Laboratorijska vježba 6

Web servisi i coroutines

Zadatak 1

Registrovati se na stranici The Movie Database i upoznati se sa pretragom po nazivu

Prvi korak implementacije zadataka ove vježbe je registracija na stranici: The Movie Database. Nakon registracije, potrebno je zatražiti izdavanje **API KEY-A**. Isti je potreban prilikom poziva endpoint-ova servisa. Detaljnije o izdavanju istog imate na sljedećem link-u:Introduction.

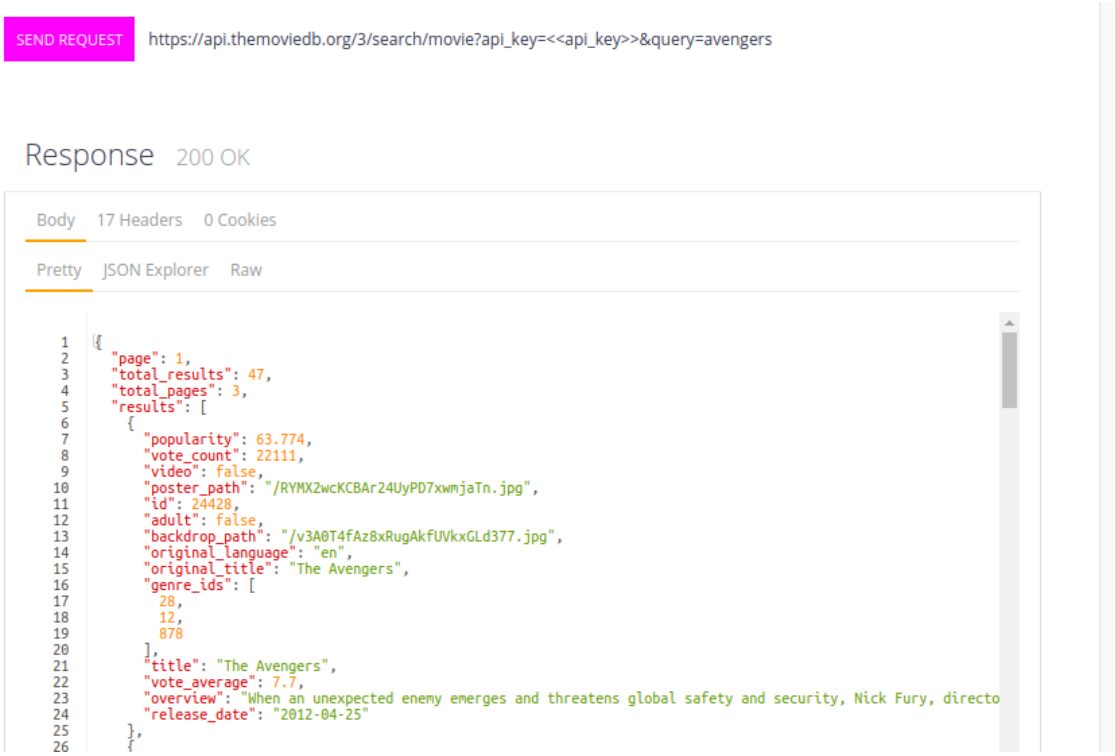
Za potrebe prvog zadatka potrebno je pogledati dokumentaciju pretrage filmova po nazivu, koja je data na stranici Search Movies za servis Movie.

Parametri koji se mogu proslijediti prilikom poziva ovog web servisa su dati u tabeli.

Parametar	Opis
api_key	Obavezni parametar - omogućava korištenje servisa.
query	Obavezni parametar - predstavlja traženi upit
language	Neobavezni parametar (en-US) - izbor jezika
page	Neobavezni parametar (1) - redni broj stranice prikaza. Svaki query može vratiti veliki broj rezultata, pri čemu servis vraća samo određeni broj, te omogućava paginaciju.
include_adult	Neobavezni parametar (false) - da li će vratiti filmove za odrasle
region	Neobavezni parametar - speicifikacija ISO kôda za datum
year	Neobavezni parametar - godina
primary_release_year	Neobavezni parametar - godina prvog prikazivanja

Ovaj web servis možete isprobati u sklopu same dokumentacije ili tako što upišete u web browser URL web servisa zajedno sa parametrima po kojim želite izvršiti pretragu.

Na sljedećoj slici je dat primjer pretrage filmova s nazivom: *Avengers*.



U sklopu URL-a prosljeđujemo `api_key` i `query`. Kao rezultat dobijamo JSON objekat koji kaže da smo na prvoj stranici pretrage, da je ukupan broj rezultata 47 i da su ukupno 3 stranice. Filmovi se nalaze u sklopu niza objekata `results` vraćenog objekta. Ono što možemo primijetiti da određene stavke o filmu mi nemamo u implementiranoj Movie klasi, npr. `poster\_path` itd. Uočimo da se žanr vraća u vidu niza integera, i da web stranica ne postoji.

Zadatak 2

Potrebno je izvršiti izmjenu postojeće implementacije i prilagoditi je web servisu.

U prvom koraku ćemo dodati odgovarajuće `INTERNET` permisije u `Manifest` file. Klasu `Movie` ćemo proširiti sa `posterPath` i postavit ćemo da su atributi žanr i homepage `nullable`. Proširivanjem klase `Movie` trebamo izmijeniti listu statičkih podataka. U layout pretrage ćemp dodati `RecyclerView` ispod `EditText`, u kojem će biti prikazani rezultati pretrage.

Napomena: Za sada nećemo mijenjati statičke podatke za vlastite filmove i najnovije filmove. Izmjene nad ovim stavkama ćemo vršiti kada naučimo bolji način za dohvatanje podataka s web servisa.

Zadatak 3

Potrebno je napraviti funkcionalnost pretrage filmova po nazivu putem web servisa. Korisnik treba imati mogućnost upisivanja naziva filma i pokretanja pretrage putem button-a. Ukoliko je pretraga uspješna, lista filmova se treba ispuniti sa najviše šest pronađenih filmova.

Kako odgovor od web servisa može trajati proizvoljno dugo (loša konekcija, pad performansi servera na kojem se nalazi web servis i sl) izvršavanje navedene funkcionalnosti u niti aktivnosti nije poželjno. Takva implementacija bi zaustavljala rad naše aplikacije sve dok web servis ne vrati odgovor. Zbog svega rečenog ima smisla kreirati novu nit i unutar nje implementirati pozivanje web servisa i obradu njegovog odgovora. Za izvršavanje van glavne niti ćemo koristiti `Kotlin Coroutines`. Da bi koristili `Coroutines` prvo ćemo dodati dependency-u u `Gradle.module`:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```

U prvom koraku ćemo pripremiti sve potrebno u `SearchFragment`-u. Referencirat ćemo odgovarajuće `view`-ove, postaviti adapter, kreirat ćemo instancu `MovieListViewModel`-a, te ćemo postaviti `onClickListener` na `AppCompatImageButton`.

Poziv ka web servis-u će se izvršavati u pozadinskoj niti, pri čemu će biti `main-safe` odnosno neće blokirati izmjene nad korisničkim interfejsom. Kako bi se postiglo da funkcija koja vrši poziv ka web servis-u bude `main-safe` koristit će se `withContext()` funkcija iz coroutine biblioteke koja vrši prebacivanje na drugu nit. Poziv ka web servisu će se izvršavati iz `MovieRepository` klase za sada.

Kreirat ćemo prvo `Result` klasu koja će modelirati povrat mrežnog poziva.

```
sealed class Result<out R> {
    data class Success<out T>(val data: T) : Result<T>()
    data class Error(val exception: Exception) : Result<Nothing>()
}
```

U `MovieRepository` ćemo dodati funkciju `searchRequest` koja prima parametar pretrage.

```
suspend fun searchRequest(
    query: String
): Result<List<Movie>>{
    return withContext(Dispatchers.IO) {
        try {
            val movies = arrayListOf<Movie>()
            val url1 = "https://api.themoviedb.org/3/search/movie?
api_key=$tmdb_api_key&query=$query" //1
            val url = URL(url1) //2
            (url.openConnection() as? HttpURLConnection)?.run { //3
                val result = this.inputStream.bufferedReader().use { it.readText() }
//4
                val jo = JSONObject(result)//5
                val results = jo.getJSONArray("results")//6
                for (i in 0 until results.length()) {//7
                    val movie = results.getJSONObject(i)
                    val title = movie.getString("title")
                    val id = movie.getInt("id")
                    val posterPath = movie.getString("poster_path")
                    val overview = movie.getString("overview")
                    val releaseDate = movie.getString("release_date")
                    movies.add(Movie(id.toLong(), title, overview, releaseDate, null,
null, posterPath))
                    if (i == 5) break
```

```
                }
            }
            return@withContext Result.Success(movies);//8
        }
        catch (e: MalformedURLException) {
            return@withContext Result.Error(Exception("Cannot open
HttpURLConnection"))
        } catch (e: IOException ) {
        return@withContext Result.Error(Exception("Cannot read stream"))
        } catch (e: JSONException) {
        return@withContext Result.Error(Exception("Cannot parse JSON"))
        }
    }
}
```

`Dispatchers.IO` koji se prosljeđuje `withContext` ukazuje da će se coroutine-a izvršiti na niti rezervisanoj za I/O operacije. Iskoristili smo i `suspend` ključnu riječ kako bi forsirali da se ova funkcija zove samo iz `coroutine`.

Objašnjenje metode je dato u nastavku:

1. Varijabla `tmdb\_api\_key` je trenutno privatni atribut metode.
2. Formira se ispravni URL.
3. Otvara se konekcija i vrši se poziv web servisa
4. Rezultat poziva web servisa je u obliku `InputStream-a`. Ovaj objekat ćemo pretvoriti u `String`.
5. Navedeni string sadrži rezultat poziva web servisa. Ovaj rezultat je u `JSON` formatu. Da bi radili sa podacima u JSON formatu koristimo `JSONObject` klasu. Kreirat ćemo novi `JSONObject` kojeg ćemo incializirati sa stringom rezultata. Ovaj objekat će da sadrži čitav JSON rezultata.
6. Iz navedenog objekta možemo izdvojiti djelove rezultata koji nas interesuju. Da bi dobili niz objekata koji nas interesuju, izdvojit ćemo `JSONArray` sa nazivnom `results` iz JSON objekta rezultata.
7. Dalje treba proći kroz listu svih rezultata, preuzeti podatke koji nas interesuju (naziv filma, id, posterPath, overview, releaseDate).
8. Vraćamo podatke.

Odgovarajuću funkciju ćemo pozivati iz metode `MovieListViewModel-a`. Metoda unutar ove klase će isto tako biti coroutine, ali će se ona izvršavati na glavnoj niti, te će samim tim moći izvršiti izmjene nad UI, odnosno proslijediti odgovarajuće podatke fragmentu nakon što ih dobije. Prvo ćemo definisati `CoroutineScope` kao privatnu varijablu.

```
val scope = CoroutineScope(Job() + Dispatchers.Main)
```

`CoroutineScope` vodi računa o svim pokrenutim `Coroutine`-ama.

Kreirat ćemo odgovarajuću metodu:

```
fun search(query: String){
    // Kreira se Coroutine na UI
    scope.launch{
        // Vrši se poziv servisa i suspendira se rutina dok se `withContext` ne završi
        val result = MovieRepository.searchRequest(query)
```

```
        // Prikaže se rezultat korisniku na glavnoj niti
        when (result) {
            is Result.Success<List<Movie>> -> searchDone?.invoke(result.data)
            else-> onError?.invoke()
        }
    }
}
```

Unutar ove metode se pozivaju metode `searchDone` i `onError`. Ove metode su `nullable HighOrderFunction`, koje su privatni atributi klase.

```
class MovieListViewModel(private val searchDone: ((movies: List<Movie>) -> Unit)?,
    private val onError: (()->Unit)?
) {
```

Ove metode su implementirane unutar `SearchFragment-a` i služe za izmjenu UI-a.

Preostaje još da pogledamo izmjene nad `SearchFragment-om`, odnosno implementacije click na dugme, dobivanje ispravnog rezultata i rezultata s greškom.

Instanciranje `MovieListViewModel`-a:

```
movieListViewModel =
MovieListViewModel(this@SearchFragment::searchDone,this@SearchFragment::onError)
```

`OnClick` metoda:

```
private fun onClick() {
    val toast = Toast.makeText(context, "Search start", Toast.LENGTH_SHORT)
    toast.show()
    movieListViewModel.search(searchText.text.toString())
}
```

`searchDone` metoda:

```
fun searchDone(movies:List<Movie>){
    val toast = Toast.makeText(context, "Search done", Toast.LENGTH_SHORT)
    toast.show()
    searchMoviesAdapter.updateMovies(movies)
}
```

`onError` metoda:

```
 fun onError() {
     val toast = Toast.makeText(context, "Search error", Toast.LENGTH_SHORT)
     toast.show()
}
```

S ovim smo uspješno napravili pretragu u aplikaciji.

Zadatak 4

Prikazati poster filma umjesto slike žanr-a.

Izmijenit ćemo adapter i prilagoditi ga pretrazi. U sklopu njega, ako je atribut `posterPath` postavljen, isti će biti prikaz u odgovarajućem view-u. Kako bi prikazali poster, koristit ćemo Glide biblioteku. Istu dodajemo preko dependency-a:

```
implementation 'com.github.bumptech.glide:glide:4.12.0'
annotationProcessor 'com.github.bumptech.glide:compiler:4.12.0'
```

Glide predstavlja image loader za Android. Jednostavan je za korištenje i zahtjeva minimalnu konfiguraciju. Korištenjem Glide-a omogućen je prikaz slika dobijenih iz URL-a. Detaljnije o Glide možete pogledati na linku:Glide. Postavljanje slike unutar adaptera je dato u nastavku:

```
val genreMatch: String? = movies[position].genre
val context: Context = holder.movieImage.getContext()
var id: Int = 0;
if (genreMatch!==null)
id = context.getResources()
    .getIdentifier(genreMatch, "drawable", context.getPackageName())
if (id===0) id=context.getResources()
    .getIdentifier("picture1", "drawable", context.getPackageName())
Glide.with(context)
        .load(posterPath + movies[position].posterPath)
        .centerCrop()
        .placeholder(R.drawable.picture1)
        .error(id)
        .fallback(id)
        .into(holder.movieImage);
```

Varijabla posterPath predstavlja privatni atribut adaptera i ona glasi:

```
private val posterPath = "https://image.tmdb.org/t/p/w342"
```

Istu append-amo sa putanjom koju nam vrati web servis. Izvršimo centerCrop kako bi slika stala u okvir, te dodamo placeholder, error i fallback u slučaju da nije moguće load-ati sliku. Error i fallback su default slika ili slika po žanru.

Izgled aplikacije na kraju zadatka:

Zadaci

1. Preuredite fragment za prikaz detalja o filmu tako da se prikazuju detalji dobijeni putem web servisa, prosljeđivanjem id-a filma. Preuredite aplikaciju tako da i dalje rade funkcionalnosti sa prethodnih vježbi.
2. Korištenjem web servisa napravite da se prikazuju glumci filma i slični filmovi.

Napomena: Koristite sljedeće end-pointove:

- https://developers.themoviedb.org/3/movies/get-movie-details
- https://developers.themoviedb.org/3/movies/get-movie-credits
- https://developers.themoviedb.org/3/movies/get-similar-movies