Univerzitet u Beogradu

Matematički fakultet

Komparativna analiza modernih tehnologija u razvoju Android aplikacija

Master rad

Kandidat:

Stefan Bačević

Mentor:

Prof. dr Vladimir Filipović

Sadržaj

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) može se 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] princip inverzije kontrole započeto je korišćenje bilblioteka koje omogućavaju umetanje zavisnosti kao što su RoboGuice i Dagger.

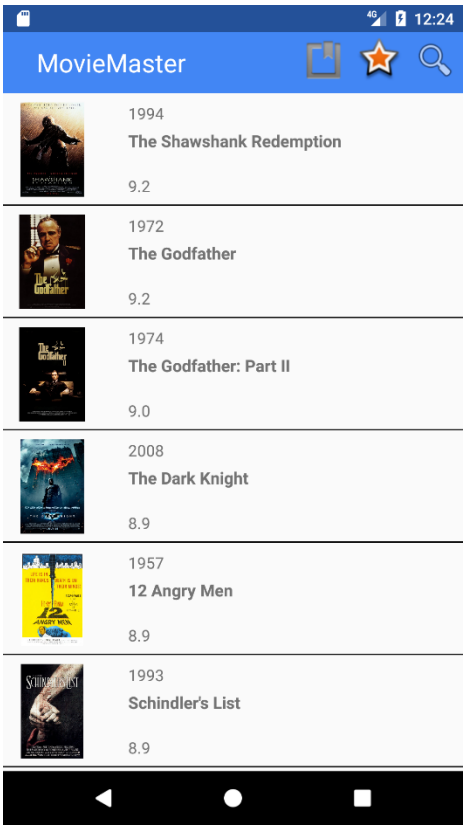
U okviru rada kreirana je i aplikacija otvorenog koda Movie Master koja pored pomenutih arhitekturalnih šablona i biblioteka prikazuje još i reaktivni način programiranja i njegovu podršku u Androidu kroz RxAndroid kao i podršku za Javu 8. Takođe predstavljene su i biblioteke Butter Knife za povezivanje 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 služi za pregled informacija o filmovima, korisniku je prikazana lista filmova koju može da pretražuje a klikom na željeni film prikazuju mu se dodatne informacije i otvara mogućnost da ga sačuva u personalizovanu listu, podeli i oceni. Na taj način je, pored teorijske analize i komparacije različitih tehnologija, prikazana i njihova implementacija u realnom okruženju i praktična primena u rešavanju najčešće viđenih problema u razvoju Android aplikacija. Aplikacija teži 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 se svaka od ovih tehnologija sagleda i analizira iz različitih perspektiva pri čemu su istaknute pozitivne i negativne strane i napravljeno kritičko poređenje sa osvrtom na neka ranija rešenja. Shodno tome rad predstavlja svojevrsno istraživanje najkorišćenijih modernih tehnologija u razvoju Android aplikacija.

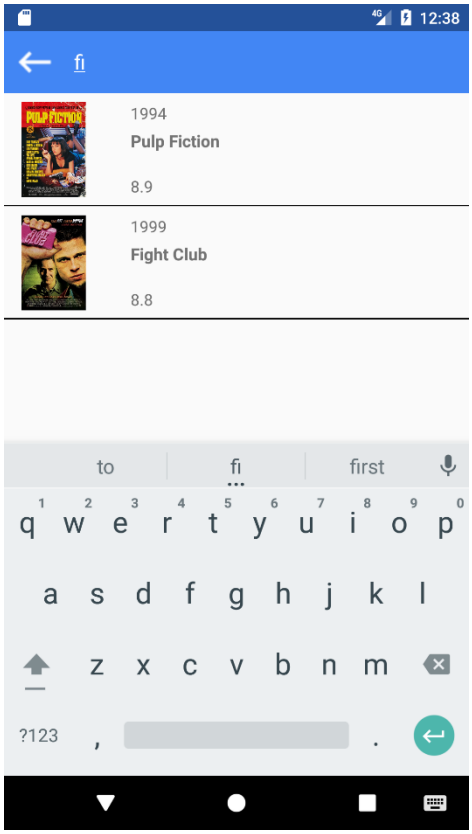
1 SOLID (Single responsibility, Open/closed, Liskov Substitution, Interface segregation, Dependancy Injection) je skraćenica za pet principa dizajna koji doprinose fleksibilnosti i lakšem održavanju softvera.

Specifikacije aplikacije

Aplikacija razvijena u okviru ovog rada ima ulogu da se na njom predstave tehnologije, arhitekturalni šabloni i dobre prakse u realnom kontekstu. Aplikacija služi za pregled filmova, korisnik inicijalno dobija listu filmova sa servera gde su prikazani podaci o imenu, godini izdanja filma, oceni kao i poster filma. Takođe implementirana je i pretraga filmova iz liste filmova dobijenih sa servera.

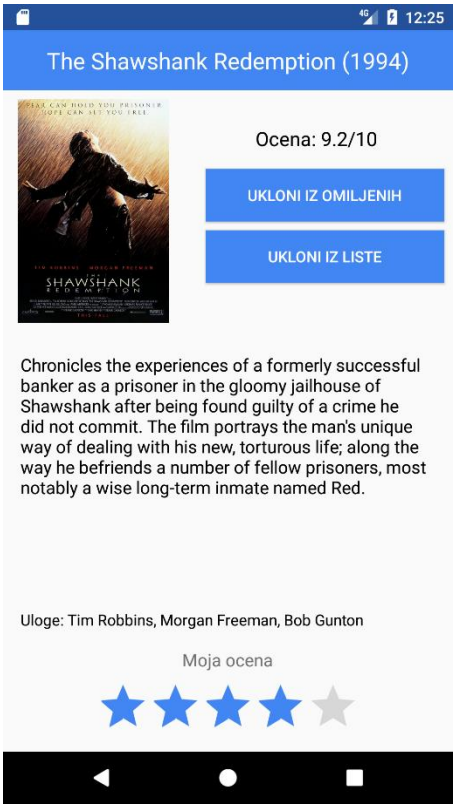


Slika 1 - Inicijalni ekran sa prikazanom listom filmova

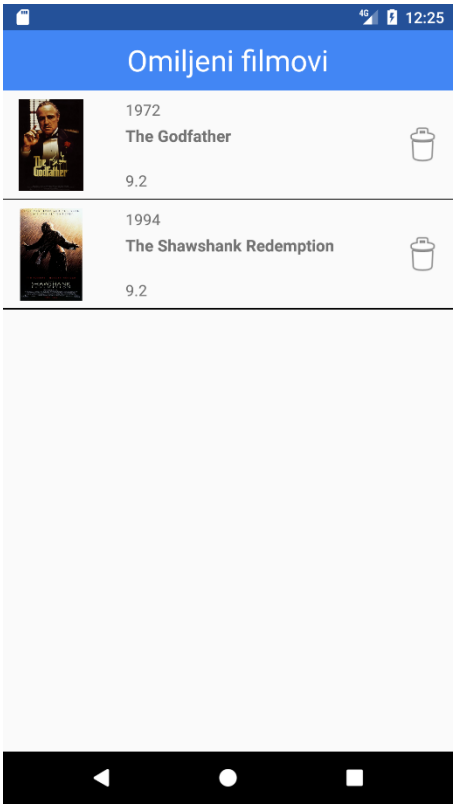


Slika 2 - Pretraga liste filmova

Klikom na film iz liste otvara se novi ekran koji prikazuje više informacija o izabranom filmu, dodatne informacije o filmu takođe dolaze sa servera u odvojenom zahtevu. Pored dodatnih informacija koje uključuju kratko opisanu radnju filma, uloge i ocenu korisnika, data je i mogućnost korisniku da doda film u listu omiljenih filmova ili listu filmova koje planira kasnije da pogleda kao i da da sopstvenu ocenu filma. Pomenute liste kao i ocena filma se čuvaju u lokalnoj bazi aplikacije i moguće je pregledati liste u posebnim ekranima do kojih se dolazi klikom na ikonice u zaglavlju glavnog ekrana.



Slika 3 - Detalji filma



Slika 4 - Omiljeni filmovi

Specifikacije aplikacije su slične zahtevima koji se često javljaju u aplikacijama slične funkcionalnosti i omogućavaju praktično demonstriranje svih tema obrađenih u radu.

Arhitektura aplikacije

Aplikacija se sastoji iz jednostavnog serverskog dela i klijentske Android aplikacije.

Server

REST server je razvijen zbog nezavisnosti od spoljnih servisa kao i zbog veće kontrole nad resursima i boljeg predstavljanja bitnih koncepta. Serverski kod je napisan u programskom jeziku JavaScript i koristeći programski okvir Express koji će se izvršavati na NodeJS izvršnom okruženju. Tehnologije koje će se koristiti na serverskoj strani su izbrane prevashodno zbog brzine i lakoće razvijanja jednostavnih servisa.

Postoje dva REST servisa: servis */all* za vraćanje liste svih filmova i servis */movieDetails* za vraćanje detalja o izabranom filmu. Pored toga na serveru se nalaze statički resursi u vidu slika odnosno postera filmova.

Kada klijent uputi GET zahtev servisu na putanji */all* servis vraća 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 sličan način kada klijent uputi GET zahtev servisu na putanji */movieDetails/:id,* gde je *id* jedinstveni identifikator filma na serveru, dobije odgovor koji pored ovih informacija sadrži i dodatne informacije o filmu.

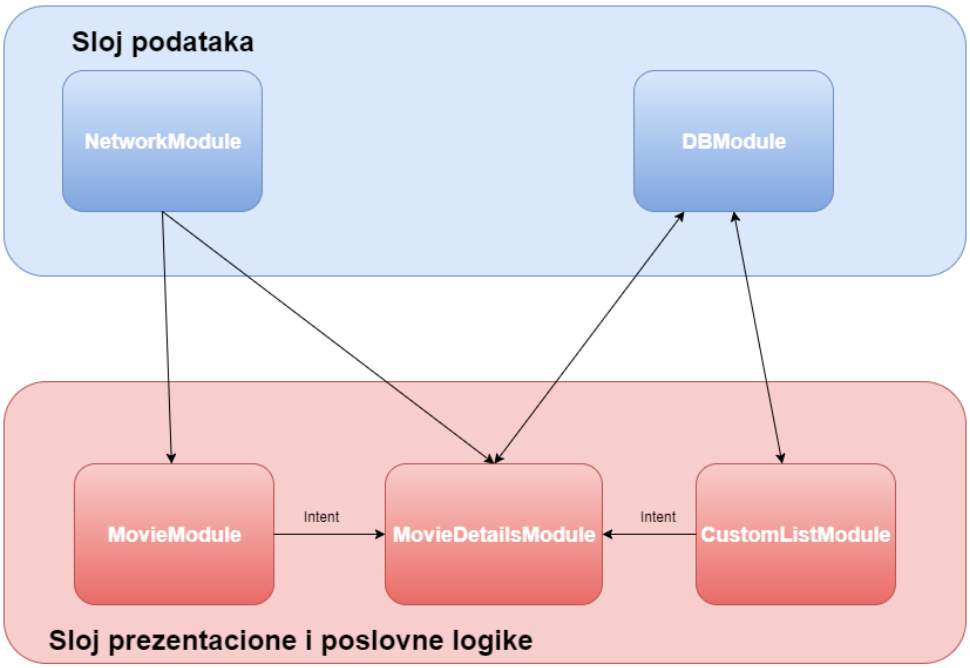
Android aplikacija

Akcenat pri razvijanju globalne arhitekture Android aplikacije bio je na modularnosti i ideji da moduli između sebe budu povezani ali nezavisni. Modularna arhitektura donosi mnoštvo prednosti u odnosu na monolitnu, nema spregnutosti između različitih delova aplikacije i moguće je zameniti ceo modul ili funkcionalnost uz minimalne promene. Iako je inicijalno potreban nešto veći napor da se aplikacija izgradi na ovaj način, prednosti u

održavanju, rejuzabilnost koda i lakoći pravljenja izmena jako brzo nadoknade vreme utrošeno na inicijalnu postavku arhitekture.

Posmatrajući globalno, aplikacija se sastoji iz dva sloja, sloja podataka i sloja koji sadrži poslovnu i prezentacionu logiku (poslovna logika je dalje u okviru modula razdvojena od prezentacione logike). Aplikacija je arhitekturalno razdvojena na sledeće module:

- **NetworkModule** - modul koji obezbeđuje komunikaciju sa serverom preko mreže
- **DBModule** - modul koji obezbeđuje CRUD[2] operacije nad lokalnom bazom
- **MovieModule** - modul koji služi za prikaz ekrana sa listom svih filmova, odnosno modul koji povezuje model, pogled i prezenter liste svih filmova u MVP šablonu
- **MovieDetailsModule** – modul koji služi za prikaz ekrana sa detaljima odabranog filma, povezuje model, pogled i pogled-model u MVVM šablonu
- **CustomListModule** – modul koji omogućava prikaz personalizovanih korisničkih lista (omiljenih filmova i filmova za kasnije gledanje)



Dijagram 1 - Globalna arhitektura aplikacije

U okviru *MovieModule*-a implementiran je Model-Pogled-Prezanter šablon radi daljeg razdvajanja poslovne logike i u okviru *MovieDetailsModule*-a je implementiran Model-Pogled-Pogled Model šablon iz istih razloga, o svemu tome će biti reči kasnije u radu.

[2] CRUD (eng. Create, Read, Update, Delete) – kreiranje, čitanje, ažuriranje i brisanje iz baze

Razlog iz kog su korišćene različite arhitekture u ova dva modula je prikaz i poređenje ova dva arhitekturalna šablona u Androidu.

NetworkModule je povezan sa *MovieModule*-om i *MovieDetailsModule*-om da bi omogućio vraćanje liste filmova i detalja izabranog filma sa mreže. *DBModule* je povezan sa *MovieDetailsModule*-om i *CustomListModule*-om da bi omogućio čuvanje i dohvatanje filmova iz personalizovanih lista sačuvanih u bazi i ocena filma koji je dao korisnik. Jedina zavisnost koja postoji između modula u prezentacionom sloju je što se id filma prosleđuje *MovieDetailsModule*-u kroz nameru (eng. Intent). Namera je abstraktan opis operacije koja treba da se izvrši i, između ostalog, koristi se za startovanje aktivnosti i u okviru namere je moguće proslediti parametre koje će startovana aktivnost da koristi. [1]

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 [2].

Glavne motivacije za uvođenje MVP šablona su 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 poslovnom 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:

- **Pogled** koji je statičan, 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 poslovna 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.

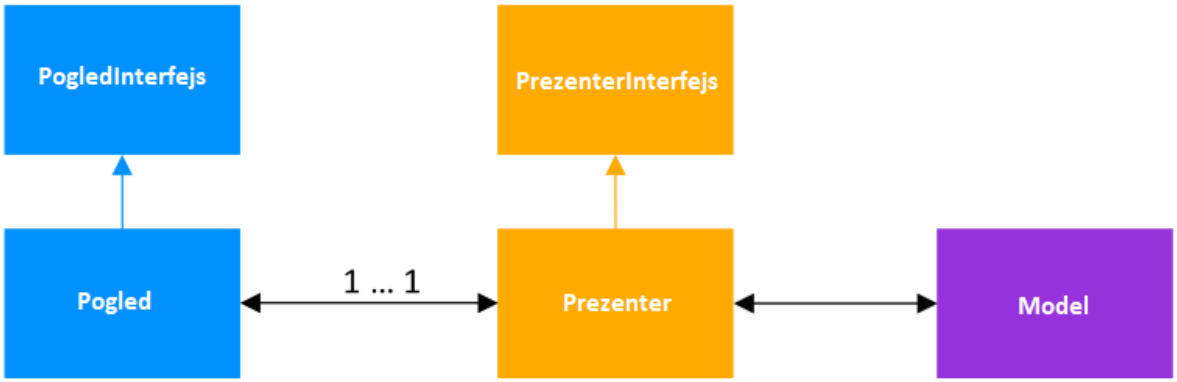
Iako je ovaj šablon doneo izvesnu modularnost i podelu odgovornosti i dalje je u određenoj meri postojala spregnutost između prikaza i modela i poslovne logike što se odražavalo na otežano pisanje automatskih testova.

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 poslovnu 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.



DIjagram 2 - Model-Pogled-Prezenter

Na ovaj način se postiže povezanost između slojeva a da se pritom ne gubi fleksibilnost. Bitno je napomenuti još da iako je prezenter u odnosu 1-1 sa pogledom to ne znači da treba da čuva u sebi stanje pogleda, ili da ima duplirane metode životnog ciklusa aktivnosti. Prezenter ne treba da prolazi kroz bilo kakva stanja, ako je potrebno sačuvati stanje podataka, to je moguće učiniti u modelu bilo privremenim čuvanjem u memoriji (keširanjem) ili čuvanjem u lokalnoj bazi što se može iskoristiti i prilikom narednog pokretanja aplikacije.

Implementacija u aplikaciji

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 je predstavljena je zasnovana na preporukama kompanije Google [3] i prilagođena potrebama aplikacije.

Kao što je već pomenuto, implementacija MVP šablona u aplikaciji je prikazana na modulu koji služi za prikazivanje liste filmova. 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. U aplikaciji se koriste biblioteke Dagger 2 za umetanje zavisnosti kao 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 se prelazi na implementaciju.

Klasa `MainMovieListModel` služi kao model za listu filmova koju prikazujemo odnosno služi za dohvatanje liste filmova sa servisa. Ova klasa je potpuno nezavisna od prezentera i pogleda. Jedina metoda koju poziva 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 vraća `Observable` objekat nad listom filmova. Ukratko ovde je u okviru ReactiveX biblioteke implementiran posmatrač šablon [4] (eng. Observer pattern), gde postoji objekat koji se posmatra (eng. Observable), koji je u ovom slučaju lista filmova, dok metoda u prezenteru sa potpisom `void loadMovies()` poziva istoimenu metodu iz modela i "osluškuje" i reaguje 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 je u okviru modela implementirano i keširanje podataka. Kada prezenter zahteva podatke on ih uvek dobija u istom obliku ali se u okviru modela kontroliše da li su to već učitani podaci ili je potrebno učitati podatke sa servera.

Pogled i prezenter implementiraju 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(String query);
        void resetView();
    }

    interface View {
        void showMovies(ArrayList<Movie> movies);
        void hideProgressBar();
        void gotoMovieDetails(int movieId);
    }
}
```

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 može se 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.

Pogled je 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 se kontroliše životni ciklus aktivnosti. Logika je smeštena u okviru prezentera koji je umetnut u pogled. Pogled u `onResume` metodi postavlja referencu prezenteru ka sebi pozivanjem metode prezentera `void setView(MainMovieListContract.View view)`, zatim prikazuje animaciju za učitavanje liste filmova i poziva 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 otpadaka (eng. garbage collector).

Prezenter u metodi `loadMovies` poziva metodu umetnutog modela sa potpisom `Observable<List<Movie>> loadMovies()` i kada se završi učitavanje liste filmova prezenter reaguje na to tako što uklonja animaciju za učitavanje pozivanjem metode `void hideProgressBar()` pogleda a zatim poziva 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;
    @BindView(R.id.header_name)
    TextView headerName;
    @BindView(R.id.search)
    View search;
    @BindView(R.id.search_clear)
    View searchClear;
    @BindView(R.id.search_box)
    EditText searchBox;
    @BindView(R.id.navigation_holder)
    ViewGroup navigationHolder;


    @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);
    headerName.setText(getString(R.string.movie_master));

    setupSearch();
}


@Override
protected void onResume() {
    super.onResume();
    presenter.setView(this);
    progressBar.setVisibility(View.VISIBLE);
    presenter.loadMovies("");
}

@Override
public void hideProgressBar() {
    progressBar.setVisibility(View.GONE);
}

@Override
public void gotoMovieDetails(int movieId) {
    Intent intent = new Intent(this, MovieDetailsActivity.class);
    intent.putExtra(Constants.MOVIE_ID, movieId);
    startActivity(intent);
}

@Override
protected void onPause() {
    super.onPause();
    presenter.resetView();
}

@Override
public void showMovies(ArrayList<Movie> movies) {
    if (adapter == null) {
        adapter = new MainMovieListAdapter(movies, this);
        movieList.setAdapter(adapter);
    } else {
        adapter.update(movies);
    }

}

private void setupSearch() {
    search.setVisibility(View.VISIBLE);
    search.setOnClickListener(v -> {
        navigationHolder.getChildAt(0).setVisibility(View.INVISIBLE);
        navigationHolder.getChildAt(1).setVisibility(View.VISIBLE);
        LayoutUtils.showKeyboard(searchBox);
    });
    searchClear.setOnClickListener(v -> {
        navigationHolder.getChildAt(0).setVisibility(View.VISIBLE);
        navigationHolder.getChildAt(1).setVisibility(View.INVISIBLE);
```

```
            LayoutUtils.hideKeyboard(searchBox);
            searchBox.setText("");
        });

        searchBox.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start, int
count, int after) { }

            @Override
            public void onTextChanged(CharSequence s, int start, int before,
int count) { }

            @Override
            public void afterTextChanged(Editable s) {
                String query = s.toString();
                presenter.loadMovies(query);
            }
        });

    }

    @OnClick(R.id.watchlist)
    public void gotoWatchlist(){
        Intent intent = new Intent(this, CustomListActivity.class);
        intent.putExtra(Constants.TYPE, Constants.WATCHLIST);
        startActivity(intent);
    }

    @OnClick(R.id.favourites)
    public void gotoFavourites(){
        Intent intent = new Intent(this, CustomListActivity.class);
        intent.putExtra(Constants.TYPE, Constants.FAVOURITES);
        startActivity(intent);
    }
}



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(String query) {
```

```
        movieSubscription = mainMovieListModel.loadMovies(query)
                .subscribe(new Subscriber<List<Movie>>() {
                    @Override
                    public void onCompleted() {
                        Log.i("INFO", "Completed");
                    }

                    @Override
                    public void onError(Throwable e) {
                        e.printStackTrace();
                        Crashlytics.logException(e);
                    }

                    @Override
                    public void onNext(List<Movie> movies) {
                        view.hideProgressBar();
                        view.showMovies((ArrayList<Movie>) movies);
                    }
                });
    }

    @Override
    public void resetView() {
        view = null;
    }

}
```

Zaključna razmatranja za MVP

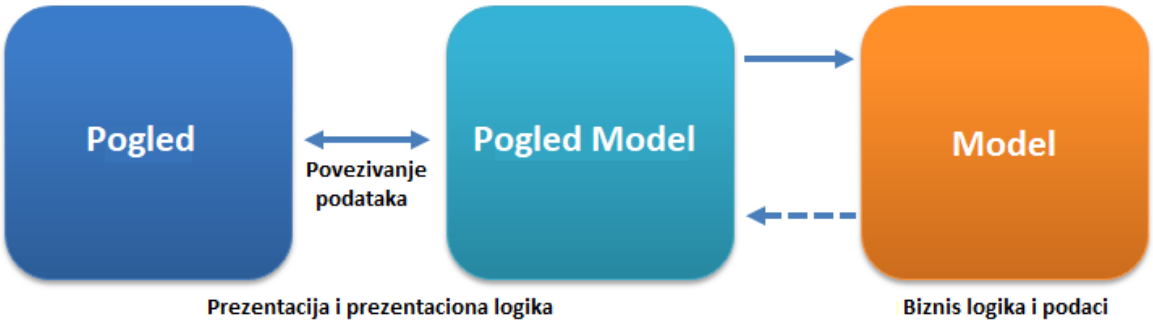
MVP je šablon koji omogućava efikasno razdvojanje prezentacione i poslovne logike Android aplikacije. Na taj način aplikacija postaje modularnija a to za posledicu ima lakše održavanje, izmene celog prezentacionog sloja ili poslovne logike kao i lakše automatsko testiranje. Implementacija i prakse prikazane u ovom poglavlju predstavljaju opšte smernice za pravilnu implementaciju MVP šablona, ali naravno u zavisnosti od konkretnog problema implementacija može i drugačije izgledati. Nepostojanje čvrstih odrednica za implementaciju MVP šablona može dovesti i do problema da interfejsi koje implementiraju prezenter i pogled postanu previše komplikovani i usko povezani. MVP je najpogodniji za veće projekte, dok 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 prednosti koje pruža ovakva arhitektura.

Model-Pogled-Pogled Model

Model-Pogled-Pogled Model (u daljem tekstu MVVM) je arhitekturalni šablon koji olakšava podelu između razvoja grafičkog korisničkog interfejsa i poslovne logike. MVVM šablon je, slično kao i MVP šablon, izveden iz MVC šablona. Prvobitno je razvijen od strane Microsoft-ovih inženjera Ken Kupera i Ted Patersa sa ciljem da pojednostave programiranje korisničkih interfejsa, naročito interakciju sa korisničkim interfejsom koristeći funkcionalnosti Windows Presentation Foundation (WPF) - grafičkog sistema u .NET programskom okviru [5]. Džon Gosman, jedan od arhitekta WPF-a je objavio MVVM na svom blogu 2005. godine. MVVM se takođe naziva i Model-Pogled-Povezivač (Model-View-Binder).

MVVM se sastoji od sledećih komponenata:

- **Model** – model ima potpuno istu ulogu kao i kod MVC i MVP šablona. Služi za učitavanje podataka i poslovnu logiku aplikacije i nezavisan je od ostalih slojeva.
- **Pogled** – pogled je odgovoran za predstavljanje podataka. Slično kao kod MVP šablona implementacija pogleda u Androidu je u aktivnostima ili fragmentima. Ono što se razlikuje je da se iz aktivnosti ne kontrolišu komponente u xml prikazu već samo uspostavi povezivanje podataka između prikaza opisanog xml datotekom i modela pogleda. Takođe u okviru XML datoteke je moguće imati određenu logiku, međutim dobra je praksa izostaviti bilo kakvu logiku iz pogleda ili eventualno imati što jednostavniju logiku u pogledu usko vezanu za prikaz. U suprotnom može doći do velikog broja grešaka koje se teško otkrivaju i do visoke spregnutosti pogleda i logike.
- **Pogled model** – pogled model je abstrakcija pogleda u kojoj su izložene javna svojstva i komande nad pogledom. Pogled model učitava sve neophodne podatke iz modela i zatim izlaže sve što je potrebno pogledu. Na taj način se postiže da pogled bude pasivan i da se svaka promena na podacima automatski prosleđuje ka pogledu bez potrebe da programer piše kod koji bi ažurirao pogled i bez potrebe za eksplicitnom referencom ka pogledu.
- **Povezivanje podataka** – u MVVM-u je takođe potrebna komponenta koja povezuje model pogled sa pogledom definisanim u xml datoteci. Android pruža tu mogućnost za sve verzije Androida počev od Androida 2.1 potrebno je samo omogućiti biblioteku za povezivanje podataka.



DIjagram 3 - Model-Pogled Model-Pogled

Ciljevi MVVM šablona su isti kao i ciljevi MVP šablona, to su razdvajanje odgovornosti, modularnost i testabilnost aplikacije. Razlika je samo u načinu na koji se to ostvaruje, kod MVVM šablona postoji automatsko propagiranje promena za razliku od MVP šablona što je u nekim slučajevima pogodno ali ne pruža toliku kontrolu nad promenama kao MVP. Takođe za razliku od MVP šablona može se koristiti jedan pogled model za više različitih pogleda i pogled model neće imati nikakvu referencu ka pogledu. Implementacija korišćena u aplikaciji je zasnovana na preporukama kompanije Google [6], naravno u zavisnosti od potreba implementacija može da varira.

Implementacija u aplikaciji

Neka se opet razmatra scenario opisan u specifikaciji aplikacije. Potrebno je na odabir filma iz liste odvesti korisnika na ekran koji prikazuje detalje filma. Pored toga, potrebno je omogućiti korisniku dodavanje filma u listu za kasnije gledanje ili u favorite, kao i mogućnost davanja sopstvene ocene filmu. Dati ekran i funkcionalnosti su implementirane koristeći MVVM šablon.

Za početak je potrebno omogućiti povezivanje podataka dodavanjem sledećih linija u `build.gradle` datoteku:

```
dataBinding {
   enabled = true
}
```

Aktivnost, pored XML datoteke koja opisuje izgled, predstavlja pogled. Odatle sledi da u aktivnosti je potrebno imati što manje logike, konkretno u aktivnosti je samo preuzet identifikacioni broj filma (potreban da bi se sa servisa učitali detalji filma), zatim kreiran pogled model i povezan sa xml datotekom koja opisuje izgled ekrana. Nakon toga se inicira učitavanje podataka iz pogleda modela. Pored toga u okviru pogleda je kontrolisan životni ciklus aktivnosti. Pogled je implementiran u okviru klase `MovieDetailsActivity`.

```
public class MovieDetailsActivity extends Activity {

    private MovieDetailsViewModel viewModel;
    private ActivityMovieDetailsBinding binding;
    private int movieId;


    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        movieId = getIntent().getIntExtra(Constants.MOVIE_ID, 0);

        if (viewModel == null) {
            viewModel = new MovieDetailsViewModel(movieId,
getApplicationContext());
        }

        binding = DataBindingUtil.setContentView(this,
R.layout.activity_movie_details);
        binding.setMovieDetailsViewModel(viewModel);
        viewModel.loadMovieDetails();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        viewModel.finishSubscriptions();
    }
}
```

U modelu pogledu su metode koje dohvataju podatke iz modela i izlažu ih tako da pogled može da ih koristi i da se promene automatski propagiraju. Pored toga u pogledu modelu su metode koje prihvataju korisničku interakciju i pozivaju metode u modelu (koji je putem Daggera umetnut u pogled model) koje npr. čuvaju film u lokalnoj bazi u tabeli omiljenih filmova. U okviru pogled modela nema poslovne logike, sva logika je usko vezana za sam prikaz. U nastavku je prikazan deo klase `MovieDetailsViewModel` kao i bitni delovi izgleda ekrana opisanih u `activity_movie_details.xml` datoteci.

```
public class MovieDetailsViewModel extends BaseObservable {
    private int movieId;
    private Context context;
    private Subscription subscription;
    private MovieDetails movieDetails;
    public final ObservableBoolean dataLoading = new ObservableBoolean(true);
    public final ObservableBoolean isFavourite = new
ObservableBoolean(false);
    public final ObservableBoolean isInWatchlist = new
ObservableBoolean(false);
```

```
    @Inject
    MovieDetailsModel model;


    public MovieDetailsViewModel(int movieId, Context context) {
        this.movieId = movieId;
        this.context = context;
        ((MovieMaster) context).getMovieDetailsComponent().inject(this);
        isFavourite.set(model.isMovieFavourite(movieId));
        isInWatchlist.set(model.isMovieInWatchlist(movieId));
    }

    public void loadMovieDetails() {
        if (movieDetails == null) {
            subscription = model.getMovieDetails(movieId)
                    .subscribe(new Subscriber<MovieDetails>() {
                        @Override
                        public void onCompleted() {
                            Log.i("INFO", "Completed");
                        }

                        @Override
                        public void onError(Throwable e) {
                            e.printStackTrace();
                        }

                        @Override
                        public void onNext(MovieDetails newMovieDetails) {
                            movieDetails = newMovieDetails;
                            dataLoading.set(false);
                            notifyChange();
                        }
                    });
        } else {
            dataLoading.set(false);
        }

    }

    public void finishSubscriptions() {
        if (!subscription.isUnsubscribed()) {
            subscription.unsubscribe();
        }
    }

    public void toggleFavourites(View view) {
        if (isFavourite.get()) {
            Toast.makeText(context,
context.getString(R.string.movie_deleted_from_favourites),
Toast.LENGTH_SHORT).show();
            model.deleteFromFavourites(movieId);
            isFavourite.set(false);
        } else {
```

```
            Toast.makeText(context,
context.getString(R.string.movie_added_to_favourites),
Toast.LENGTH_SHORT).show();
            model.addToFavourites(movieDetails);
            isFavourite.set(true);
        }
    }

    public void rateMovie(View view, float newRating, boolean fromUser) {
        model.rateMovie(movieId, newRating);
    }


    @Bindable
    public float getMyRating() {
        return model.getMyMovieRating(movieId);
    }


    @Bindable
    public String getName() {
        if (movieDetails != null) {
            return movieDetails.getName();
        }
        return "";
    }

    @Bindable
    public String getCast() {
        if (movieDetails != null && movieDetails.getCast() != null) {
            String cast = "";
            for (int i = 0; i < movieDetails.getCast().size(); i++) {
                cast += movieDetails.getCast().get(i);
                if (i != movieDetails.getCast().size() - 1) {
                    cast += ", ";
                }
            }
            return cast;
        }
        return "";
    }

    @Bindable
    public String getRating() {
        if (movieDetails != null) {
            return context.getResources().getString(R.string.rating)
                    + String.valueOf(movieDetails.getRating())
                    + "/10";
        }
        return "";
    }

    @Bindable
    public Drawable getDrawable() {
        if (movieDetails != null) {
```

```
            return new BitmapDrawable(context.getResources(),
movieDetails.getPosterBitmap());
        }
        return null;
    }
}
```

Ako se razmotri način povezivanja između podataka i prikaza, uočava se da su u modelu pogledu podaci izloženi na tri načina [7]. Prvi način je preko polja koja se posmatraju (eng. Observable fields) kao što je npr. `ObservableBoolean isFavourite`. Ono što je značajno je da ovo polje mora biti deklarisano kao javno i finalno jer se u pogledu detektuje samo promena u vrednosti polja a ne u samom polju [8]. Vrednost ovog polja se inicijalizuje prilikom kreiranja modela pogleda pozivanjem metoda iz modela koja proverava da li je film favorizovan, odnosno da li se nalazi u lokalnoj bazi omiljenih filmova. Takođe, ako korisnik klikne na dugme koje služi za dodavanje ili uklanjanje iz favorita poziva se metoda `public void toggleFavourites(View view)` koja pored uklanjanja ili dodavanja u omiljene filmove takođe i menja vrednost polja `isFavourite`. Promena vrednosti polja se automatski oslikava na pogledu. Relevantni elementi XML koda vezani za čitanje vrednosti promenljive kao i za pozivanje metode su dati u nastavku.

XML element gde je definisana promenljiva koja predstavlja pogled model:

```
    <data>
        <import type="android.view.View" />

        <variable
            name="movieDetailsViewModel"

type="com.master.movie.moviemaster.moviedetails.MovieDetailsViewModel" />
    </data>
```

XML element koji predstavlja dugme na čiji se klik dodaje ili briše film iz liste omiljenih. Lako se može primetiti da natpis na dugmetu zavisi od promenljive `isFavourite` i da se na klik poziva metoda iz pogleda modela:

```
        <Button
            android:id="@+id/add_to_favourites"
            android:layout_width="0dp"
            android:layout_height="48dp"
            android:layout_marginEnd="8dp"
            android:layout_marginLeft="24dp"
            android:layout_marginRight="8dp"
            android:layout_marginStart="24dp"
            android:layout_marginTop="16dp"
            android:background="@color/colorPrimary"
            android:onClick="@{movieDetailsViewModel::toggleFavourites}"
            android:text="@{movieDetailsViewModel.isFavourite ?
@string/remove_from_favourites : @string/add_to_favourites}"
            android:textColor="@color/white"
```

```
            android:textSize="14sp"
            app:layout_constraintHorizontal_bias="1.0"
            app:layout_constraintLeft_toRightOf="@+id/poster"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/rating" />
```

Drugi način je preko objekata koji se posmatraju (eng. Observable objects). Klasa koja implementira `Observable` interfejs dozvoljava vezivanje i osluškivanje objekta i promena bilo kog njegovog svojstva. Interfejs `Observable` poseduje mehanizme za dodavanje i uklanjanje osluškivanja objekta, ali samo obaveštavanje o promena nad objektom ostaje na programeru. Da bi se olakšao razvoj kreirana je klasa `BaseObservable` koja implementira mehanizme za registraciju osluškivanja objekta. Svaki objekat koji se posmatra ima svoju metodu za čitanje vrednosti (eng. getter) koja je obeležena anotacijom `@Bindable`.

Klasa `MovieDetailsViewModel` nasleđuje klasu `BaseObservable` i sadrži mnogo polja koja se posmatraju. Posmatrajmo na primer objekat `name` koje ima metodu za čitanje vrednosti označenu anotacijom `@Bindable`. Međutim kada korisnik dođe na stranicu sa detaljima filma, objekat nije inicijalizovan jer se tek u tom trenutku poziva REST servis koji vraća detalje filma u JSON formatu, iz tog razloga se u metodi `loadMovieDetails()` poziva metoda `notifyChange()` (moguće je pozivati i metodu `notifyPropertyChanged()` koja kao argument prima objekat i obaveštava samo o promenama tog objekta). Dat je XML element koji ispisuje ime filma u formatu `ime(godina)` gde je i godina povezana na isti način kao i ime filma.

```
            <TextView
                android:id="@+id/header"
                tools:text="Shawshank Redemption (1994)"
                android:text="@{movieDetailsViewModel.name + ` (` +
movieDetailsViewModel.year + ')'}"
                android:textAlignment="center"
                android:gravity="center"
                android:layout_gravity="center"
                android:textColor="@color/white"
                android:layout_width="wrap_content"
                android:layout_height="match_parent"
                android:textSize="21sp"/>
```

Treći način povezivanja je kroz kolekcije koje se posmatraju (eng. Observable Collections). Ove kolekcije su najsličnije mapama sa ključ-vrednost parovima i mogu da se posmatraju na sličan način kao i polja koja se posmatraju.

Model ima istu ulogu kao u MVP šablonu. Potpuno je nezavisan od prikaza i njegova glavna uloga je da sa servisa primi podatke o detaljima filma što izvršava metoda sa potpisom `Observable<MovieDetails> getMovieDetails(int movieId)`, slično kao i kod modela u MVP arhitekturi i ova metoda je implementirana koristeći RxJavu o kojoj će biti

reči u daljem radu. Pored toga sadrži i metode koje su su potrebne za komunikaciju sa bazom, kao što je dodavanje u listu omiljenih filmova, brisanje iz date liste, ocenjivanje filma itd. U nastavku su dati bitniji delovi koda modela.

```
public class MovieDetailsModel {

    DBHelper dbHelper;

    ApiServiceWrapper apiServiceWrapper;

    public MovieDetailsModel(ApiServiceWrapper apiServiceWrapper, DBHelper
dbHelper) {
        this.apiServiceWrapper = apiServiceWrapper;
        this.dbHelper = dbHelper;
    }

    public Observable<MovieDetails> getMovieDetails(int movieId) {
        return Observable.create((Observable.OnSubscribe<MovieDetails>)
subscriber ->
                MovieDetailsModel.this.getMovieDetailsSub(subscriber,
movieId))
                .doOnNext(movieDetails ->
apiServiceWrapper.getPoster(movieDetails))
                .observeOn(AndroidSchedulers.mainThread())
                .subscribeOn(Schedulers.io());
    }


    private void getMovieDetailsSub(Subscriber<? super MovieDetails>
subscriber, int movieId) {
        try {
            subscriber.onNext(apiServiceWrapper.getMovieDetails(movieId));
            subscriber.onCompleted();
        } catch (Exception e) {
            e.printStackTrace();
            subscriber.onError(e);
        }
    }

    public void addToFavourites(MovieDetails movieDetails) {
        dbHelper.insertMovieToFavourites(movieDetails);
    }

    public void rateMovie(int movieId, float newRating) {
        dbHelper.rateMovie(movieId, newRating);
    }

    public boolean isMovieFavourite(int movieId) {
        return dbHelper.isMovieFavourite(movieId);
    }

    public void deleteFromFavourites(int movieId) {
        dbHelper.removeFromFavourites(movieId);
    }
}
```

Zaključna razmatranja za MVVM i poređenje MVP i MVVM

MVVM je šablon koji je, kao i MVP, jako dobar za razdvajanje poslovne logike i podataka od prikaza. To doprinosi čistijem kodu, lakšem održavanju i lakšem testiranju. Naravno, ni ovaj šablon ne rešava sve probleme i ima svojih nedostataka. U odnosu na MVP šablon pozitivno je to što je povezivanje podataka i ažuriranje pogleda jednostavnije i u pogledu modelu ne postoji nikakva referenca ka pogledu, za razliku od prezentera koji ima referencu ka pogledu i gde je programer odgovoran za ažuriranje podataka. Takođe, eksplicitna komunikacija između pogleda i ostalih slojeva je praktično nepostojeća što donekle pojednostavljuje kod. Moguće je imati i više pogleda modela za jedan pogled što može biti korisno u pojedinim situacijama. Međutim MVVM zahteva uključivanje povezivanja podataka i može imati kod u okviru XML pogleda. Ovo može dovesti do gomilanja koda vezanog za prikaz u okviru XML datoteke. Pored toga što je taj kod “skriven” od programera, on je i praktično nemoguć za otkrivanje grešaka naročito alatima dostupnim u razvojnom okruženju. Iz tog razloga treba biti pažljiv i strogo izbegavati složeniju logiku u pogledu. Još jedna prednost MVP šablona je što programer ne mora imati iskustva sa bibliotekom za povezivanje podataka - to dozvoljava bržu početnu primenu implementacije šablona i pruža veću kontrolu nad podacima.

Bitno je istaći da oba šablona imaju svojih dobrih i loših strana, na kraju se izbor svodi na lične preferencije programera. Trenutno je trend okrenut više ka MVP šablonu. Međutim, mnogo bitnije od izbora šablona je praćenje smernica dobrog dizajna i jasno razgraničavanje slojeva aplikacije. Ako je to ispoštovano prilikom kreiranja onda će i koristi od ova dva šablona biti gotovo jednake. Moguće je čak, u zavisnosti od konkretne situacije, i koristiti MVP šablon za aktivnost aplikacije gde je konkretnu funkcionalnost lakše implementirati na taj način i MVVM za neku drugu aktivnost, kao što je urađeno u okviru aplikacije MovieMaster.

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 šablona (eng. Iterator pattern) kao i funkcionalnog programiranja [12]. 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. Pre prelaska na RxAndroid biće reči o Androidu i Javi 8 jer se RxAndroid oslanja na lambda izraze da bi kod bio čitljiviji.

Android i Java 8

Android koristi Java programski jezik za razvijanje aplikacija. Međutim Android ima sopstveni Android SDK[3] gde koristi Java sintaksu i semantiku ali ne prati Java SE[4] ili ME[5] standarde što rezultuje time da aplikacije napisane za te platforme i za Android platformu nisu kompatibilne. Pored toga Android nema sve klase i API-je koje su deo Jave SE ili ME dok poseduje dodatne API-je i klase potrebne za razvoj Android aplikacija. Takođe

[3] Skraćenica za Software Development Kit – skup računarskih alata koji se koriste za razvijanje softvera
[4] Java Standard Edition
[5] Java Micro Edition

koristi i sopstvenu virtuelnu mašinu - ART[6] ili, na starijim verzijama Androida, Dalvik virtuelnu mašinu. Android SDK trenutno uključuje podršku za većinu Java 7 funkcionalnosti ali nema zvanične podrške za Java 8 funkcionalnosti.

Do marta 2017. godine Google je omogućio programerima da koriste Jack kompilator [9] i alate koji idu uz njega da bi kompilirao Java 8 kod u Android dex bajtkod. Bilo je lako uključiti Jack kompilator u projekat međutim vreme kompilacije je bilo jako dugo što se pokazalo nepraktično pri svakodnevnom radu. Google je najavio da će u okviru Android Studija 3.0 dodati podršku za kompilaciju Java 8 koda dok je Jack kompilator proglašen zastarelim. Najbitnije funkcionalnosti Jave 8 koje će biti podržane u okviru Android Studija 3.0 su lambda izrazi, reference na metode, anotacije tipova, anotacije koje se ponavljaju i predefinisane i statičke metode interfejsa koje će biti podržane počev od Android API nivoa 24 [10]. Pored toga Java API-ji koji će biti podržani počev od Android API nivoa 24 su: `java.util.stream`, `java.util.function`, `java.lang.FunctionalInterface`, `java.lang.annotation.Repeatable`, `java.lang.reflect.AnnotatedElement.getAnnotationsByType(Class)`, `java.lang.reflect.Method.isDefault()`.

Budući da se u projektu koristi RxAndroid, koji se oslanja na Java 8 lambda izraze radi čitljivijeg zapisa, potrebno je rešenje koje je trenutno podržano i omogućava korišćenje Java 8 lambda izraza. Biblioteka koja omogućava to i koja će biti uključena u projekat je Retrolambda [11]. Da bi se biblioteka koristila potrebno je samo dodati sledeće linije u `gradle.build` fajl.

```
buildscript {
   repositories {
      mavenCentral()
   }

dependencies {
      classpath 'me.tatarka:gradle-retrolambda:3.6.1'
   }
}


repositories {
   mavenCentral()
}

apply plugin: 'com.android.application'
apply plugin: 'me.tatarka.retrolambda'

    compileOptions {
        targetCompatibility 1.8
        sourceCompatibility 1.8
    }
```

[6] Android Runtime

RxAndroid

Osnovni gradivni elementi reaktivnog programiranja su emiter[7] (eng. Observable) i pretplatnik[8] (eng. Subscriber) [13]. 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 su prikazani najbitniji koncepti i najkorišćeniji operatori u RxAndroidu. Emiter može biti kreiran na sledeći način:

```
    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. Pretplatnik se može kreirati na sledeći način:

```
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) { }
};
```

Kada se pretplatnik i emiter spoje dobija se odštampano `"Hello, world!"`.

```
myObservable.subscribe(mySubscriber);
```

[7] 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

[8] Subscriber se može prevesti i kao “osluškivač”

Kod može biti i jednostavniji, postoji operator `Observable.just()` koji emituje jedan objekat a zatim se završava isto kao u prethodni kod.

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

Može se pojednostaviti i kod vezan za pretplatnika. Jedino što je tu bitno je `onNext()` metoda koja može biti izdvojena.

```
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()` ali takođe može primiti i samo `onNext()` akciju. Kada se to uzme u obzir dobija se:

```
myObservable.subscribe(onNextAction);
```

Odnosno kada se spoje kodovi za emiter i pretplatnika u jedan izraz dobija se:

```
Observable.just("Hello, world!")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
              System.out.println(s);
        }
    });
```

A kada se još na ovaj izraz primene Java 8 lambda izrazi rezultat je:

```
Observable.just("Hello, world!")
.subscribe(s -> System.out.println(s));
```

Neka je potrebno nadovezati još jednu nisku na postojeću. To je moguće učiniti na više načina, na primer moguće je nadovezati nisku u emiteru ili u pretplatniku. Ali umesto nadovezivanja niske, zahtev je mogao da bude i bilo koja druga složenija transformacija emitovanih podataka. Iz tog razloga 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 moguće je 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));
```

Neka je data metoda koja vraća emiter liste url-ova na osnovu upita.

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

Potrebno je odštampati svaki od datih url-ova, to je moguće učiniti na sledeći način:

```
query("Hello, world!")
    .subscribe(urls -> {
        for (String url : urls) {
            System.out.println(url);
        }
    });
```

Međutim na ovaj način nije moguće koristiti operator `map()` za transformaciju url-ova, moguće je jedino transformisati u okviru pretplatnika ali to nije optimalno rešenje jer je uloga pretplatnika često samo da ažurira prikaz informacija i u tim situacijama se izvršava na glavnoj niti. Postoji operator `Observable.from()` koji prima kolekciju objekata i emituje ih jednog po jednog, tako da je moguće transformisati postojeći izraz u sledeći:

```
query("Hello, world!")
.subscribe(urls -> {
    Observable.from(urls)
        .subscribe(url -> System.out.println(url));
});
```

Na ovaj način su dobijeni ugnježdeni emiteri 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 se taj operator primeni na postojeći slučaj dobije se:

```
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 se to pojednostavi lambda izrazima dobija se:

```
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 se dolazi do koristi koje donosi. Ako se pretpostavi da postoji metoda `Observable<String> getTitle(String URL);` koja na osnovu url-a vraća naziv vebsajta ili `null` ako sajt nije pronađen. Ako bi se ovaj problem rešavao na tradicionalni način postojao bi asinhroni poziv koji bi učitavao listu url-ova a zatim u okviru njega ugnježdene nove asinhrone pozive za naslov svakog od url-ova. Korišćenjem RxAndroida izbegavaju se sinhronizacije više poziva i svi probleme koji mogu tu da se jave. Sa RxAndroidom je:

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .subscribe(title -> System.out.println(title));
```

Neka je još potrebno izbaciti `null`, prikazati samo prvih 5 naslova i pritom ih sačuvati na disku. Sve je to moguće uraditi sledećim izrazom:

```
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()` omogućava da se svaki put kada se objekat emituje izvrši neka akcija, u ovom slučaju čuvanje naziva.

Značajno je pomenuti da ako bilo koja od metoda u operatorima izbacuje potencijalnu grešku, u samim operatorima nije potrebno hvatati i reagovati na grešku već je dovoljno to učiniti u `Subscriber.onError()` metodi. Na taj način dobijeno je centralizovano mesto za hvatanje i reagovanje na greške. Ono što je takođe korisno je to što operator `Observable.subscribe()` vraća objekat tipa `Subscription`, ako je iz bilo kog razloga potrebno da pretplatnik više ne prati ono što emiter emituje moguće je 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 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 slične operacije koje mogu da traju određeni vremenski period, ne obavljaju na glavnoj niti kako 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, dok sa druge strane operator `observeOn()` određuje nit na kojoj će se slati notifikacije pretplatniku. Uobičajno je da se celo izračunavanje i komunikacija sa mrežom obavljaju na niti za ulaz/izlaz dok se slanje notifikacija 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. 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 koji se javlja u vezi životnog ciklusa 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 otpadaka 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

U aplikaciji MovieMaster potrebno je prvo učitati sa mreže listu filmova a zatim za svaki od filmova iz liste učitati sa statičke adrese na serveru poster za taj film. Pored toga potrebno je i keširati podatke i da bi se na taj način izbegli nepotrebni i skupi pozivi ka servisu.

Prvo se razmatra 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 [14]. 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 se vidi iz priloženog, izvršavanje asinhronog poziva je prilično složeno u Androidu. U posmatranom slučaju bilo bi potrebno izvršiti asinhroni poziv da bi se dobili svi filmovi a zatim bi se u `onPostExecute(Result)` metodi za svaki film u petlji pravio još po jedan asinhroni poziv za učitavanje 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 bi se promenio scenario i dodao treći poziv zavisan od prethodnog, dobili bismo još jedan nivo ugnježdenosti i još teži kod za održavanje.

Pomoću RxAndroida rešenje je mnogo jednostavnije, u klasi `MainMovieListModel` imamo metodu `loadMovies()` koja služi za učitavanje liste filmova koji će biti prikazani.

```
public Observable<List<Movie>> loadMovies(String query) {
    if (cachedMovies != null) {
```

```
            return Observable.create((Observable.OnSubscribe<List<Movie>>)
this::getMoviesFromCache)
                    .flatMap(Observable::from)
                    .filter(movie -> filterMovies(movie, query))
                    .toList()
                    .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());
        }
    }
```

Ako se posmatra prvo else grana 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);
                }
            })
```

Ovde metoda `void getMovies(Subscriber<? super List<Movie>> subscriber)` 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);
        }
    }
```

Pri tome, u `apiServiceWrapper.getAllMovies()` metodi je enkapsulirana logika slanja zahteva preko mreže, obrade odgovora i vraćanja liste filmova. Na taj način je kreiran emiter koji emituje listu filmova.

Zatim se koristi `flatMap(Observable::from)`, ova dva operatora u kombinaciji su već objašnjena, i njihov konačni efekat je da razdvoje emitovanje liste filmova u emitovanje jednog po jednog filma. Svaki emitovani objekat klase `Movie` zatim prima `map()` operator i učitava sa mreže njegov poster korišćenjem operatora `map(movie -> apiServiceWrapper.getPoster(movie))`. Pošto metoda vraća emiter liste filmova a ne listu emitera koristi se operator `toList()` da bi se spojili emiteri natrag u listu. Zatim se keširaju podaci za svako emitovanje 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 se koristi operator `observeOn(AndroidSchedulers.mainThread())` koji određuje da će se notifikacije pretplatniku slati na glavnoj niti i operator `subscribeOn(Schedulers.io())` koji određuje da će emiter vršiti emitovanje i obavljati transformacije na niti predviđenoj za operacije u pozadini. Uprošćeno moglo bi se reći da je pomoću RxAndroida u samo nekoliko redova zapisano sledeće: učitati listu filmova sa mreže, razdvojiti ih u pojedinačne objekte, za svaki objekat učitati poster sa mreže, spojiti ih ponovo u listu i keširati ih, sve to uraditi u pozadini, obaveštenja slati na glavnoj niti.

If grana metode je vrlo slična, jedine razlike su što se kompletno formirana lista dobija iz memorije a ne sa mreže i što je moguće vršiti pretragu nad keširanom listom filtriranjem ako se prosledi argument `query`. U produžetku su date metode `void getMoviesFromCache(Subscriber<? super List<Movie>> subscriber)` i `private boolean filterMovies(Movie movie, String query)`.

```
private void getMoviesFromCache(Subscriber<? super List<Movie>> subscriber) {
        try {
            subscriber.onNext(cachedMovies);
            subscriber.onCompleted();
        } catch (Exception e) {
            e.printStackTrace();
            subscriber.onError(e);
        }
    }


    private boolean filterMovies(Movie movie, String query) {
            if (query.isEmpty()) {
                return true;
            } else if
      (movie.getName().toLowerCase().contains(query.toLowerCase())) {
                return true;
            }
            return false;
        }
```

Pretplatnik se nalazi u klasi `MainMovieListPresenter` i ima ulogu da pri svakom emitovanju sakrije animaciju za učitavanje i pozove metodu pogleda da ažurira prikaz. U slučaju greške ili završenog emitovanja štampa 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);
                }
            });
}
```

Zaključna razmatranja za reaktivno programiranje

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 da i ovaj pristup ima svojih mana, promena razmišljanja sa imperativne na reaktivnu paradigmu inicijalno zahteva puno vremena i rada. Pored toga za jednostavne zadatke kao i neke specifične slučajeve nije pogodno koristi RxAndroid. Takođe postoje i problemi koji se ne javljaju u imperativnom programiranju. Na primer, situacija gde emiter emituje brže nego što operator ili pretplatnik mogu da obrađuju podatke vodi do problema specifičnog za reaktivno programiranje.

Umetanje zavisnosti u Androidu

Prilikom dizajniranja arhitekture aplikacije potrebno je voditi računa o tome da funkcionalno različiti delovi aplikacije ne budu usko spregnuti odnosno da budu što nezavisniji. Inverzija kontrole je način projektovanja aplikacije koji omogućava nezavisnost između različitih modula aplikacije. Jedan od načina ostvarivanja inverzije kontrole je umetanje zavisnosti (eng. dependency injection).

Nek je dat objekat neke klase i servisi od kojih taj objekat zavisi (servisi mogu biti bilo koje klase koje su potrebne objektu). Servisi bi se mogli kreirati u objektu, što bi dovelo do spregnutosti i ponavljanja koda, otežanog testiranja i potrebe da se menja kod klase ukoliko se promeni implementacija servisa. Umetanje zavisnosti je tehnika pri kojoj se konkretne implementacije servisa prosleđuju objektima prilikom kreiranja ili inicijalizacije objekata koje zavise od servisa.

U Androidu je moguće postići umetanje zavisnosti na više načina. Trenutno najkorišćenije biblioteka za ostvarivanje umetanja zavisnosti u Androidu je Dagger 2. Pre toga je bio popularan radni okvir Roboguice [15] koji je koristio refleksiju za postizanje umetanja zavisnosti.

Dagger 2

Dagger je statički radni okvir za Javu i Android i služi za umetanje zavisnosti gde se kod neophodan za umetanje zavisnosti generiše u toku kompilacije. Prvobitno je razvijen u kompaniji Square a trenutno ga održava kompanija Google. [16].

Da bi bilo moguće umetanje željenog objekta u neki drugi objekat, Dagger mora da „zna“ kako se konstruiše taj objekat, odnosno koji objekti su potrebni da bi se kreirao željeni objekat. Moduli su klase koje definišu i obezbeđuju zavisnosti[9], zavisnosti treba da budu grupisani po modulima tako da je svaki modul jedna logička celina. Moduli su klase u Daggeru označeni anotacijom `@Module`, dok su zavisnosti date kao povratni argumenti metoda koje su anotirane sa `@Provides`. Na primer:

```
@Module
public class SubsystemModule {
    @Provides
    public SubsystemPartOne provideSubsystemPartOne() {
        return new SubsystemPartOneImplementation();
    }
}
```

[9] eng. Dependencies – objekti koje je potrebno umetnuti

U primeru je kao zavisnost data samo klasa `SubsystemPartOne` koja ima konstruktor koji ne sadrži nijedan argument, što naravno nije realna situacija. Neka klasa `SubsystemPartOne` kao argument prima objekat klase `SubsystemPartTwo`, dovoljno je da se u modulu navede kako se gradi objekat klase `SubsystemPartTwo`. U slučaju da je potrebno konstruisati objekat klase `SubsystemPartOne` Dagger će na osnovu metode za izgradnju objekta klase `SubsystemPartTwo` moći prvo da konstruiše taj objekat a zatim da ga prosledi da bi izgradio traženi objekat. Ukoliko ima i više zavisnosti, Dagger će automatski da detektuje redosled kojim treba da konstruiše objekte.

```
@Module
public class DependencyModule {
    @Provides
    public Dependency provideDependency(
                SecondaryDependency secondaryDependency) {
        return new DependencyImplementation(secondaryDependency);
    }

    @Provides
    public SecondaryDependency provideSecondaryDependency() {
        return new SecondaryDependencyImplementation();
    }
}
```

Pored statičkih objekata moguće je proslediti i objekat koji je dostupan tek u vreme izvršavanja programa.

```
@Module
public class DependencyModule {
    public RuntimeParameter runtimeParameter;
    public DependencyModule(RuntimeParameter runtimeParameter) {
        this.runtimeParameter = runtimeParameter;
    }
    @Provides
    public Dependency provideDepedency() {
        return new Dependency(runtimeParameter);
    }
}
```

Proces pri kome se razrešavaju zavisnosti u iole većem projektu na kraju vodi ka mreži međuzavisnosti, ova mreža se naziva graf zavisnosti. Kada je potrebno konstruisati i umetnuti neki objekat, on se konstruiše pomoću ovog grafa i vraća traženi objekat. U Daggeru jedan modul često ne sadrže sve sekundarne zavisnosti potrebne da bi se kreirao objekat koji taj modul obezbeđuje. Drugi moduli takođe mogu sadržati zavisnosti potrebne za kreiranje traženog objekta. Da bi moduli mogli da obezbeđuju jedni drugima zavisnosti neophodne su komponente. Interfejs ili apstraktna klasa označena anotacijom `@Component`, uz navedeni skup modula koji su povezani sekundarnim zavisnostima, omogućava automatsko generisanje komponente odnosno klase koja služi za obezbeđivanje svih zavisnosti koje su zahtevane. Generisana klasa ima naziv kao

komponenta samo sa prefixom Dagger. Komponente sadrže metode koje obezbeđuju zavisnosti (eng. provision methods) ili metode umetanja članova (eng. members-injections contracts) [17].

Metode koje obezbeđuju zavisnosti nemaju ulazne parametre i vraćaju objekat klase koja se umeće. Dat je primer komponente koja sadrži ovakvu metodu:

```
@Component(modules = {DependencyModuleOne.class, DependencyModulesTwo.class}
public interface DependencyComponent {
    ModuleOneDependency moduleOneDependency();
}
```

Opisani interfejs generiše klasu `DaggerDependencyComponent` koja koristi konstrukcioni obrazac graditelj [4] (eng. Builder pattern) za konstruisanje komponente. Pozivanje `builder()` metode je neophodno samo ukoliko su potrebni parametri koji su dostupni u vreme izvršavanja za izgradnju nekog od modula. U sledećem primeru je predstavljena konstrukcija objekta klase `DaggerDependencyComponent` i dohvatanje instance klase `ModuleOneDependency`.

```
DependencyComponent dependencyComponent =
            Dagger_DependencyComponent.builder()
                .dependencyModuleTwo(new DependencyModuleTwo(
                RuntimeParameter runtimeParameter)
                .build();

ModuleOneDependency moduleOneDependency = dependencyComponent
            .moduleOneDependency();
```

Kod metode umetanja članova, metode u komponenti imaju samo jedan parametar i zavisnosti se automatski umeću u sva polja sa anotacijom `@Inject` prosleđenog parametra.

```
@Component()
public interface DependencyComponent {
    void inject(DependencyReceiver dependencyReciever);
}

public class DependencyReceiverActivity extends AppCompatActivity {
  @Inject
  Dependency dependency;

 @Override
    public void onCreate(Bundle savedInstanceState) {
        DaggerDependencyComponent.Builder()
                .build().inject(this);
    }
}
```

Podkomponente su komponente čija je implementacija generisana unutar roditeljske komponente i nasleđuje ceo graf zavisnosti. Podkomponenta se definiše kao bilo koja komponenta a zatim se roditeljskoj komponenti dodaje metod koji prima module podkomponente i vraća podkomponentu.

```
@Component(
    modules = {...}
)
public interface DependencyComponent {
    DependencyOne dependencyOne();
    DependencySubcomponent dependencySubcomponent(DependencyModule
dependencyModule);
}
```

Samo umetanje zavisnosti se vrši isto kao kod komponenti, pri čemu je bitno istaći da je moguće imati i podkomponente podkomponenata i izgraditi jako složen graf zavisnosti na taj način.

Aleternativa podkomponentama je navođenje zavisnosti komponente od druge komponente. Ovaj pristup omogućava veću kontrolu nad zavisnostima jer je moguć pristup samo zavisnostima koje su deklarisane u metodama koje obezbeđuju zavisnost (eng. provision methods).

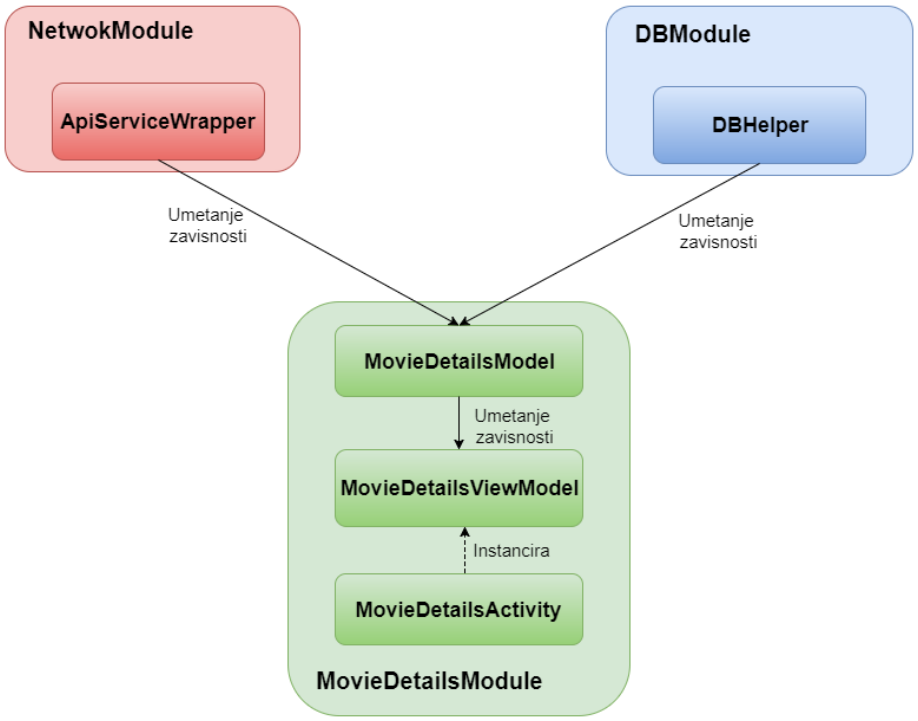
Jedna od funkcionalnosti Daggera koja takođe omogućava veću kontrolu nad kreiranjem i dostupnošću zavisnosti je obim zavisnosti (eng. scope). Obim zavisnosti se odnosi na dostupnost neke zavisnosti u kodu, odnosno držanja jedne instance zavisnosti u memoriji dok je potrebna. U Daggeru je definisan obim `@Singleton` koji označava da će samo jedna instanca anotirane klase postojati na aplikacionom nivou (dok god se aplikacija izvršava). Moguće je definisati i druge obime npr. obim određenog korisnika ili obim određene aktivnosti, gde će anotirane klase postojati samo dok je korisnik prijavljen ili aktivnost postoji.

Implementacija u aplikaciji

U poglavlju “Arhitektura Android aplikacije” je ilustrovana podela na module unutar Android aplikacije i ti moduli predstavljaju logičke celine i Dagger module. U radu će biti prikazano umetanje zavisnosti vezano za *MovieDetailsModule*, na sličan način je izvršeno umetanje zavisnosti i za ostale module.

Kada korisnik pristupi ekranu sa detaljima filma, pogled inicijalizuje pogled model. Pogled model ima umetnuti model (gde je poslovna logika i pristup podacima) dok model

ima umetnute klase iz *NetworkModule*-a i *DBModule*-a koje mu omogućavaju da pristupa podacima sa servisa i iz lokalne baze.



DIjagram 4 - Umetanje zavisnosti prikazano za MovieDetailsModule

Prvo se razmatra *NetworkModule* iz koga je izostavljen deo koda nevezan za Dagger:

```
@Module
public class NetworkModule {

    @Provides
    @Singleton
    ApiServiceWrapper provideApiServiceWrapper() {
        Retrofit retrofit = provideRetrofit();
        ApiService apiService =
retrofit.create(ApiService.class);
        return new ApiServiceWrapper(apiService);
    }
}
```

Modul izlaže klasu `ApiServiceWrapper` koja je unikat (eng. singleton) na nivou aplikacije i ne zavisi ni od kakvih parametara koji su dostupni u vreme izvršavanja programa. U klasi je implementirana komunikacija i primanje podataka od servisa.

Nasuprot tome klasa `DBHelper` a samim tim i *DBModule* zavise od konteksta koji je dostupan u vreme izvršavanja programa pa modul ima eksplicitno naveden konstruktor:

```
@Module
public class DBModule {

    private Application context;

    public DBModule(Application context) {
        this.context = context;
    }

    @Provides
    @Singleton
    DBHelper provideDBHelper() {
        return new DBHelper(context);
    }
}
```

MovieDetailsModule obezbeđuje *MovieDetailsModel* koristeći konstruktor koji kao argumente prima `ApiServiceWrapper` i `DBHelper` iz *NetworkModule*-a i *DBModule*-a.

```
@Module
public class MovieDetailsModule {
    @Provides
    @Singleton
    MovieDetailsModel provideMovieDetailsModel(ApiServiceWrapper
apiServiceWrapper, DBHelper dbHelper) {
        return new MovieDetailsModel(apiServiceWrapper, dbHelper);
    }
}
```

Na kraju je potrebno sve potrebne module povezati u komponentu i omogućiti umetanje instance klase `MovieDetailsModel` u `MovieDetailsViewModel` klasu korišćenjem metode umetanja članova (@Inject anotacije). Dat je kod `MovieDetailsComponent` klase:

```
@Singleton
@Component(modules={MovieDetailsModule.class, NetworkModule.class,
DBModule.class})
public interface MovieDetailsComponent {
    void inject(MovieDetailsViewModel viewModel);
}
```

U klasi `MovieMaster` komponenta je inicijalizovana kodom:

```
movieDetailsComponent = DaggerMovieDetailsComponent.builder()
        .dBModule(new DBModule(this))
        .build();
```

Zaključna razmatranja za Dagger 2

Dagger 2 je trenutno najkorišćeniji radni okvir za umetanje zavisnosti. Ono što generalno nudi umetanje zavisnosti je modularnost koja se slaže sa principima i MVP i MVVM šablona, zatim lakše testiranje i nezavisnost između modula. Što se tiče Daggera on je u odnosu na radne okvire koje koriste refleksiju brži i koristi manje resursa uređaja [18]. Sa obzirom da se radi o mobilnim uređajima sa ograničenim resursima to je jako bitna karakteristika. Mana je, u odnosu na radne okvire koji koriste refleksiju, to što je potrebno da programer sam definiše odnose i način izgradnje grafa zavisnosti, što oduzima vreme i dodaje kod. Međutim, u najvećem broju situacija rezultat je vredan toga.

Pomoćne biblioteke za razvoj Android aplikacija

Butter Knife

Butter Knife je biblioteka koja se koristi za povezivanje polja i metoda sa XML komponentama. Za povezivanje koristi procesiranje anotacija za automatsko generisanje koda koji se ponavlja na mnogim mestima. [19] Procesiranje se vrši pretraživanjem XML elemenata a ne refleksijom, što za rezultat ima dobre performanse. Biblioteka je jednostavna za uključivanje u projekat i korišćenje, potrebno je samo dodati u `build.gradle` datoteku:

```
compile 'com.jakewharton:butterknife:8.6.0'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.6.0'
```

Da bi Butter Knife mogao da pristupi i poveže polja i metode u trenutnoj aktivnosti potrebno je nakon postavljanja XML pogleda povezati Butter Knife na sledeći način:

```
ButterKnife.bind(this);
```

Nakon toga moguće je pristupiti elementima u XML pogledu preko anotacija ili dodati direktno akciju na klik bez potrebe za anonimnom unutrašnjom klasom. Na primer umesto:

```
EditText searchBox = (EditText) findViewById(R.id.search_box);
```

Dobija se:

```
@BindView(R.id.search_box)
EditText searchBox;
```

Ili umesto:

```
 findViewById(R.id.watchlist)
.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(this, CustomListActivity.class);
                intent.putExtra(Constants.TYPE, Constants.WATCHLIST);
                startActivity(intent);
            }
        });
```

Dobija se:

```
@OnClick(R.id.watchlist)
public void gotoWatchlist(){
    Intent intent = new Intent(this, CustomListActivity.class);
    intent.putExtra(Constants.TYPE, Constants.WATCHLIST);
```

```
        startActivity(intent);
    }
```

Jasno je da je druga varijanta mnogo čitljivija i jednostavnija. Pored toga, moguće je jednostavno vezati više pogleda za istu akciju ili dodati anotaciju `@Nullable` za polja ili `@Optional` za metode i na taj način izbeći izbacivanje izuzetaka ukoliko element XML pogleda nije pronađen [20].

Stetho

U toku razvijanja aplikacije javlja se potreba za jednostavnim i preglednim načinom ispitivanja zahteva poslatih preko mreže, stanja lokalne baze i pregleda prikazanih komponenti. Stetho je alat koji je razvila kompanija Facebook koji omogućava programerima pristup i korišćenje Google Chrome alata za programere nad Android aplikacijama koje razvijaju [21]. Stetho se jednostavno uključuje u projekat, dovoljno je dodati sledeće linije u `build.gradle` datoteku:

```
compile 'com.facebook.stetho:stetho:1.3.1'
compile 'com.facebook.stetho:stetho-okhttp3:1.3.1'
```

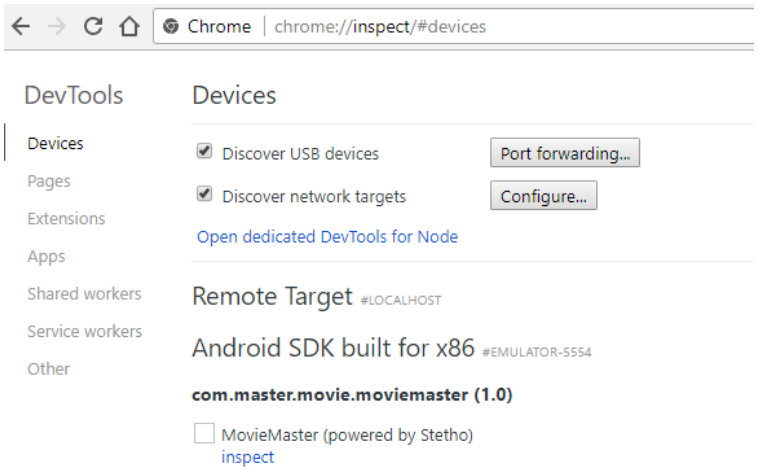
Pored toga, potrebno je inicijalizovati Stetho što se u okviru aplikacije MovieMaster izvršava odmah nakon startovanja aplikacije u klasi `MovieMaster` koja nasleđuje klasu `Application`.

```
Stetho.initializeWithDefaults(this);
```

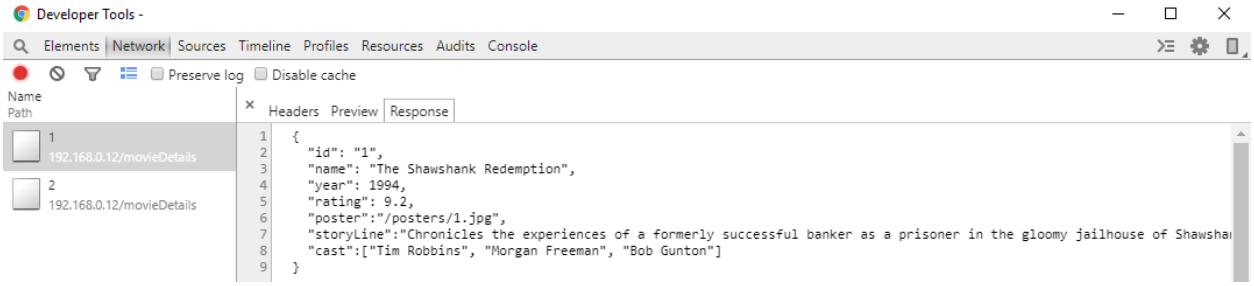
Takođe da bi se pratile aktivnosti na mreži potrebno je dodati presretač zahteva (eng. Network Interceptor). Tako da se u klasi `NetworkModule` postavlja sledeći kod:

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
    .readTimeout(30, TimeUnit.SECONDS)
    .connectTimeout(30, TimeUnit.SECONDS)
    .retryOnConnectionFailure(false)
    .addNetworkInterceptor(new StethoInterceptor())
    .build();
```

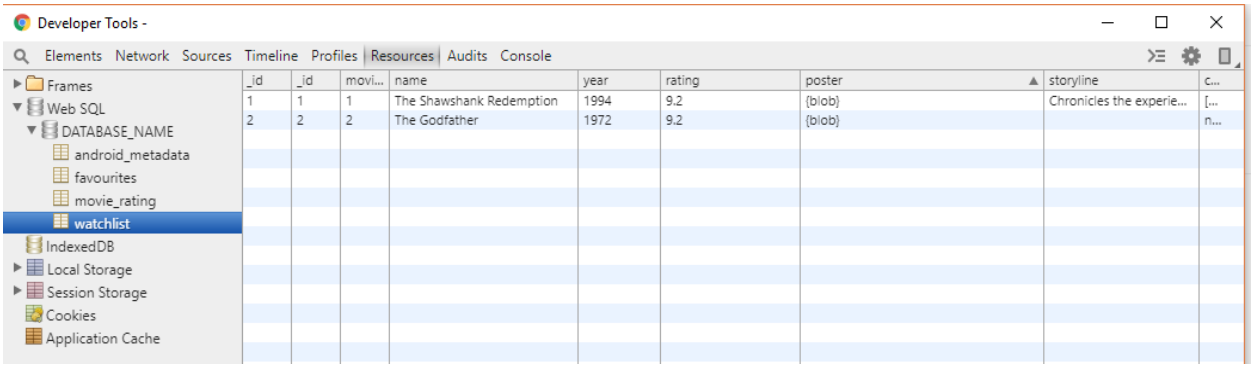
Praktično uz par linija koda se dobija moćan alat za praćenje baze i poziva na mreži preko alata internet pregledača Google Chrome. U praksi to izgleda ovako:



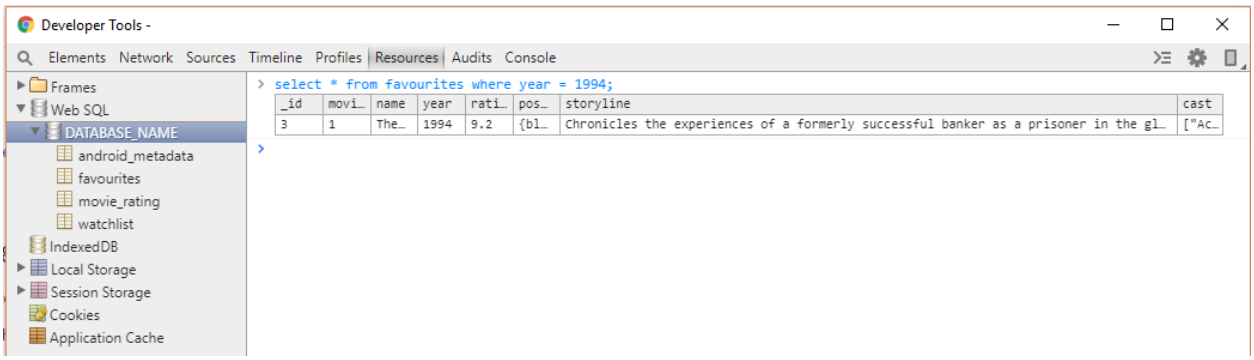
Slika 5 - Prikaz aplikacije u okviru Google Chrome pretraživača



Slika 6 - Prikaz komunikacije sa serverom preko mreže



Slika 7 - Prikaz baze korišćenjem Google Chrome alata za programere



Slika 8 - Prikaz izvršavanja upita iz konzole

Fabric

Fabric je mobilna platforma sa modulima koji se kombinuju da pruže različite funkcionalnosti [22]. Neki od značajnijih modula su:

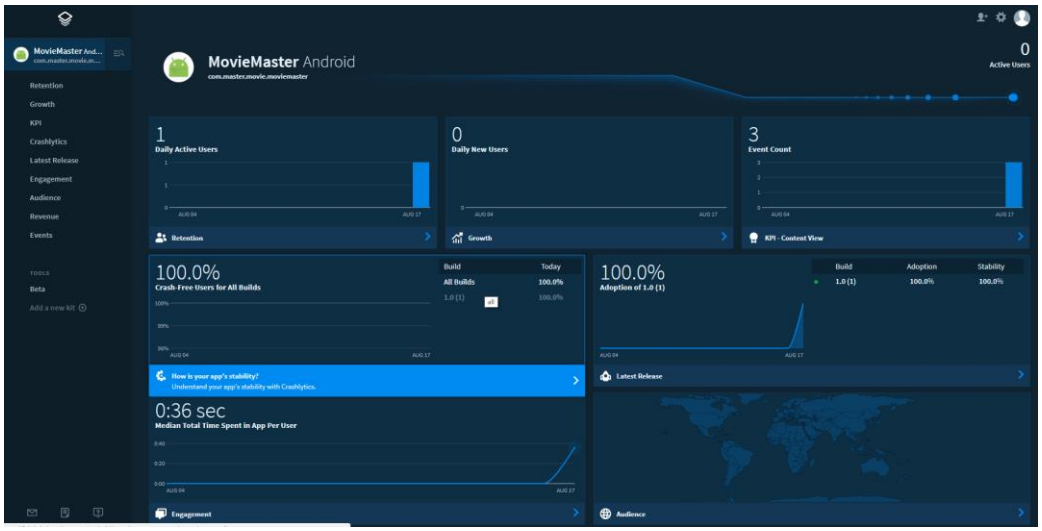
- **Crashlytics** – modul koji služi za analizu problema i grešaka u aplikaciji, svaki abnormalan prestanak rada aplikacije se šalje serverima Fabrica koji dalje organizuju podatke u celine pogodne za prikaz. Takođe programer može da zabeleži svaki uhvaćeni izuzetak.
- **Answers** – modul za upisivanje u dnevnik korisničkih akcija u aplikaciji, npr. upisivanje šta je korisnik pretraživao, da li je obavljao kupovinu preko aplikacije, koje ekrane je najviše posećivao itd.
- **Beta** – distribucija beta verzije aplikacije
- **Digits** – omogućavanje prijavljivanja korisnika u aplikaciju koristeći samo broj telefona

Pored ovih modula ponuđeni su moduli koji nisu razvijeni od strane Fabric tima.

Uključivanje Fabrica u projekat je jednostavno, potrebno je samo pratiti instalaciju Fabric-ovog dodatka za Android Studio koji će automatski generisati potreban kod. Takođe potrebno je registrovati se i dodati aplikaciju na https://fabric.io. U aplikaciji MovieMaster Fabric se inicijalizuje odmah pri startovanju aplikacije u klasi `MovieMaster` koja nasleđuje klasu `Application`.

```
Fabric.with(this, new Crashlytics());
```

Fabric upisuje u dnevnik broj korisnika, opšte podatke o njima kao i abnormalan prestanak rada aplikacije bez intervencije programera i u veb aplikaciji https://fabric.io prikazuje te podatke na pregledan način.



Slika 9 - Prikaz početnog ekrana Fabric platforme

U aplikaciji se upisuje u dnevnik npr. događaj dodavanja u omiljene filmove na sledeći način:

```
public void addToFavourites(MovieDetails movieDetails) {
        dbHelper.insertMovieToFavourites(movieDetails);
        Answers.getInstance().logContentView(new ContentViewEvent()
                        .putCustomAttribute("Added to favourites",
    movieDetails.getName()));
    }
```

Kada korisnik klikne na dugme “Dodaj u omiljene” u pozadini će se to upisati na Fabric-ov server i programer može to videti na veb aplikaciji https://fabric.io pod sekcijom events.

Upis u dnevnik uhvaćenih izuzetaka (eng. Exceptions) se vrši samo jednom linijom koda:

```
@Override
public void onError(Throwable e) {
    e.printStackTrace();
    Crashlytics.logException(e);
}
```

Upisani izuzeci biće jasno razdvojeni od izuzetaka koji nisu “uhvaćeni” i koji su automatski upisani, pri čemu nije bilo oporavka od greške.

Postoji još mnoštvo funkcionalnosti koje Fabric pruža ali u ovom radu su prikazane samo najkorišćenije. Prednost Fabrica u odnosu na slične alate, uključujući i Google Analytics, je u minimalnoj implementaciji od strane programera u aplikaciji, automatski upis u dnevnik svakog abnormalnog prestanka rada aplikacije kao i centralizovanog mesta za pregled svih značajnih podataka o korišćenju aplikacije.

Zaključak

U radu su predstavljeni arhitekturalni šabloni, dobre prakse i radni okviri koji olakšavaju razvijanje Android aplikacija. Akcenat je bio na tome da se analiziraju i uporede aktuelne tehnologije i da se kreira aplikacija koja simulira realan scenario na kome to može biti prikazano. Pored toga, predstavljeno je korišćenje arhitekturalnih šablona i dobrih praksi koje doprinose tome da aplikacija bude jednostavna za održavanje, izmene i dodavanje funkcionalnosti. Prikazano je i kako se reaktivna paradigma uklapa u tradicionalni razvoj Java aplikacija i koje su njene primene u Androidu. Naravno, svaki od šablona i tehnologija ima i svojih mana koje su već istaknute zato je za svaku aplikaciju potrebno razmotriti i neka alternativna rešenja ili modifikovanje rešenja predstavljenih u ovom radu.

Razvoj Android aplikacija je prešao dugačak put od kada je operativni sistem prvi put predstavljen. Tehnologije prikazane u ovom radu su trenutna dostignuća u ovoj oblasti, ali Android nije i dalje dostigao punu zrelost kao platforma tako da se mogu očekivati nove promene u razvoju Android aplikacija u narednih nekoliko godina. Postoje sve jači zagovornici Kotlin programskog jezika umesto Jave pri razvijanju Android aplikacija. U maju 2017. Na Google I/O konferenciji je objavljeno da Google dodaje Kotlin pored Jave kao zvanični jezik za razvijanje Android aplikacija i da će u razvojnom okruženju Android 3.0 biti uključena puna podrška za Kotlin [23]. Vreme će pokazati da li će Kotlin zaista da uspe da zameni Javu za razvoj Android aplikacija. Takođe postoji i mišljenje da će Android aplikacije u sadašnjoj formi nestati i da će postojati samo veb aplikacije koje su optimizovane i za telefone. I sada postoje takve aplikacije ali trenutno je njihov kvalitet i performanse mnogo lošije od aplikacija pisanih u Javi. Sa daljim razvojem hardvera i internet pregledača nije nemoguće zamisliti da se ovaj scenario ostvari. Postoje i aplikacije koje su pisane u Javascript radnom okviru React Native koji omogućava da razvijena aplikacija radi i na iOS operativnom sistemu i Androidu. Mišljenje autora je da je trenutno kvalitet takvih aplikacija takođe lošiji od aplikacija pisanih u Javi i vizuelno se takve aplikacije ne uklapaju dobro ni u iOS ni u Android operativni sistem. U svakom slučaju, realno je očekivati u skorijoj budućnosti velike izmene u razvoju Android aplikacija i tehnologijama koje se koriste.

Literatura

[1] "Intent," Google, [Na mreži]. Dostupno na: https://developer.android.com/reference/android/content/Intent.html. [Poslednji pristup 20.8.2017.].

[2] J.-P. Boodhoo, "Model View Presenter," *MSDN Magazine,* Avgust 2006.

[3] "Android Architecture todo-mvp," Google, [Na mreži]. Dostupno na: https://github.com/googlesamples/android-architecture/tree/todo-mvp/. [Poslednji pristup 26.8.2017.].

[4] R. H. R. J. J. V. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software., Addison Wesley, 1994.

[5] J. Smith, "Patterns - WPF Apps With The Model-View-ViewModel Design Pattern," *MSDN Magazine,* 2009.

[6] "Android Architecture MVVM databinding," Google, [Na mreži]. Dostupno na: https://github.com/googlesamples/android-architecture/tree/todo-mvvm-databinding/. [Poslednji pristup 28.8.2017.].

[7] "Data binding library," Google, [Na mreži]. Dostupno na: https://developer.android.com/topic/libraries/data-binding/index.html. [Poslednji pristup 14.8.2017.].

[8] "ObservableBoolean," Google, [Na mreži]. Dostupno na: https://developer.android.com/reference/android/databinding/ObservableBoolean.html. [Poslednji pristup 13.8.2017.].

[9] "Use Java 8 language features," Google, [Na mreži]. Dostupno na: https://developer.android.com/guide/platform/j8-jack.html. [Poslednji pristup 27.8.2017.].

[10] "Use Java 8 language features," Google, [Na mreži]. Dostupno na: https://developer.android.com/studio/write/java8-support.html. [Poslednji pristup 27.8.2017.].

[11] "Retrolambda," [Na mreži]. Dostupno na: https://github.com/orfjackal/retrolambda. [Poslednji pristup 27.8.2017.].

[12] "ReactiveX," [Na mreži]. Dostupno na: http://reactivex.io/. [Poslednji pristup 27.8.2017.].

[13] D. Lew, "Grokking RxJava," 15.9.2014. [Na mreži]. Dostupno na: http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/. [Poslednji pristup 27.8.2017.].

[14] "AsyncTask," Google, [Na mreži]. Dostupno na: https://developer.android.com/reference/android/os/AsyncTask.html. [Poslednji pristup 27.8.2017.].

[15] "Roboguice," [Na mreži]. Dostupno na: https://github.com/roboguice/roboguice. [Poslednji pristup 28.8.2017].

[16] "Dagger," 16.7.2017. [Na mreži]. Dostupno na: https://google.github.io/dagger/. [Poslednji pristup 28.8.2017].

[17] "Component," [Na mreži]. Dostupno na: https://google.github.io/dagger/api/2.0/dagger/Component.html. [Poslednji pristup 18.7.2017].

[18] A. Krasov, "Comparing the Performance of Dependency Injection Libraries," 7 3 2016. [Na mreži]. Dostupno na: http://blog.nimbledroid.com/2016/03/07/performance-of-dependency-injection-libraries.html. [Poslednji pristup 22.8.2017.].

[19] J. Wharton, "Butter Knife," [Na mreži]. Dostupno na: https://github.com/JakeWharton/butterknife/blob/master/README.md. [Poslednji pristup 18.8. 2017.].

[20] J. Wharton, "Butter Knife," [Na mreži]. Dostupno na: http://jakewharton.github.io/butterknife/. [Poslednji pristup 19.8.2017.].

[21] "Stetho," Facebook, [Na mreži]. Dostupno na: http://facebook.github.io/stetho/. [Poslednji pristup 16.8.2017.].

[22] "Fabric," [Na mreži]. Dostupno na: https://docs.fabric.io/android/fabric/overview.html. [Poslednji pristup 28.8.2017.].

[23] M. Cleron, "Android Announces Support for Kotlin," Google, 17 5 2017. [Na mreži]. Dostupno na: https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html. [Poslednji pristup 23.8.2017.].

[24] "The Model-View-Presenter (MVP) Pattern," Microsoft, [Na mreži]. Dostupno na: https://msdn.microsoft.com/en-us/library/ff649571.aspx. [Poslednji pristup 26.8.2017.].

Lista dijagrama i slika