

**Žilinská univerzita v Žiline**  
**Fakulta riadenia a informatiky**

**Vývoj aplikácií pre mobilné zariadenia**  
(semestrálna práca)

2019/2020  
Letný semester

Kristián Hargaš  
5ZY022

## 1. Popis a analýza riešeného problému

V mojej práci som sa snažil vyriešiť problém týkajúci sa veľmi rýchleho obsadenia skúškových termínov po ich vyhlásení na serveri <https://vzdelavanie.uniza.sk/vzdelavanie>. Vždy mi zostanú len tie najmenej atraktívne skúškové termíny, pretože zabúdam kontrolovať stránku vypísaných termínov. Tak som sa rozhodol spraviť mobilnú aplikáciu, ktorá to bude sledovať za mňa, a o nových termínoch ma bude informovať prostredníctvom notifikácie.

Bolo by teda vhodné, keby táto aplikácia umožňovala prihlásiť sa na školský server, následne načítala všetky zapísané predmety z oboch semestrov a dovolila vybrať si predmety, ktorých skúškové termíny má sledovať. Potom na pozadí, v pravidelných intervaloch môže kontrolovať všetky vybrané predmety, a pri nájdení nového skúškového termínu upozorniť používateľa notifikáciou. Takto vie študentovi zautomatizovať život a ušetriť čas.

Táto aplikácia je vytvorená mnou, podľa potrieb, intuície, kreativity, nemal som nejakú predlohu, podľa ktorej som šiel, alebo ktorú by som vylepšoval. Na trhu určite sú nejaké podobne fungujúce watchdog aplikácie, ktoré sledujú napríklad ceny na e-shopoch, ale o tejto aplikácii som dlho uvažoval a navrhol som ju aj bez potreby nejakej inšpirácie podobnou aplikáciou. Vo svojej podstate som ale vychádzal zo štruktúry webových stránok školského servera. Zapúzdрил som tento koncept do mobilnej aplikácie a pridal nejaké automatizačné prvky.

## 2. Návrh riešenia problému

Ako je asi aj z povahy aplikácie zrejmé, jadrom bude interakcia so školským serverom. Najskôr som chcel zvoliť racionálny prístup a snažil som sa nájsť nejaké *REST* rozhranie školského servera, ktoré by mi vracalo požadované dáta napr. vo formáte *JSON*. Navštívil som aj *Centrum vzdelávacích a informačných systémov na UNIZA (CeIKT)*, no kým som niečo zaujímavé zistil, prišla pandémia a nejako na to nebol čas.

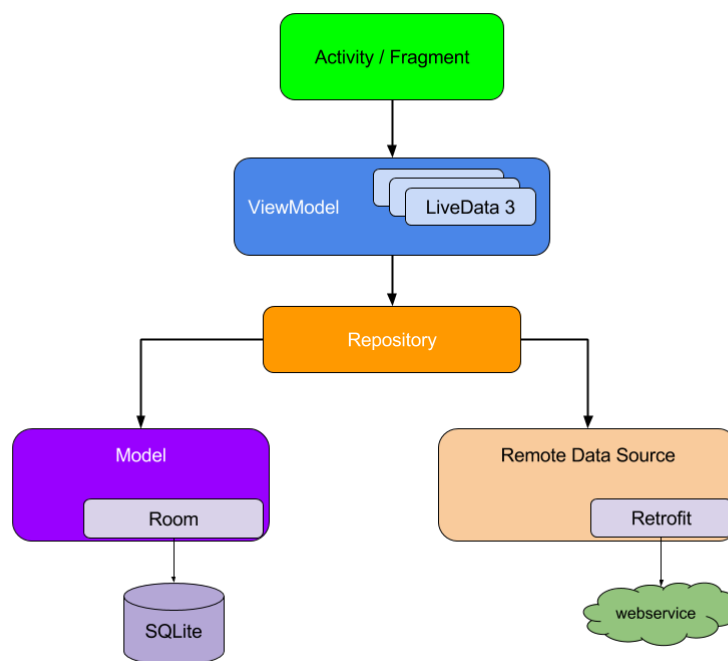
Takže bez nejakého *REST* rozhrania na strane servera mi nezostalo nič iné, než čítať požadované dáta priamo z *HTML* súborov webových stránok, čiže prostredníctvom tzv. techniky *web scraping*. K realizácii sa vrátim pri časti dokumentácie popisujúcej implementáciu.

Keď som našiel spôsob, akým by bola komunikácia so serverom možná, pokračoval som v prieskume ďalej. Vedel som že budem musieť server opakovane dotazovať, získané dáta lokálne ukladať. Keďže aplikácia prakticky z väčšej časti funguje na nejakých vstupno-

výstupných operáciach, potreboval som aj spôsob jednoduchého a elegantného asynchrónneho programovania. Nakoniec sa ukázalo, že *Android Jetpack* nástroje v kombinácii s programovacím jazykom *Kotlin* bude ten najlepší smer, akým sa môžem vydať, neskôr vysvetlím prečo.

Aplikácia využíva z hľadiska používateľského rozhrania princíp jednej aktivity, to znamená že obsahuje iba jednu hlavnú aktivitu ako vstupný bod do aplikácie a jednotlivé obrazovky sú reprezentované fragmentmi.

Okrem iného je architektúra aplikácie rozdelená do podobných vrstiev, aké môžeme vidieť na nasledujúcom obrázku. Zaujímavým prvkom v modeli je *Repository*, ktorý predstavuje abstrakciu rôznych zdrojov dát (databáza, sieť, ...) pre vyššie vrstvy.



Obr. 1 Architektúra aplikácie

Ďalej by som ešte rád uviedol snímky a popis jednotlivých obrazoviek pre lepšie pochopenie fungovania aplikácie.

Aplikácia začína na prihlasovacej obrazovke, bez prihlásenia totižto stráca zmysel. Prijíma rovnaké údaje ako školský server. Po prihlásení sú dáta uložené lokálne v zašifrovanej podobe, je to nutné z dôvodu opätovného prihlásenia po vyexpirovaní relácie.

Obr. 2 Prihlasovacia obrazovka

Po úspešnom prihlásení dochádza k načítaniu zapísaných predmetov z webových stránok školského servera a následnému uloženiu týchto dát do lokálnej databázy. Zároveň sú zobrazené aj v zozname s možnosťou prepínania medzi semestrami.

Obr. 3 Hlavná obrazovka

Môžeme tu zapnúť alebo vypnúť sledovanie konkrétneho predmetu pomocou prepínača pri príslušnom predmete. Nájde tu aj čas poslednej kontroly skúškových termínov daného

predmetu. Sledovanie vybraných predmetov sa dá zapnúť alebo vypnúť aj na globálnej úrovni pomocou tlačidla “oka” horného menu, ktoré reflektuje aktuálny stav sledovania. To znamená, že na obrázku vyššie je oko neprečiarknuté, a teda vybrané predmety (majú zapnutý prepínač) sú sledované.

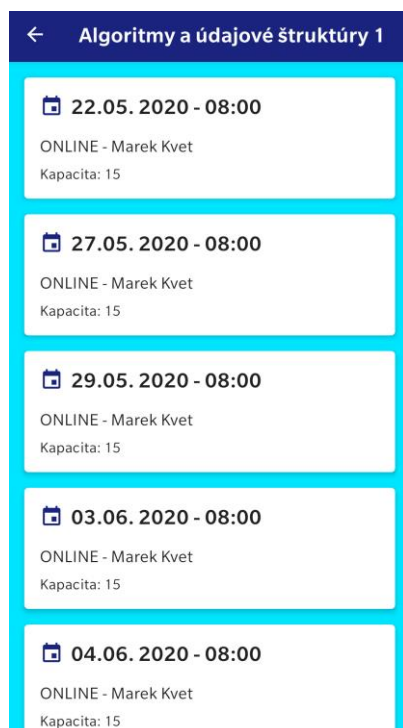
Po globálnom zapnutí sledovania dochádza k prvej kontrole skúškových termínov vybraných predmetov po 20 sekundách a následne každých 15 minút za predpokladu, že zariadenie má zapnuté internetové pripojenie.

Sledovanie skúškových termínov prebieha na pozadí, a preto nevyžaduje spustenie aplikácie. Ak ale dôjde k objaveniu nových skúškových termínov, používateľ bude upozornený prostredníctvom notifikácie.



*Obr. 4 Notifikácia*

Po kliknutí na nejaký predmet zo zoznamu predmetov dôjde k otvoreniu obrazovky, ktorá zobrazí všetky doposiaľ objavené skúškové termíny pre daný predmet. Po kliknutí na skúškový termín sa zobrazí poznámka, ak nejakú skúšajúci pri vypisovaní termínu uviedol.



*Obr. 5 Obrazovka skúškových termínov*

### 3. Popis implementácie

#### 3.1. Autentifikácia

Prihlasovanie prebieha prostredníctvom fragmentu s názvom *LoginFragment*, ktorý čerpá dáta z view modelu s názvom *LoginViewModel*.

V prípade autentifikácie používam *HTTP* knižnicu *Retrofit 2*, keďže školský server vracia v rámci prihlasovania *JSON*. Stačí iba zadeklarovať potrebné rozhranie s použitím anotácií, ktoré definujú jednotlivé požiadavky. Formát komunikácie so serverom som zistil prostredníctvom vývojárskych nástrojov prehliadača. Na nasledujúcom obrázku môžeme vidieť deklaráciu rozhrania pre prihlasovanie v súbore *AuthService.kt*.

```
@FormUrlEncoded
@POST( value: "login.php")
suspend fun login(
    @Field( value: "meno") name: String,
    @Field( value: "heslo") password: String
): Response<AuthRes>
```

Obr. 6 *AuthService.kt*

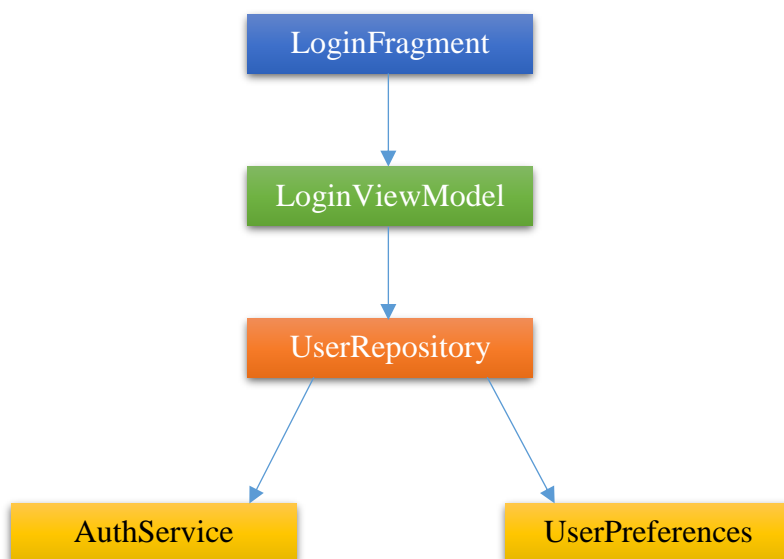
Toto rozhranie využíva trieda *UserRepository*, čo je teda repozitár používateľa a spravuje všetky dáta a operácie spojené s používateľom vrátane autentifikácie. Čiže v metóde *UserRepository.login()* dochádza k autentifikácii a k spracovaniu odpovede zo servera. Dôležité je extrahovanie “*session cookie*”, ktorý je asociovaný s danou reláciou práve prihláseného používateľa.

Získaný “*cookie*” je ďalej uložený aj lokálne v zašifrovanej podobe prostredníctvom *EncryptedSharedPreferences*, ktoré sú zapúzdrené v triede *UserPreferences*, jedná sa o veľmi bezpečný spôsob ukladania dát lokálne na zariadení. Na nasledujúcom obrázku môžeme vidieť, ako takýto súbor po zašifrovaní vyzerá.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="AXSKDbKh+rnplHmaRcvTS/pYfubH+0w10Ak2CM">Aue1SQN+5g7Fy56s51b/
wc30oVKqZcQE1tCoE8FkuPSR5djAYZM6nDb15UrSRmgzWRDy7E=</string>
  <string name="__androidx_security_crypto_encrypted_prefs_key_keyset">12a901adf5be4e7a5387789f12d4eaba4cbb073d9d0f436f9
2157f119f901bc94237718c5c0be5539081b1cb60a1d5c83c7f73a1b55806a0495a06038697648216a584ee62af0c64f79ccf0b97af13b8d05c8f
d2c96febe2c427d5214e2956dbca9f0a15366edde46afac05b01dddbfae2144a747c8865a87fae539e5e69ed3dfc57f30df1cfc84437c765e02cd82
1e6c2a0b9a61922d9d0dc414596a1228f3c326123bc6455b108aa691a4408b29ba8a407123c0a30747970652e676f676c65617069732e636fd2f6
76f676c652e63727970746f2e74696e6b2e4165735369764b6579100118b29ba8a4072001</string>
  <string name="__androidx_security_crypto_encrypted_prefs_value_keyset">128801dfbb531042d64526535f258580e4c2ad54e70e4e
39a8acd53b34fc7d5431793fd3f1b8ec02aac88f47b00dfc58dec59e5fe4cb72098298f6d2470b63abac1e39c8d921aca8b8bca1b22276cc534022b7
071f5a4c43ac79580a971bf720e3af62d9d40c4af19986613d4b66597c2e1f9532e8dc5ae817ebf9e6a901e572ba04625da1dcfbfbc591a4408839
295bd04123c0a30747970652e676f676c65617069732e636fd2f676f676c652e63727970746f2e74696e6b2e41657347636d4b6579100118839
295bd042001</string>
  <string name="AXSKDbISpKfItSupPV/A174X0dkTYTsJ/3rwL+ud3RU01ZU=">
Aue1SQMq4Dwn6Y9e06+pMKzCf0+8JDKC5SM7QoBwaCgef0ZZ5xet8pbu2r6JqjvmbNtoz0YdvF7X06YR+cy2NQ0xhaD00RaJBS955Kxbz+fc=</string>
  <string name="AXSKDbLYmkHf295GgM6+7FPscdLL3RHLBTJZE=">Aue1SQNWQTYFawCtX67zsuPG/t5QvHk/RWUtNCJuzaHuH/qhQ67mftCq9q/1v87Q
</string>
</map>
```

Obr. 7 *EncryptedSharedPreferences*

Keď to teda všetko spojíme, výsledná architektúra prihlasovania vyzerá nasledovne. Mala by byť veľmi podobná tej teoretickej, ktorá je uvedená v predchádzajúcej sekcii.

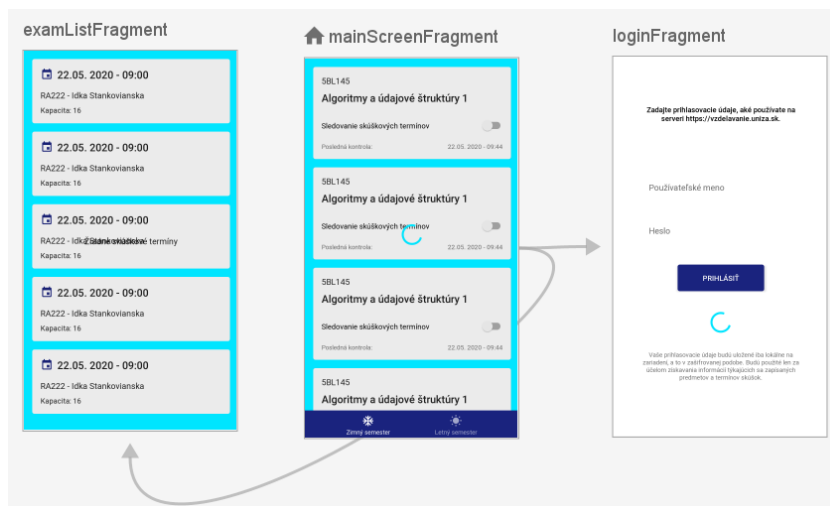


Obr. 8 Architektúra autentifikácie

Komunikácia medzi jednotlivými entitami je zabezpečená prostredníctvom metód, ale aj pomocou *LiveData* objektov, ktoré sleduje ten, kto má o ne záujem a v prípade zmeny je zavolaný zaregistrovaný *callback*. Samotný layout (*fragment\_login.xml*) využíva *two-way data binding*, a je teda naviazaný priamo na view model, tým pádom prípadný reštart fragmentu nespôsobí výpadok dát z textových polí.

### 3.2. Navigation component

Ako už bolo spomenuté, aplikácia využíva princíp jednej hlavnej aktivity (*MainActivity*), ktorá predstavuje vstupný bod a jednotlivé obrazovky sú koncentrované vo fragmentoch. Celé je to realizované prostredníctvom *navigation component Jetpacku*. Zadefinoval som si *navigation graph*, jednotlivé obrazovky a prechody medzi nimi. Následne už len prostredníctvom *NavigationController* (súčasť *navigation component*) riadim prechod medzi obrazovkami. Na nasledujúcom obrázku môžeme vidieť spomínaný navigačný graf.



Obr. 9 Navigation graph

### 3.3. Web scraping

Na získanie informácií o zapísaných predmetoch a skúškových termínoch z *HTML* súborov som použil nástroj *Jsoup*. Jedná sa o veľmi šikovnú knižnicu, ktorá umožňuje po načítaní a parsnutí *HTML* súborov elegatne selektovať relevantné dáta.

Ďalej to už bolo len o analýze štruktúry *HTML* kódu stránok vrátených zo školského servera. Celá logika scrapovania, či už zapísaných predmetov alebo skúškových termínov, sa nachádza v súbore *Scraper.kt*.

Napr. v prípade zapísaných predmetoch sú predmety zapísané v tabuľke, ktorá je v tabuľke s identifikátorom *id-tabulka-predmety-s*, a jednotlivé predmety sú v riadkoch danej tabuľky. Výber množiny riadkov predmetov z parsnutého *HTML* dokumentu vyzerá nasledovne.

```
// select all rows from table of user's subjects
val tableRows: Elements = doc.select( cssQuery: "table#id-tabulka-predmety-s table tbody > tr")
```

Obr. 10 Výber množiny riadkov predmetov

Táto tabuľka obsahuje okrem predmetov aj popisky pre semestre. Po skúmaní som zistil, že element predmetu má práve 5 potomkov (stĺpcov), takže keď na taký narazím, extrahujem dané dáta zo stĺpcov a plním nimi objekt triedy popisujúci predmet – *Subject*.

```
// subject has 5 columns and follows header for winter or summer term
if (term != null && it.childrenSize() == 5) {
    val subjectIdAndName = it.child( index: 0 ).text().trim()
    val subjectId = subjectIdAndName.substring(0, subjectIdAndName.indexOf( char: ' ' )).trim()
    val subjectName = it.child( index: 0 ).select( cssQuery: "a" ).text().trim()
    val subjectExamsUrl = it.child( index: 3 ).select( cssQuery: "a" ).attr( attributeKey: "abs:href" ).trim()

    // create new subject object
    subjects.add(Subject(subjectId, subjectName.capitalize(), term!!, subjectExamsUrl))
}
```

Obr. 11 Extrahovanie dát o predmete

### 3.4. Room

Pre uchovávanie dát o zapísaných predmetoch a skúškových termínoch lokálne na zariadení používam *Room* databázu. Všetky súbory a triedy týkajúce sa práce s databázou sa nachádzajú v balíčku *.database*. V súbore *Room.kt* je definovaný hlavný objekt databázy.

Pre každý objekt mám vytvorenú entitu, teda popisnú triedu, ktorej inštancie budú uchovávané v databáze vo forme riadkov tabuľky. Napr. v prípade skúškového termínu vyzerá trieda *DatabaseExam* nasledovne.

```
@Entity(tableName = "exams")
data class DatabaseExam(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "exam_id")
    var id: Int = 0,
    val date: Date,
    val room: String,
    val teacher: String,
    val capacity: Int,
    val note: String,
    @ColumnInfo(name = "subject_id")
    val subjectId: String
)
```

Obr. 12 DatabaseExam

Ďalej mám zdefinované *DAO* rozhranie pre prácu s entitami (*DAO* – *database access object*), ktoré implementuje za mňa *Room*. V prípade skúškových termínov vyzerá nasledovne.

```
@Dao
interface ExamDao {
    @Query(value = "SELECT * FROM exams WHERE subject_id = :subjectId")
    suspend fun getExamsForSubject(subjectId: String): List<DatabaseExam>

    @Insert
    suspend fun insertAll(exams: List<DatabaseExam>)

    @Query(value = "DELETE FROM exams")
    suspend fun deleteAllExams()
}
```

Obr. 13 ExamDao

S týmito *DAO* rozhraniami opäť pracujú repozitáre – *SubjectRepository* a *ExamRepository*, ktoré navyše pracujú aj s funkciami zo súboru *Scraper.kt*, keď potrebujú dáta získať z webu. Vyššie vrstvy podľa potreby pracujú s repozitármi, a tie zas vedia, na aký zdroj dát sa majú obrátiť.

### 3.5. Kotlin coroutines

Pre vykonávanie operácií s databázou alebo sieťou používam *coroutines*. Je to veľmi jednoduchý a flexibilný spôsob, ako neblokovať hlavné vlákno pri vstupno-výstupných operáciach. Dokonca aj nástroje ako *Room* a *Retrofit 2* obsahujú vstavanú podporu pre

*coroutines*. Spúšťanie *coroutines* sa nachádza prakticky naprieč celým projektom, podľa potreby. Stačí len spustiť *job*, podľa potreby prepnúť *kontext* vykonávania napr. do vlákna vstupno-výstupných operácií, vykonať operáciu a hotovo. Kód je čistý a ľahko sa píše. V mojom prípade spúšťam *coroutines* najmä vo view modeloch v ich *viewModelScope*, pri rušení view modelu dôjde automaticky k zrušeniu všetkých *coroutines* bežiacich v danom *scope*.

Napr. v prípade view modelu *MainViewModel*, ktorý dodáva dáta o predmetoch cez *SubjectRepository* obrazovke *MainScreenFragment*, dochádza po zavolaní metódy *loadSubjects()* k spusteniu coroutine.

```
fun loadSubjects() = viewModelScope.launch { this: CoroutineScope
    subjectRepository.loadSubjects()
}
```

Obr. 14 Spustenie coroutine z *MainViewModel*

Následne sa v danej metóde (*SubjectRepository.loadSubjects()*), ktorá je *suspend*, a tak musí byť volaná z *coroutine*, prepína kontext na *Dispatchers.IO*, pretože sa chystáme pracovať s databázou a s webom, tým uvoľňujeme hlavné vlákno. Potom už len dochádza k zisťovaniu, či máme v databáze nejaké predmety. V prípade prázdnej databázy je obnovená relácia prihláseného používateľa, následne dochádza k získaniu predmetov z webu pomocou scrapovania a výsledok je uložený do databázy.

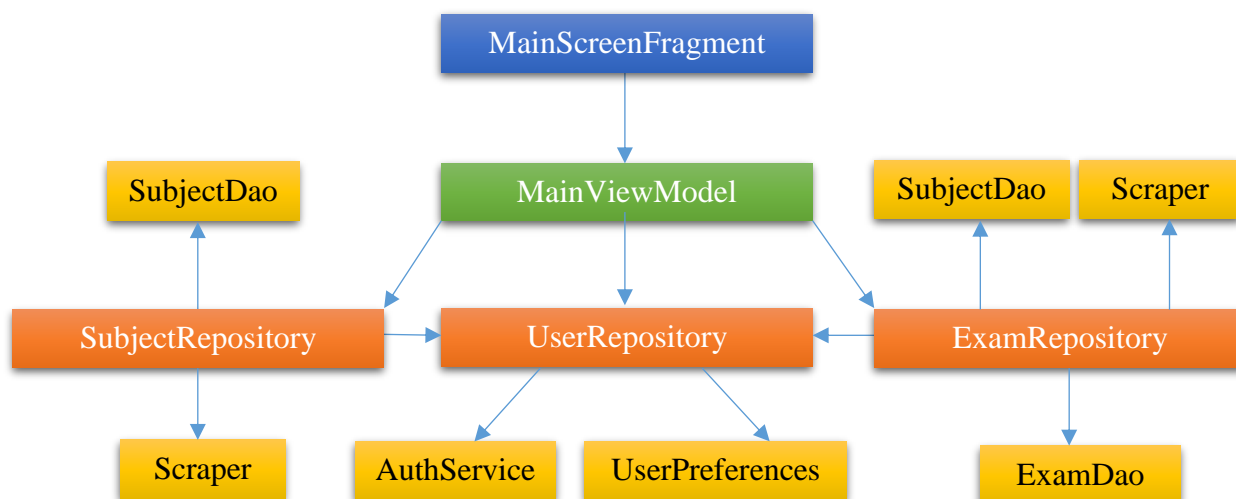
```
suspend fun loadSubjects() = withContext(Dispatchers.IO) { this: CoroutineScope
    val subjectCount: Int = subjectDao.getSubjectCount()
    // db is empty, lets scrape the web
    if (subjectCount == 0) {
        // refresh session in case it expired
        if (userRepository.refreshSessionFromApp()) {
            val subjects: List<Subject> = scrapeSubjects(userRepository.getSessionCookie())
            subjectDao.insertAll(subjects.asDatabaseModel())
        }
    }
}
```

Obr. 15 Načítavanie zapísaných predmetov

### 3.6. Hlavná obrazovka

Hlavná obrazovka – *MainScreenFragment* obsahuje zoznam všetkých zapísaných predmetov prihláseného používateľa. V prípade, že nie je žiaden používateľ prihlásený, otvára fragment *LoginFragment*.

Dáta opäť dodáva *MainViewModel*, ktorý pracuje s repozitármi – *UserRepository*, *SubjectRepository* aj *ExamRepository*. Na nasledujúcom obrázku môžeme vidieť známu architektúru.



Obr. 16 Architektúra hlavnej obrazovky

Čiže *MainScreenFragment* obsahuje *RecyclerView*, ktorý je napájaný adaptérom *SubjectListAdapter*, a teda zobrazuje zapísané predmety prihláseného používateľa. Okrem iného obsahuje aj hlavné menu, ktoré umožňuje globálne zapnutie a vypnutie sledovania vybraných predmetov a možnosť odhlásiť sa.

Vo všetkých layoutoch je vlastne implementovaný *two-way data binding* pre spojenie dát s používateľským rozhraním.

Samotná položka predmetu umožňuje zapnúť alebo vypnúť sledovanie konkrétneho predmetu a po kliknutí na ňu sa otvorí fragment *ExamListFragment*, ktorý zobrazí všetky doposiaľ objavené predmety pre daný predmet. Principiálne funguje ale rovnako.

### 3.7. Notifikácie

Aplikácia v prípade objavenia nových skúškových termínov zobrazí notifikáciu. Celý kód spojený s notifikáciami je extrahovaný do súboru *NotificationManager.kt*, ktorý obsahuje jednu metódu pre vytvorenie notifikačného kanála na nových zariadeniach a druhú metódu, ktorá zobrazí notifikáciu.

### 3.8. WorkManager

Pre pravidelné kontrolovanie nových skúškových termínov pri sledovaných predmetoch je použitý ďalší komponent *Jetpacku* – *WorkManager*. Celá jeho definícia a konfigurácia je sústredená v súbore *WatchdogWorker.kt*.

Veľkou výhodou tohto objektu je možnosť zadefinovania podmienok, ktoré musia byť splnené, aby sa práca vykonala, napr. pripojenie k internetu. Ďalšou výhodou je možnosť zadefinovania periodického opakovania, v našom prípade každých 15 minút.

Po kliknutí na položku “oka” v hlavnom menu hlavnej obrazovky dochádza k zaradeniu daného objektu *workera* do fronty, následne sa už samotný operačný systém stará o spustenie. Nastavenie periodického spúšťania každých 15 minút neznamena, že to bude presne každých 15 minút. Operačný systém sa rozhodne aj na základe aktuálnej situácie v danej chvíli. Spustenie je však garantované.

Samotná konfigurácia podmienok a spúšťania *workera* vyzerá nasledovne. Po zapnutí sa teda worker spustí po 20 sekundách a následne každých 15 minút.

```
// work manager is executed only when all of the following conditions are met.
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresBatteryNotLow(false)
    .build()

// configuring timing
val request = PeriodicWorkRequestBuilder<WatchdogWorker>(repeatInterval: 15, TimeUnit.MINUTES)
    .setInitialDelay(duration: 20, TimeUnit.SECONDS)
    .setConstraints(constraints)
    .build()
```

Obr. 17 WatchdogWorker konfigurácia

Po spustení sa vykoná metóda *WatchdogWorker.doWork()*. Tá pomocou *ExamRepository.checkExams()* zistí nové skúškové termíny pre každý sledovaný predmet. *Repozitár* informuje *workera* o nových termínoch prostredníctvom posluchača v reakcii na čo *worker* vyšle notifikáciu.

```
override suspend fun doWork(): Result {
    // listener for newly added exams basically shows notification to the user
    val examDiscoveryListener: ExamDiscoveryListener = object : ExamDiscoveryListener {
        override fun newExamsDiscovered(subject: Subject, newExamsCount: Int) {
            showWatchdogNotification(context, subject, newExamsCount)
        }
    }

    // check watched subjects and return result
    return when (ExamRepository.getInstance(context).checkExams(examDiscoveryListener)) {
        true -> Result.success()
        else -> Result.failure()
    }
}
```

Obr. 18 Implementácia WatchdogWorker

Pre zaujímavosť uvediem aj kód metódy *ExamRepository.checkExams()*, ktorá sa stará o samotnú detekciu nových skúškových termínov.

Na začiatku dochádza k získaniu všetkých sledovaných predmetov (prepínač je pri danej položke predmetu zapnutý). Ak nesledujeme nič, končíme. V prípade sledovania nejakých predmetov najskôr obnovíme reláciu so serverom, pretože mohla vypršať.

```
// get all of the watched subjects
val watchedSubjects: List<Subject> = subjectDao.getWatchedSubjects().asDomainModel()
// there is nothing to check
if (watchedSubjects.isEmpty()) return@withContext true
// refresh user session
val sessionCookie: String = UserRepository.refreshSessionFromBackgroundAndGetCookie(context)
?: return@withContext false
```

Obr. 19 ExamRepository() - 1. časť

Ďalej dôjde k spusteniu paralelných *jobov* (čiže *coroutines*), pričom každý kontroluje nové skúškové termíny pre jeden zo sledovaných predmetov. Najskôr sa získa zoznam doposiaľ objavených skúškových termínov pre daný predmet z lokálnej databázy, následne dôjde ku scrapovaniu všetkých skúškových termínov pre daný predmet z webu. Hneď potom dochádza k porovnaniu dvoch zoznamov, čiže ak je taký termín, ktorý bol načítaný z webu ale nenachádza sa v lokálnej databáze, tak asi je novo pridaný. Na záver len aktualizujeme poslednú kontrolu predmetu na aktuálny čas, informujeme posluchača o prírastku skúškových termínov a uložíme tieto termíny do lokálnej databázy.

```
// parallel jobs each to check newly added exams on the web for every watched subject
val jobs: List<Deferred<Subject?>> = watchedSubjects.map { subject ->
    async { this: CoroutineScope
        // get locally stored exams for the current subject
        val databaseExams: List<Exam> =
            examDao.getExamsForSubject(subject.id).asDomainModel()
        // get all exams from the web for the current subject
        val webExams: List<Exam> = scrapeExams(subject, sessionCookie)

        // filter newly added exams
        val newExams: List<Exam> = webExams.filter { it: Exam
            !databaseExams.contains(it)
        }

        // subject checked, update timestamp
        subjectDao.updateLastCheck(subject.id, Calendar.getInstance().time)

        // broadcast notification and save newly found exams to database
        if (newExams.isNotEmpty()) {
            listener.newExamsDiscovered(subject, newExams.size)
            examDao.insertAll(newExams.asDatabaseModel())
        }

        // if there are newly found exams for the current subject,
        // return subject as the result of this job, null otherwise
        if (newExams.isNotEmpty()) subject else null ^async
    }
}
```

Obr. 20 ExamRepository() - 2. časť

Nakoniec počkáme len na vykonanie všetkých jobov a v prípade, že všetky prebehli úspešne, vrátime *true*, inak *false*.

```
try {  
    // await until all jobs are completed  
    jobs.awaitAll()  
    true ^withContext  
} catch (e: Throwable) {  
    false ^withContext  
} ^withContext
```

*Obr. 21 ExamRepository() - 3. časť*

## 4. Záver

Toto bol teda stručný popis aplikácie a jej implementácie. Všetko som otestoval a odladil a malo by to byť funkčné. Dúfam, že budete mať ako aplikáciu otestovať, keďže vyžaduje študentské prihlásenie a najlepšie aj nejaké skúškové termíny. Každopádne prikladám aj video, kde si tiež môžete pozrieť funkcionality. Video miestami sekne, ale v realite to ide krásne plynule.