ažuriranje prikaza. Međutim za programiranje Android aplikacija koristi se programski jezik Java koji nije reaktivni programski jezik. Reaktivne ekstenzije (poznate kao ReactiveX) je skup alata koje dozvoljavaju imperativnim programskim jezicima da manipulišu nad tokovima podataka i događajima na reaktivan način. ReactiveX je kombinacija šablona posmatrača (eng. Observer pattern), iterator šablon (eng. Iterator pattern) kao i funkcionalnog programiranja. Implementacija ReactiveX-a koja se koristi za Javu je RxJava dok je implementacija za Android koja je jako slična RxJavi RxAndroid, obe implementacije su otvorenog koda.

**RxAndroid**

Osnovni gradivni elementi reaktivnog programiranja su emiter[[2]](#footnote-2) (eng. Observable) i pretplatnik[[3]](#footnote-3) (eng. Subscriber). Glavna uloga emitera je da emituje podatke dok pretplatnik te podatke prima i reaguje na promene podataka. Emiter emituje određeni broj objekata a zatim završava emitovanje uspešno ili usled neke greške koja se javila. Sa druge strane za svakog pretplatnika kog ima emiter poziva njegovu metodu Subscriber**.**onNext**()** onoliko puta koliko ima objekata a nakon toga poziva se Subscriber**.**onComplete**()** pri uspešnom završetku emitovanja ili Subscriber**.**onError**()** kada dođe do neke greške.

Postepeno i kroz primere pokazaćemo najbitnije koncepte i najkorišćenije operatore u RxAndroidu. Prvo kreirajmo emiter:

**new** Observable**.**OnSubscribe**<**String**>()** **{**

@Override

public void call**(**Subscriber**<?** **super** String**>** sub**)** **{**

sub**.**onNext**(**"Hello, world!"**);**

sub**.**onCompleted**();**

**}**

**}**

**);**

Emiter emituje nisku "Hello, world!" a zatim završava emitovanje. Kreirajmo sada i pretplatnika:

Subscriber**<**String**>** mySubscriber **=** **new** Subscriber**<**String**>()** **{**

@Override

public void onNext**(**String s**)** **{** System**.**out**.**println**(**s**);** **}**

@Override

public void onCompleted**()** **{** **}**

@Override

public void onError**(**Throwable e**)** **{** **}**

**};**

Sada kada imamo pretplatnika i emitera ako ih spojimo dobićemo odštampano "Hello, world!" .

myObservable**.**subscribe**(**mySubscriber**);**

Pokušaćemo da pojednostavimo kod, postoji operator Observable**.**just**()** koji emituje jedan objekat a zatim se završava isto kao u prethodni kod.

Observable**<**String**>** myObservable **=** Observable**.**just**(**"Hello,world!"**);**

Možemo pojednostaviti i kod vezan za pretplatnika. Jedino što je tu bitno je onNext**()** metoda tako da možemo da je izdvojimo.

Action1**<**String**>** onNextAction **=** **new** Action1**<**String**>()** **{**

@Override

public void call**(**String s**)** **{**

System**.**out**.**println**(**s**);**

**}**

**};**

Metoda Observable**.**subscribe**()** može umesto pretplatnika kao argument da primi akcije onNext**()**, onError**()** i onComplete**()** sali takođe može primiti samo onNext**()** akciju**.** Kada to uzmemo u obzir dobijamo:

myObservable**.**subscribe**(**onNextAction**);**

Odnosno kada spojimo kod za emiter i pretplatnika u jedan izraz dobijamo:

Observable**.**just**(**"Hello, world!"**)**

**.**subscribe**(new** Action1**<**String**>()** **{**

@Override

public void call**(**String s**)** **{**

System**.**out**.**println**(**s**);**

**}**

**});**

A kada još na to primenimo Java 8 lambda izraze imamo:

Observable**.**just**(**"Hello, world!"**)**

**.**subscribe**(**s **->** System**.**out**.**println**(**s**));**

Pretpostavimo još da želimo da nadovežemo još jednu nisku na postojeću. To bismo mogli da učinimo na više načina, na primer mogli bismo da nadovežemo nisku u emiteru ili u pretplatniku. Ali umesto nadovezivanja niske mogli bismo da imamo bilo koju drugu složeniju transformaciju emitovanih podataka. Za to je pogodno izdvojiti tu logiku. Postoji operator map**()** koji prihvata emitovane objekte i transformiše ih. Pritom transformisan objekat uopšte ne mora biti istog tipa kao originalan objekat, npr. Može prihvatiti nisku i vratiti njenu heš kod u obliku celobrojne vrednosti.

Observable**.**just**(**"Hello, world!"**)**

**.**map**(**s **->** s **+** " Hello!"**)**

**.**subscribe**(**s **->** System**.**out**.**println**(**s**));**

Takođe možemo nadovezivati više map**()** operatora.

Observable**.**just**(**"Hello, world!"**)**

**.**map**(**s **->** s **+** " Hello!"**)**

**.**map**(**s **->** s**.**hashCode**())**

**.**map**(**i **->** Integer**.**toString**(**i**))**

**.**subscribe**(**s **->** System**.**out**.**println**(**s**));**

Pretpostavimo sada da imamo metodu koja vraća emiter liste url-ova na osnovu upita.

Observable**<**List**<**String**>>** query**(**String text**);**

Želimo da odštampamo svaki od datih url-ova, to možemo učiniti na sledeći način :

query**(**"Hello, world!"**)**

**.**subscribe**(**urls **->** **{**

**for** **(**String url **:** urls**)** **{**

System**.**out**.**println**(**url**);**

**}**

**});**

Međutim na ovaj način ne možemo da koristimo operator map**()**za transformaciju url-ova, moguće je jedino transformisati u okviru pretplatnika što nam često ne odgovara jer je uloga pretplatnika često samo da ažurira prikaz informacija i zato se često izvršava na glavnoj niti. Postoji operator Observable**.**from**()** koji prima kolekciju objekata i emituje ih jednog po jednog, tako da možemo transformisati postojeći izraz:

query**(**"Hello, world!"**)**

**.**subscribe**(**urls **->** **{**

Observable**.**from**(**urls**)**

**.**subscribe**(**url **->** System**.**out**.**println**(**url**));**

**});**

Ovim smo dobili ugnježdene emitere što je teško za održavanje i sklono greškama programera. Postoji operator Observable**.**flatMap**()** koji prima objekte emitovane jednog emitera, transformiše svakog od njih u emitera a zatim emitovanja svakog od tih emitera spaja u jedno emitovanje (odnosno sekvencu svih emitovanih objekata različitih emitera). Kada primenimo to na naš slučaj dobijamo:

query**(**"Hello, world!"**)**

**.**flatMap**(new** Func1**<**List**<**String**>,** Observable**<**String**>>()** **{**

@Override

public Observable**<**String**>** call**(**List**<**String**>** urls**)** **{**

**return** Observable**.**from**(**urls**);**

**}**

**})**

**.**subscribe**(**url **->** System**.**out**.**println**(**url**));**

Kada to pojednostavimo lambda izrazima imamo:

query**(**"Hello, world!"**)**

**.**flatMap**(**urls **->** Observable**.**from**(**urls**))**

**.**subscribe**(**url **->** System**.**out**.**println**(**url**));**

Ono što je ovde ključno je da umesto liste url-ova sada pretplatnik dobija jedan po jedan url koji štampa. Iako na prvi pogled ovaj način rešavanja problema deluje nepotrebno komplikovan, sada dolazimo do benefita koje donosi. Recimo da imamo metodu Observable**<**String**>** getTitle**(**String URL**);** koja na osnovu url-a vraća naziv vebsajta ili null ako sajt nije pronađen. Ako bismo rešavali problem na tradicionalni način imali bismo asinhroni poziv koji bi dovlačio listu url-ova a zatim u njemu ugnježdene nove asinhrone pozive za naslov svakog od url-ova. Na taj način izbegavamo sinhronizovanje više poziva i sve probleme koji mogu tu da se jave. Sa RxAndroidom imaćemo:

query**(**"Hello, world!"**)**

**.**flatMap**(**urls **->** Observable**.**from**(**urls**))**

**.**flatMap**(**url **->** getTitle**(**url**))**

**.**subscribe**(**title **->** System**.**out**.**println**(**title**));**

Pretpostavimo da još želimo da izbacimo null, da prikažemo samo prvih 5 naslova i da pritom ih sačuvamo na disku. Sve to možemo na sledeći način:

query**(**"Hello, world!"**)**

**.**flatMap**(**urls **->** Observable**.**from**(**urls**))**

**.**flatMap**(**url **->** getTitle**(**url**))**

**.**filter**(**title **->** title **!=** **null)**

**.**take**(**5**)**

**.**doOnNext**(**title **->** saveTitle**(**title**))**

**.**subscribe**(**title **->** System**.**out**.**println**(**title**));**

Gde operator filter**()** vraća iste objekte koje prima ali samo ako zadovoljavaju logički uslov. Operator take**()** emituje najviše onoliko objekata koliko je navedeno. Operator doOnNext**()** nam omogućava da svaki put kada se objekat emituje izvršimo neku akciju, u ovom slučaju sačuvamo naziv.

Značajno je pomenuti da ako bilo koja od metoda u operatorima baca potencijalnu grešku, u samim operatorima ne moramo da je hvatamo i reagujemo na nju već je dovoljno da to učinimo u Subscriber**.**onError**()** metodi. Na taj način imamo centralizovano mesto za hvatanje i reagovanje na greške. Ono što je takođe korisno je to što operator Observable**.**subscribe**()** vraća objekat tipa Subscription ako iz bilo kog razloga želimo da pretplatnik više ne prati ono što emiter emituje možemo pozvati subscription**.**unsubscribe**()**, na taj način ne samo da pretplatnik neće pratiti šta se dalje emituje već će i prekinuti lanac izvršavanja operatora, odnosno programer ne mora da da piše dodatni kod da bi na siguran način prekinuo izvršavanje operatora.

U programiranju Android aplikacija naročito je bitno da se svi pozivi ka mreži, skupa izračunavanja i sične operacije koje mogu da traju određeni vremenski period, ne obavljaju na glavnoj niti da ne bi blokirale prikaz. Iz tog razloga je bitno da se takve operacije i u reaktivnom programiranju obavljaju van glavne niti ali da se pritom ažuriranje prikaza obavlja na glavnoj niti.Operator subscribeOn**()** služi da odredi na kojoj će niti emiter emitovati i vršiti transformacije nad podacima, sa druge strane operator observeOn**()** određuje nit na kojoj će se slati notifikacije pretplatniku. Uobičajno je da se cela komputacija i komunikacija sa mrežom obavljaju na glavnoj niti dok se slanje notifikacije obavlja na glavnoj niti, na taj način se postiže da prikaz nikada nije “zamrznut” a sa druge strane su na prikazu uvek najsvežije informacije. Primer dohvatanja slike sa mreže na niti za ulaz/izlaz i prikazivanja na glavnoj niti:

service**.**getImage**(**url**)**

**.**subscribeOn**(**Schedulers**.**io**())**

**.**observeOn**(**AndroidSchedulers**.**mainThread**())**

**.**subscribe**(**bitmap **->** myImageView**.**setImageBitmap**(**bitmap**));**

Pored pomenutih RxAndroid pruža još mnoštvo drugih operatora za najrazličitije operacije nad podacima ali naravno RxAndroid nije rešenje za sve probleme na koje se nailazi u razvoju Android aplikacija i ne treba da se koristi u svakoj situaciji. Neki od problema koja se najčešće javljaju u implementaciji su povezani sa životnim ciklusom aplikacije. Naime pri promeni orijentacije ekrana ili pri prebacivanju aplikacije u pozadinu a zatim ponovnom vraćanju u fokus, pretplatnik ponovo osluškuje emitera što dovodi do ponovnog izvršavanja koda emitera, što može biti skupa operacija dohvatanja podataka sa mreže ili iz baze. Rešenje za ovo je keširanje podataka, na taj način se mogu koristiti keširani podaci i izbeći skupi pozivi. Drugi problem u vezi životnog ciklusa koji se javlja je curenje memorije pri prebacivanju aplikacije u pozadinu jer emiter zadržava Context aplikacije. Iz tog razloga je potrebno da svi pretplatnici u onDestroy**()** metodi životnog ciklusa pozovu prestanu da prate emitera tako što se nad njima poziva metoda unsubscribe**().** Na taj način se oslobođa referenca ka emiteru da bi sakupljač smeća mogao da ukloni objekat iz memorije. Za razliku od problema sa životnim ciklusom, problem za koji ne postoji jednostavno rešenje je ukoliko emiter emituje podatke brže nego što ih pretplatnik ili operator obrađuju. Takva situacija dovodi do nagomilavanja podataka na baferu i do iscrpljivanja sistemskih resursa. Postoje različite strategije za rešavanje ovog problema kao i operatori koji se koriste za to. Na svu sreću ovaj problem nije karakterističan za Android, najviše zbog toga što se u savremenom programiranju Android aplikacija teži tome da klijent bude što lakši dok se sva zahtevna logika premešta na server.

**Implementacija**

U aplikaciji MovieMaster imamo zadatak da prvo učitamo sa mreže listu filmova a zatim za svaki od filmova iz liste učitamo sa statičke adrese na serveru poster za taj film. Pored toga potrebno je i da keširamo podatke i da na taj način izbegnemo nepotrebne i skupe pozive ka servisu.

Razmotrimo prvo rešavanje ovog problema na standardan način, koristeći asinhrone pozive. Za tu svrhu se koristi klasa AsyncTask koja služi za izvršavanje skupih operacija u pozadini i ažuriranje prikaza na glavnoj niti. Asinhroni poziv je definisan sa tri tipa:

* Params - parametri koji su potrebni za izvršavanje asinhronog poziva (npr. url sa kog se dobijaju podaci)
* Progress - podaci o napretku izvršavanja pozadinske operacije (npr. celobrojna procentualna vrednost)
* Result – rezultat izvršavanja asinhronog poziva (npr. lista objekata sa mreže)

Asinhroni poziv ima četiri faze izvršavanja:

* onPreExecute**()** - izvršava se na glavnoj niti pre glavnog poziva, služi za obavljanje pripreme za glavni poziv, na primer za prikazivanje animacije za učitavanje
* doInBackground**(**Params**...)** – izvršavanje asinhronog poziva koji dugo traje. Parametri se koriste za sam poziv, i ova metoda vraća rezultat koji se kasnije koristi u onPostExecute**(**Result**)** metodi. Takođe pozivanjem metode publishProgress**(**Progress..**)** se poziva metoda onProgressUpdate**(**Progress**...)**
* onProgressUpdate**(**Progress**...)** - izvršava se na glavnoj niti i prikazuje napredovanje, npr. ažurira animaciju za napredovanje
* onPostExecute**(**Result**)** – izvršava se na glavnoj niti kada se završi asinhroni poziv

Kao što vidimo izvršavanje asinhronog poziva je prilično složeno u Androidu. U našem slučaju morali bismo da izvršimo asinhroni poziv da bismo dobili sve filmove a zatim u onPostExecute**(**Result**)** metodi da za svaki film u petlji pravimo još po jedan asinhroni poziv za dohvatanje postera. Uz to je potrebno voditi računa o prikazivanju animacije za učitavanje i upravljati greškama. I pored dobre organizacije koda to bi dovelo do koda koji nije čitljiv, težak je za održavanje i sklon greškama. Ako bismo promenili scenario i dodali treći poziv dobili bismo još jedan nivo ugnježdenosti i još teži kod za održavanje.

Pogledajmo sada kako se to rešava pomoću RxAndroida, u klasi MainMovieListModel imamo metodu loadMovies**()** koja služi za učitavanje liste filmova koji će biti prikazani.

public Observable**<**List**<**Movie**>>** loadMovies**()** **{**

**if** **(**cachedMovies **!=** **null)** **{**

**return** Observable**.**create**((**Observable**.**OnSubscribe**<**List**<**Movie**>>)** **this::**getMoviesFromCache**)**

**.**observeOn**(**AndroidSchedulers**.**mainThread**())**

**.**subscribeOn**(**Schedulers**.**io**());**

**}** **else** **{**

**return** Observable**.**create**((**Observable**.**OnSubscribe**<**List**<**Movie**>>)** **this::**getMovies**)**

**.**flatMap**(**Observable**::**from**)**

**.**map**(**movie **->** apiServiceWrapper**.**getPoster**(**movie**))**

**.**toList**()**

**.**doOnNext**(**movies **->** cacheMovies**(**movies**))**

**.**observeOn**(**AndroidSchedulers**.**mainThread**())**

**.**subscribeOn**(**Schedulers**.**io**());**

**}**

**}**

Posmatrajmo prvo else granu metode, kada nema keširanih mečeva, tada se prvo kreira emiter pozivanjem metode: Observable**.**create**((**Observable**.**OnSubscribe**<**List**<**Movie**>>)** **this::**getMovies**)** gde operator create prihvata pretplatnika kao parametar gde su definisane akcije onNext, onError i onCompleted. Ukoliko je ovaj način zapisa nije dovoljno jasan, ovo može biti zapisano i bez Java 8 sintakse:

Observable**.**create**(new** Observable**.**OnSubscribe**<**List**<**Movie**>>()** **{**

@Override

public void call**(**Subscriber**<?** **super** List**<**Movie**>>** subscriber**){**

MainMovieListModel**.this.**getMovies**(**subscriber**);**

**}**

**})**

Gde metoda getMovies služi za dohvatanje filmova sa mreže:

private void getMovies**(**Subscriber**<?** **super** List**<**Movie**>>** subscriber**)** **{**

**try** **{**

subscriber**.**onNext**(**apiServiceWrapper**.**getAllMovies**());**

subscriber**.**onCompleted**();**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

subscriber**.**onError**(**e**);**

**}**

**}**

Gde apiServiceWrapper**.**getAllMovies**()** sadrži logiku slanja zahteva preko mreže i obrade odgovora i vraća listu filmova. Na taj način je kreiran emiter koji emituje listu filmova.

Zatim imamo flatMap**(**Observable**::**from**)** gde smo ova dva operatora u kombinaciji već objašnjena, i čiji je konačni efekat da razdvoje emitovanje liste filmova u emitovanje jednog po jednog filma. Svaki emitovani objekat klase Movie zatim prima map**()** operator i dohvata sa mreže njegov poster map**(**movie **->** apiServiceWrapper**.**getPoster**(**movie**))**. Pošto metoda vraća emiter liste filmova a ne listu emitera koristimo operator toList**()** da bismo spojili natrag sve u listu. Onda keširamo podatke za svako emitovanje liste operatorom **.**doOnNext**(**movies **->** cacheMovies**(**movies**))**. Gde je cacheMovies**(**List**<**Movie**>** movies**)** jednostavna metoda koja kešira filmove u memoriji.

private void cacheMovies**(**List**<**Movie**>** movies**)** **{**

cachedMovies **=** movies**;**

**}**

Na kraju imamo operator observeOn**(**AndroidSchedulers**.**mainThread**())** koji određuje da će se notifikacije pretplatniku slati na glavnoj niti i subscribeOn**(**Schedulers**.**io**())** koji određuje da će emiter vršiti emitovanje i obavljati transformacije na niti predviđenoj za operacije u pozadini. Uprošćeno mogli bismo reći da smo pomoću RxAndroida u samo nekoliko redova zapisali sledeće: dohvati listu filmova sa mreže, razdvoj ih u pojedinačne objekte, za svaki dohvati poster, spoji ih ponovo u listu i keširaj ih, sve to uradi u pozadini dok obavesti na glavnoj niti.

If grana metode je vrlo slična, jedina razlika je što listu kompletno formiranu dobijamo iz memorije a ne sa mreže i nije potrebno vršiti dodatne transformacije.

private void getMoviesFromCache**(**Subscriber**<?** **super** List**<**Movie**>>** subscriber**)** **{**

**try** **{**

subscriber**.**onNext**(**cachedMovies**);**

subscriber**.**onCompleted**();**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

subscriber**.**onError**(**e**);**

**}**

**}**

Pretplatnik će se nalaziti u klasi MainMovieListPresenter i imaće ulogu da pri svakom emitovanju sakrije animaciju za učitavanje i pozove metodu pogleda da ažurira prikaz. U slučaju greške ili završenog emitovanja štampaće poruku u konzoli.

@Override

public void loadMovies**()** **{**

movieSubscription **=** mainMovieListModel**.**loadMovies**()**

**.**subscribe**(new** Subscriber**<**List**<**Movie**>>()** **{**

@Override

public void onCompleted**()** **{**

Log**.**i**(**"INFO"**,** "Completed"**);**

**}**

@Override

public void onError**(**Throwable e**)** **{**

e**.**printStackTrace**();**

**}**

@Override

public void onNext**(**List**<**Movie**>** movies**)** **{**

view**.**hideProgressBar**();**

view**.**showMovies**((**ArrayList**<**Movie**>)** movies**);**

**}**

**});**

**}**

**Zaključak**

Reaktivno programiranje je nova paradigma u programiranju Android aplikacija. Sa obzirom da je Android programiranje klijentsko programiranje gde je najčešća situacija da aplikacija dobija podatke sa mreže i treba da reaguje na njih, odnosno da ih obradi i prikaže, jasno je da se ovakav oblik programiranja nameće kao prirodno rešenje. Koristeći RxAndroid u kombinaciji sa Java 8 sintaksom imamo čitljiviji i jednostavniji kod, koji je lakši za održavanje. Naravno i ovaj pristup ima svojih mana, promena razmišljanja sa imperativne na reaktivnu paradigmu inicijalno zahteva puno vremena i rada. Pored toga jednostavne zadatke kao i neke specifične slučajeve nije pogodno koristi RxAndroid. Takođe u situacijama gde emiter emituje brže nego što operator ili pretplatnik mogu da obrađuju podatke vodi do problema, koji se ne javljaju u imperativnom programiranju.

1. SOLID (Single responsibility, Open/closed, Liskov Substitution, Interface segregation, Dependancy Injection) je skraćenica za pet principa dizajna koji doprinose fleksibilnosti i lakšem održavanju softvera. [↑](#footnote-ref-1)
2. Grub prevod termina Observable je “objekat koji može biti posmatran”, zbog nepostojanja literature na srpskom o ovoj temi i zato što je glavna uloga Observable-a da emituje podatke ovaj termin je preveden kao emiter [↑](#footnote-ref-2)
3. Subscriber se može prevesti i kao “osluškivač” [↑](#footnote-ref-3)