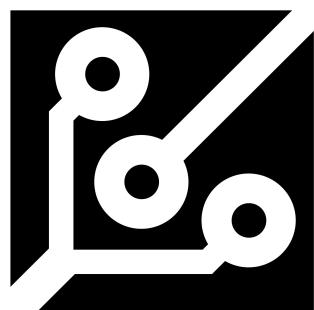


Take Your Pill

mobilní aplikace

SPŠE V Úžlabině



Vojtěch Hořánek

I4.D

13.04.2021

„Prohlašuji, že jsem tuto práci vypracoval samostatně a použil jsem literárních pramenů a informací, které cituji a uvádím v seznamu použité literatury a zdrojů informací.“

V Praze dne

.....

podpis autora

Anotace

Předložená práce je mobilní aplikace pro systém Android, která slouží k připomínání užívání léků. V aplikaci je implementovaná historie připomenutí a užití léků, statistiky a grafy. Uživatel snadno získá přehled, jaké a kdy léky vyneschal a může si nastavit opakování připomínání, aby již lék nezmeškal. Design aplikace je navrhnut tak, aby odpovídala material designu a jeho nejnovějším trendům. Aplikace je napsána v jazyce Kotlin a využívá moderních knihoven a technologií.

Anotation

The presented work is a mobile application for the Android operating system which reminds its users to take their pills. History, statistics, and graphs are implemented in the application. The user can effortlessly get an overview of which pills at what time did they miss and can set repeating reminders, so they do not forget them the next time. The application follows the material design guidelines and its latest trends. Kotlin was used as the programming language and the application utilizes modern libraries and technologies.

Obsah

Úvod	1
1 Návrh aplikace	2
2 Implementace aplikace	3
2.1 Uživatelské rozhraní	4
2.1.1 Úvodní obrazovka	4
2.1.2 Domovská obrazovka	5
2.1.2.1 Detail léku	6
2.1.2.2 Nový lék	6
2.1.3 Historie	8
2.1.3.1 Přehled	8
2.1.3.2 Grafy	9
2.1.4 Nastavení	10
2.1.4.1 O aplikaci	11
2.2 Programová implementace	12
2.2.1 Databáze	12
2.2.2 Připomínání	14
Závěr	18

Úvod

Cílem této práce bylo vytvořit aplikaci, která uživatelům usnadní pravidelné užívání léků jejich připomínáním. Uživatel se také může podívat na statistiky užívání a zobrazit si kompletní historii připomínek. Pro každý lék lze nastavit příjem, tj. aby se připomínal pouze určitý počet dní, nebo aby se připomínal v cyklu X dní aktivní, Y dní neaktivní.

Práce se skládá z mobilní aplikace pro systém android. Aplikace je navržena co nejjednodušejí a rozdělena do dvou hlavních sekcí: léky a historie. V sekci „léky“ uživatel nalezne léky, které si do aplikace přidal. V seznamu léků je zobrazen jejich název, popis, barva, fotografie, časy připomínek a příjem. V sekci „historie“ může uživatel sledovat užití svých léku pomocí statistik, zobrazí se mu kompletní historie (včetně kdy a jestli si lék vzal, kdy mu byla poslána připomínka a kolik prášků si vzal) a grafy zobrazující souhrnné informace. Součástí práce je i propagační a informační plakát.

Kapitola 1

Návrh aplikace

Design aplikace jsem navrhoval před samotnou implementací, nutno ale dodat, že při implementaci prošel design několika iteracemi. Aplikaci jsem původně koncipoval jako jedinou hlavní obrazovku, kde by se uživateli ukázali všechny důležité informace. Postupem času se toto řešení ukázalo jako nevhodné a nepraktické, zvolil jsem proto více tradiční postup a to rozdelení aplikace do tří přehledných sekcí: *Léky*, *Historie* a *Nastavení*. Každá obrazovka obsahuje velký nadpis a teprve potom samotný obsah. Mimo spodních dialogů¹ není nadpis nijak ohrazen, pouze je odsazen. Změny mezi různými obrazovkami doprovázejí animace, které jsou uváděny podle Material Designu. Aplikace díky těmto animacím vypadá svižněji.

Celé rozhraní jsem upravil pomocí vlastních stylů, které vycházejí ze stylů Material Design [14]. Pro některé prvky aplikace, jmenovitě titulky a tlačítka, jsem použil písmo *Jost* [11]. Ikony použité v aplikaci jsou z knihovny Material Design Icons [15], ikonu aplikace jsem získal od Austina Andrewse [1] a grafika léků je dostupná na GitHubu pod názvem *material-icons* [21]. Nechybí ani podpora světlého a tmavého designu, který se dá přepínat v nastavení (více v kapitole 2.1.4).

¹Spodním dialogem je myšleno BottomSheetDialogFragment.

Kapitola 2

Implementace aplikace

Při vytváření aplikace jsem vycházel ze zadání a využil jsem vlastních znalostní a zkušeností. Na naprogramování aplikace jsem použil programovací jazyk Kotlin. Zvolil jsem ho, jelikož je preferovaný společností Google, a oproti jazyku Java má podle mého nespočet výhod. Mnoho Android knihoven vychází právě pro Kotlin, a tak mi jeho použití ulehčilo mnoho práce při programování. Jmenovitě knihovny z rodiny Android Jetpack [2] jsem použil hojně.

Uživatelské rozhraní aplikace je napsáno v jazyce XML a pro jeho manipulaci jsem použil knihovnu *ViewBinding* [10]. Pro snadnější práci s ViewBindingy jsem využil další knihovnu s názvem *FragmentViewBindingDelegate-KT* [22][23], která mi zpřístupnila „delegát“ by `viewBinding()`, s jehož pomocí jsem mohl layout (rozložení) inicializovat (a automaticky zahodit) jen pomocí jedné řádky kódu.

Při samotném vývoji jsem používal vývojové prostředí *Android Studio* [13] a emulátor *Android Emulator* integrovaný do zmíněného vývojového prostředí.

Aplikace je napsána tak, aby odpovídala architektuře **Model-View-ViewModel**. Znamená to, že každá obrazovka má svůj *ViewModel* a každá datová sekce má svůj *Repositář*. Tyto třídy jsou odděleny od samotných fragmentů a aktivit. Pro asynchronní práci s daty používá databázová vrstva spolu s repozitáři *Flow* [6] (vysvětlení v kapitole 2.2.1), které se ve *ViewModelu* mění na *LiveData* [7] pomocí metody `asLiveData()`.

ViewModel je třída obsahující data a metody pro její fragment/aktivitu, která má vlastní životní cyklus. Díky ViewModelu data přežijí změnu konfigurace jako například otočení obrazovky.

Repozitář (repository) je třída, která shromažďuje data z různých zdrojů a nabízí je ve vhodné formě ostatním třídám (například může data uchovat v mezipaměti). V této aplikaci přistupuje repozitář pouze do databáze a ve většině případů přímo volá metody implementované v databázové vrstvě.

Aby se aplikace snáze programovala a byla více škálovatelná, rozhodl jsem se použít tzv. „automated dependency injection“ [4]. Doporučená knihovna pro automated dependency injection **Hilt** [5] je sice stále v beta verzi, ale já jsme je ji rozhodl použít, protože mě u jiných projektů fungovala bezproblémově.

Automated dependency injection (česky automatizované vkládání závislostí) je technika, která zajistí, že třídy, které k činnosti potřebují ostatní třídy, je dostanou automaticky a v jakémkoli podporovaném kontextu. Když například repozitář potřebuje ke své funkci třídu, která přistupuje do databáze, automated dependency injection zařídí, že tuto třídu automaticky dostane a programátor ji nemusí explicitně vytvářet. V případě hiltu je možné využít i vytváření *singletonů*, což zajistí, že instance třídy existuje v aplikaci pouze jednou a jiné třídy dostanou právě tuto instanci.

2.1 Uživatelské rozhraní

Hlavní obrazovka aplikace je rozdělena na tři části: *Léky*, *Historie* a *Nastavení*. Uživatel mezi těmito sekczemi přepíná pomocí prvku **BottomNavigationView**. Do sekce *Historie* a na obrazovku *Přidat lék* se může uživatel dostat i pomocí zkratky¹ z domovské obrazovky. Celá aplikace obsahuje pouze tři aktivity: **MainActivity**, **AboutActivity** a **AppIntroActivity**. Všechny ostatní obrazovky jsou implementovány jako fragmenty a pro jejich navigaci byla použita knihovna *Navigation* [8].

2.1.1 Úvodní obrazovka

Pro přidání úvodní obrazovky jsem použil knihovnu *material-intro* [20]. Tato knihovna se postará o rozložení a logiku úvodní obrazovky. Jelikož má jednoduché API, stačilo mi do

¹Zkratky využívají App Shortcuts API.

aplikace přidat aktivitu `AppIntroActivity` dědící ze třídy `IntroActivity`. V této třídě jsem přidal slidy pomocí metody `addSlide()`. Obrázky ve slidech jsem vyfotil v android emulátoru a upravil v programu *GIMP*.

Úvodní obrazovka se zobrazí, pouze pokud uživatel aplikaci spustí poprvé. Toto je zajištěno tak, že do trvalé paměti aplikace¹ se po ukončení této aktivity uloží proměnná `firstRun` s hodnotou `false`.

Snímek obrazovky výsledku lze vidět na straně 19 (obrázek 2a).

2.1.2 Domovská obrazovka

Domovská obrazovka, neboli sekce „Léky“ (`HomeFragment`), je fragment obsahující pouze `RecyclerView` a `ExtendedFloatingActionButton` (dále jen FAB). FAB se při posunutí seznamu v `RecyclerView` zmenší, zvětší se až když seznam posuneme na začátek.

Pro zobrazení dat v `RecyclerView` je potřeba mít `RecyclerAdapter`. Tato třída se stará o zobrazení dat s příslušným `ViewHolderem`. Adaptér, který je nastavený na tomto `RecyclerView` se jmenuje `AppRecyclerAdapter`. Každý `ViewHolder` pro objekty třídy `Pill` obsahuje kartu, jejíž rozložení je definováno v souboru `item_pill.xml`. Na kartě se zobrazuje všechny potřebné informace o léku: název, popis, barva, fotografie, připomínky a příjem. Pokud uživatel nepotvrdil nejnovější připomínku v posledních 30 minutách, zobrazí se na kartě i výzva k potvrzení. Po kliknutí na kartu se otevře detail příslušného léku. Snímek obrazovky sekce „Léky“ lze vidět na straně 19 (obrázek 2b).

ViewHolder (v doslovém překladu „držitel pohledu“) je třída, která se stará o zobrazení jedné položky v `RecyclerView` adaptéru. Má za úkol nastavit rozložení tak, aby odpovídalo vstupním datům (např. jednomu léku). Pro každý typ položky, který chceme zobrazovat si musíme tuto třídu musíme sami definovat.

AppRecyclerAdapter je `RecyclerAdapter` založený na `ListAdapter`. Tento adaptér se používá pro většinu seznamů v aplikaci, jelikož umožnuje přidat titulek a zobrazit prázdný stav. Toto je dosaženo přepsáním metody `submitList` a dosazením speciálních položek (`HeaderItem` a `EmptyItem`). Adaptér podporuje třídy, které dědí z `BaseModel`, jmenovitě `Pill`, `HistoryPillItem`, `HeaderItem` a `EmptyItem`.

¹Trvalou pamětí je myšleno úložiště SharedPrefs.

2.1.2.1 Detail léku

Obrazovka léku (`DetailsFragment`) je vytořena jako fragment s rozložením `fragment_details.xml`. Na obrazovce lze vidět název léku, jeho popis, fotografii, připomínky, příjem a poslední odeslanou připomínku. Pokud nějakou z položek lék neobsahuje (například fotografii), místo pro ni se skryje. Při kliknutí na fotografii se fotografie zobrazí v plné velikosti. Pokud obrazovku otevřeme z oznámení, nebo má lék nepotvrzenou připomínku v posledních 30 minutách, zobrazí se nad názvem léku karta vyznívající k potvrzení této připomínky. Na spodní části obrazovky jsou tlačítka „smazat“, „historie“ a „upravit“. Tlačítko „smazat“ otevře dialog, kde uživatel může zvolit, zda chce smazat pouze lék a zachovat jeho historii, nebo ho smazat i s historií. Dialog je implementovaný ve třídě `DeleteDialog`. Tlačítko „historie“ otevře dialog, ve kterém se zobrazí historie pro tento lék (`HistoryViewDialog`). Více o tomto dialogu naleznete v kapitole 2.1.3.1. Tlačítko „upravit“ otevře obrazovku `EditFragment`, kde může uživatel lék upravit. Více o této obrazovce v následující sekci.

2.1.2.2 Nový lék

Obrazovka „Nový lék“ (`EditFragment`) má dvě funkce; slouží jako obrazovka pro přidávání nového léku, a zároveň se používá pro úpravu léku. Titulek obrazovky se mění dle použití. Opět je vytořena jako fragment. Pro každý lék je možno nastavit název a popis. Tyto dvě hodnoty se zapisují do prvku `TextInputLayout` z knihovny *material* [12]. Následuje nastavení barvy. Uživatel má na výběr ze sedmi barev: modrá, tmavě modrá, tyrkysová, zelená, žlutá, oranžová a červená. Vybírání barvy je implementováno pomocí prvku `RecyclerView` s atributem `orientation` nastaveným na hodnotu `horizontal`, zobrazí se tedy jako horizontální seznam. Jednotlivé barvy jsou definované třídou `PillColor`. Dále si uživatel nastaví připomínky. Při kliknutí na připomínku nebo na tlačítko „přidat připomínku“ se zobrazí `ReminderDialog`, kde uživatel může upravit/vytvořit připomínku. U připomínek lze nastavit i množství léku, jaké si v daný čas má uživatel vzít. Pro jeden lék nelze nastavit dvě připomínky se stejným časem, každá připomínka musí mít unikátní čas.

Důležitým krokem je léku nastavit příjem. Uživatel si může zvolit, zda lék bere neustále,

jen určitý počet dní, nebo v cyklu X dní aktivních, Y dní neaktivních. Tento prvek¹ je realizován v `PillOptionsView`, který dědí z `LinearLayout` a používá rozložení layout_pill_options_view.xml. Veškerá logika výběru příjmu je implementována v této třídě. Jediné, o co se `EditFragment` musí postarat, je získání `ReminderOptions` (vysvětleno v kapitole 2.2.2) z tohoto prvku pomocí metody `getOptions()`.

K léku lze přidat i fotografiu. Prvek na výběr fotografie je definován v `ImageChooserView`. Tato třída dědí z `LinearLayout` a používá rozložení definované v layout_image_chooser.xml. Pokud lék již nějakou fotografiu obsahuje, prvek zobrazí tlačítko na její odstranění. Při výběru fotografie je použita knihovna *EasyPermissions* [16] pro zajištění potřebných oprávnění a upravená verze knihovny *imagepicker* [18]. Knihovnu jsem upravil tak, aby respektovala vzhled aplikace. Zaprvé již nepoužívá standardní `AlertDialog`, nýbrž `BottomSheetDialog` a také vzhled tohoto dialogu byl upraven, aby odpovídal všem ostatním dialogům v aplikaci. Uživatel si může zvolit, zda chce fotografiu vybrat z galerie, nebo chce vyfotit fotografiu novou. Po vybrání/vyfocení se uživateli ukáže obrazovka, kde může fotografiu upravit. Pro úpravu fotografie jsem použil knihovnu *uCrop* [24], kterou jsem také upravil. Oproti originální verzi se liší v použití prvků na navigaci, ikonách, podporou automatického tmavého vzhledu a sladěním do vzhledu aplikace. Knihovna uživateli umožní fotografiu oříznout, otočit a škálovat. Knihovna má dvě verze: jednu nativní, napsanou v C++, a druhou standardní, napsanou v Javě. I když jsem z počátku zvolil verzi nativní, v konečné verzi aplikace jsem použil standardní verzi knihovny. Důvodů jsem měl hned několik; v nativní verzi nebyly podporované určité formáty fotografií (jmenovitě *.HEIC*), její plynulost nebyla v porovnání se standardní verzí znatelná, a navíc přidala k velikosti aplikace cca 1.5 MB.

Po nastavení všech hodnot může uživatel lék uložit pomocí FAB tlačítka. Aplikace lék vloží do databáze (pokud již existuje, tak jej aktualizuje) a naplánuje jeho následující připomínku.

PillColor je třída, používaná pro ukládání barvy léku. Obsahuje atributy `resource` a `isChecked`. Atribut `resource` ukládá id barvy léku, samotná barva je pak uložená v *App resources* [3]. Atribut `isChecked` vyjadřuje, zda je barva vybraná. Tento atribut se používá k zobrazování seznamu barev a zjištění, jakou barvu uživatel vybral.

¹Prvkem je myšlen prvek uživatelského rozhraní, neboli `view`.

2.1.3 Historie

Druhá hlavní sekce aplikace se nazývá „historie“. Uživatel zde může pozorovat, kdy mu přišly připomínky, kdy a jestli si lék vzal a jaké množství si vzal (nebo s vzít měl). V této sekci jsou dostupné i tři koláčové grafy, které uživateli vizuálně ukazují jeho statistiky. Sekce historie je vytvořena jako fragment, který je prvky **ViewPager2** a **TabLayout** rozdělen na dvě části. Tento fragment (**HistoryFragment**) neobsahuje žádnou logiku ohledně historie, pouze se stará o nastavení výše zmíněných prvků.

ViewPager2 a **TabLayout** jsou prvky, které rozdělují obrazovku na dva nebo více panelů, které jsou přepínatelné pomocí posouvání. **ViewPager2** je nová verze **ViewPager**, která interně využívá **RecyclerView**, jehož položkami jsou fragmenty. **TabLayout** pouze zobrazuje záložky nad panely a dovoluje jejich přepínání pomocí stisku.

2.1.3.1 Přehled

Sekce „přehled“ spadá pod sekci „historie“ a je implementována jako fragment. Tento fragment se zobrazuje jako první položka v prvku **ViewPager2** v sekci „historie“. V tomto fragmentu je pouze prvek **RecyclerView**, který zobrazuje jak veškerou historii, tak historii pro jednotlivé léky. V seznamu se zobrazí karty s názvem, barvou léku a statistikami ukazujícími kolikrát byl lék připomenut, kolikrát byl potvrzen a kolikrát byl vynechán. Po kliknutí na jednu z karet se zobrazí **HistoryViewDialog**. V tomto dialogu má uživatel k dispozici seznam připomínek, které lék obdržel (v případě vybrání karty „Všechny léky“ se zobrazí seznam všech připomínek). Pro každou připomínce může uživatel rozbalit kontextové menu a provést jednu ze čtyř akcí. Uživatel může změnit stav potvrzení připomínky (potvrzená x nepotvrzená), může změnit čas potvrzení (pokud je připomínka potvrzena), může změnit množství, které si v daný čas vzal nebo měl vzít a v neposlední řadě může tuto připomíncu smazat. Pokud by uživatel chtěl smazat všechny připomínky pro daný lék, může stisknout tlačítko smazat nacházející se v titulku dialogu.

Pro tento seznam jsem zvolil rozdílný způsob odlišení různých typů položek. Jelikož jsem potřeboval při zobrazení všech léku zobrazit i název léku a v případě první připomínky dne zobrazit i datum, zvolil jsem pouze jeden **ViewHolder**, který v sobě obsahuje všechny prvky. Toto se liší od způsobu odlišení různých položek v **AppRecyclerView**, kde je využito **ViewHolderů** několik. Rozložení jednotlivých položek je definováno v **item_history.xml**.

Jednotlivé prvku jsou skryty a zobrazeny podle potřeby a rozložení je uděláno tak, aby vypadalo konzistentně při všech typech zobrazení.

Další specialitou tohoto seznamu je aktualizování okolních připomínek při odstraňování. Toto zajistí správné zobrazení data a předělové čáry.

2.1.3.2 Grafy

Druhou sekcí historie je fragment `HistoryChartFragment`, který se stará o zobrazení grafů. Fragment obsahuje `RecyclerView` se třemi kartami s koláčovými grafy. Pro grafy jsem využil knihovnu *MPAndroidChart* [17] dovolující vytvoření velkého množství typů grafů. Já jsem použil jen graf koláčový, protože ostatní grafy by v této aplikaci nedávaly smysl. Všechny grafy v aplikaci jsou procentuální a knihovna si procenta vypočítá sama. Aplikace počítá pouze počet položek, nikoliv jejich procentuální zastoupení. Pokud se v grafu zobrazují jednotlivé léky (první a druhý graf), barva výseče souhlasí s barvou léku. Pokud některý z grafů nemá žádná data, nezobrazí se.

Graf 1 Prvním grafem je graf s názvem „všechny připomínky“. Graf zobrazuje procentuální poměr připomínek jednotlivých léků. Data se získávají následujícím způsobem:

```
val allEntries = ArrayList<PieEntry>()
val pillsHistory = history.groupBy { it.pillId }.values
pillsHistory.forEach { pillHistory ->
    val pill = getPill(pillHistory.first().pillId)
    allEntries.add(PieEntry(pillHistory.size, pill.name))
}
```

Tento kód vypočítá délku každé skupiny historie indikovanou podle ID léku a přidá ji do seznamu používaném pro vykreslení grafu.

Graf 2 Druhý graf na obrazovce pod názvem „vynechané připomínky“ zobrazuje procentuální poměr vynechaných připomínek pro jednotlivé léky. Data pro tento graf se získávají následovně (kód upraven pro snazší orientaci):

```
val missedEntries = ArrayList<PieEntry>()
val pillsHistory = history.groupBy { it.pillId }.values
pillsHistory.forEach { pillHistory ->
    val pill = getPill(pillHistory.first().pillId)
```

```

val pillMissed = pillHistory.count { !it.hasBeenConfirmed }
if (pillMissed > 0) {
    missedEntries.add(PieEntry(pillMissed, pill.name))
}
}

```

Tento kód vypočítá počet nepotvrzených připomínek každé skupinu historie indikovanou podle ID léku, a pokud je větší než nula, přidá jej do seznamu používaném pro vykreslení grafu.

Graf 3 Poslední graf s názvem „potvrzené x vynechané“ zobrazuje procentuální poměr potvrzených a vynechaných připomínek. Data pro poslední graf se získávají tímto způsobem (kód upraven pro snazší orientaci):

```

val totalConfirmed = history.count { it.hasBeenConfirmed }
val totalMissed = history.size - totalConfirmed
val confirmedEntries: ArrayList<PieEntry> = arrayListOf(
    PieEntry(totalConfirmed, R.string.confirmed),
    PieEntry(totalMissed, R.string.missed)
)

```

Výpočet pro tento graf je nejjednodušší, pouze se vypočítá počet potvrzených a počet vynechaných (od velikosti veškeré historie odečtené potvrzené) a tyto hodnoty se přidají do seznamu používaném pro vykreslení grafu.

I když jsem zde v textu kód rozdělil do tří sekcí, v aplikaci je kód na jednom místě (`HistoryChartViewModel`) a využívá pouze jeden cyklus, je tak efektivnější a méně se dotazuje databáze. I když by veškeré získávání dat pro grafy bylo možné přepsat do databázové vrstvy do jazyka SQL, já jsem zvolil tento způsob zpracování dat.

2.1.4 Nastavení

Poslední hlavní sekci aplikace je sekce „nastavení“. Tato sekce je vytvořena pomocí dvou fragmentů: `SettingsFragment` a `PreferencesFragment`. První z těchto fragmentů je rodičem toho druhého. Jediné co obsahuje je titulek „Nastavení“ a pod ním prvek `FragmentContainerView`, do kterého se vkládá právě druhý z fragmentů (`PreferencesFragment`). Tento druhý fragment nedědí ze třídy `Fragment` jako všechny ostatní fragmenty v této

aplikaci, nýbrž ze třídy `PreferenceFragmentCompat`. Díky této třídě nemusíme vymýšlet vlastní rozložení pro nastavení, pouze definujeme položky nastavení v souboru `preferences.xml` (název souboru může být jakýkoliv) a tento soubor použijeme při volání metody `setPreferencesFromResource()`, o zbytek se postará třída. Dále můžeme jednotlivé položky nastavení dynamicky manipulovat nebo můžeme nastavit akce po jejich stisknutí. Já jsem se nadefinoval následující chování položek:

- „Nastavení možností oznámení“ změní svůj popisek, pokud aplikace běží na zařízení se systémem starším než Android 8.0 Oreo, jelikož na těchto verzích nelze nastavit oznámení pro každý lék zvlášť.
- Po kliknutí na „Nastavení možností oznámení“ se otevře android nastavení s obrazovkou pro správu oznámení pro tuto aplikaci.
- Po kliknutí na „O této aplikaci“ se otevře `AboutActivity`.
- Po kliknutí na „Přidat testovací data“ se do aplikace přidají léky a historie určené na testování a demonstraci aplikace.
- Po vybrání nového vzhledu aplikace se aplikace do tohoto vzhledu přepne. Přepínání funguje na úrovni stylů automaticky a je realizováno pomocí metody `AppCompatDelegate.setDefaultNightMode()`

Aby se fragment (`PreferencesFragment`) správně zobrazil ve fragmentu s titulkem (`SettingsFragment`), bylo nutné na jeho interním prvku¹ `listView` změnit určitá nastavení; jmenovitě vypnutí zobrazení `OVER_SCROLL`², vypnutí nastavení `clipToPadding` a přidání dolní mezery `56dp`. Tento fragment vlastní i svůj `ViewModel`, v něm je avšak pouze metoda pro přidání testovacích dat sloužících pro demonstraci aplikace.

2.1.4.1 O aplikaci

Obrazovka „O aplikaci“ je poslední ze tří aktivit, které aplikace obsahuje. Na obrazovce je ikona aplikace, název, jméno autora, verze aplikace a informační položky. Mezi těmito položkami je odkaz na zdrojový kód aplikace na platformě GitHub, odkaz na grafiku

¹Tento prvek nedefinuji já, ale třída `PreferenceFragmentCompat`.

²Zobrazení zpětné vazby pokud uživatel seznam přesune na jeho začátek/konec.

použitou v aplikaci a také informace o knihovnách použitých v aplikaci. Po stisknutí této poslední položky se otevře dialog z knihovny *LicencesDialog* [19] obsahující seznam všech použitých knihoven a jejich licencí definovaných v souboru **notices.xml**

2.2 Programová implementace

V této sekci bych se chtěl věnovat podrobnějšímu popisu implementaci částí aplikace, které nejsou vidět. Popíši, jakým způsobem ukládám a používám data, logiku posílání připomínek a ostatní drobnosti, které bylo nutné v aplikaci zhodnotit.

2.2.1 Databáze

Všechny data jsem se rozhodl ukládat do lokální databáze¹, ze které je mohu efektivně získávat. Pro práci z databází používám jazyk *SQL* v implementaci *SQLite* pod abstraktní vrstvou knihovny *Room* [9]. Tato knihovna mi umožnila jednoduchou práci s databází a jejími tabulkami, aniž bych se musel zabývat vytvářením tabulek, konverzí základních typů nebo manuálními aktualizacemi zobrazených dat.

Jelikož knihovna plně podporuje funkce jazyka Kotlin zvané *Flow* a *coroutine*, nemusel jsem složitě vymýšlet a programovat asynchronním fungováním databáze. Pokud mám návratovou hodnotu metody v databázové vrstvě nastavenou na *Flow<T>*, kde T je jakýkoliv datový typ, získávám z ní aktualizace po celou dobu „sbírání“² hodnoty této metody. *Coroutines* nám narozdíl od *Flow* neumožňují sbírat data kontinuálně, můžeme s nimi avšak v jakémkoliv „coroutinním contextu“³ pracovat, jako bychom pracovali s metodami synchronními. Pro nastavení této vlastnosti na metodě se před její definicí napíše klíčové slovo **suspend**. Všechny tyto možnosti nám umožní práci s daty přesunout z hlavního vlákna na vlákna jiná⁴. Přesunutí je důležité, aby aplikace běžela svižně, neměla záseků a Android nezobrazoval uživateli varování, že aplikace nereaguje.

Abych mohl začít *Room* používat, nejprve jsem musel vytvořit abstraktní třídu mojí databáze. Databáze obsahuje entity (tabulky), které jsou definovány datovými třídami

¹Nacházející se na zařízení uživatele.

²Z anglického výrazu *collect*.

³Z anglického výrazu *Coroutine Context*.

⁴V některých případech nemusí jít ani o vlákno, o toto se stará Kotlin jako takový.

`PillEntity`, `Reminder` a `History`. Ke každé tabulce jsem vytvořil rozhraní označované jako „DAO“ neboli „Database Access Object“, umožňující přímý přístup do databáze. V tomto rozhraní definujeme jaké metody můžeme volat na tabulce (popř. databázi) a jejich SQL příkazy. Příklad takového rozhraní by byl následující (vyjmuto z `PillDao`):

```
@Dao
interface PillDao {
    @Transaction
    @Query("SELECT * FROM pill WHERE deleted = 0 ORDER BY pillId ASC")
    fun getEverythingFlow(): Flow<List<Pill>>
    ...
}
```

Na tomto příkladu můžeme vidět, že `Room` podporuje i automatické transakce. `Pill` totiž není databázová entita, ale relační třída, která spojuje entity `Reminders` a `PillEntity`. Získáme tak snadno přístup ke všem připomínkám u daného léku bez nutnosti psát `INNER JOIN` ručně. Aby třída `Pill` propojení umožňovala, musíme jí zapsat takto:

```
data class Pill(
    @Embedded val pillEntity: PillEntity,
    @Relation(
        parentColumn = "pillId",
        entityColumn = "pillId"
    )
    var reminders: List<Reminder>
)
```

Pokud bychom chtěli do databáze ukládat hodnoty, s kterými si `Room` nebo `SQL` neporadí, musíme je nejdříve převést na hodnoty, kterým tyto knihovny budou rozumět. V méém případě se jednalo o uložení fotografie, barvy léku a času u připomínky. Pro vytvoření takovýchto konvertorů si definujeme speciální třídu, jejíž metody budou anotované pomocí `@TypeConverter`. Tuto třídu přiřadíme k databázi, když databázi definujeme. Pro každou vlastní hodnotu uloženou do databáze musíme mít dvě metody; jednu na převod do formátu, kterému rozumí databáze a druhou na převod zpátky do původního formátu. Například pro uložení třídy `Calendar`, kterou aplikace hojně využívá pro práci s časem, jsem napsal následující konvertory:

```

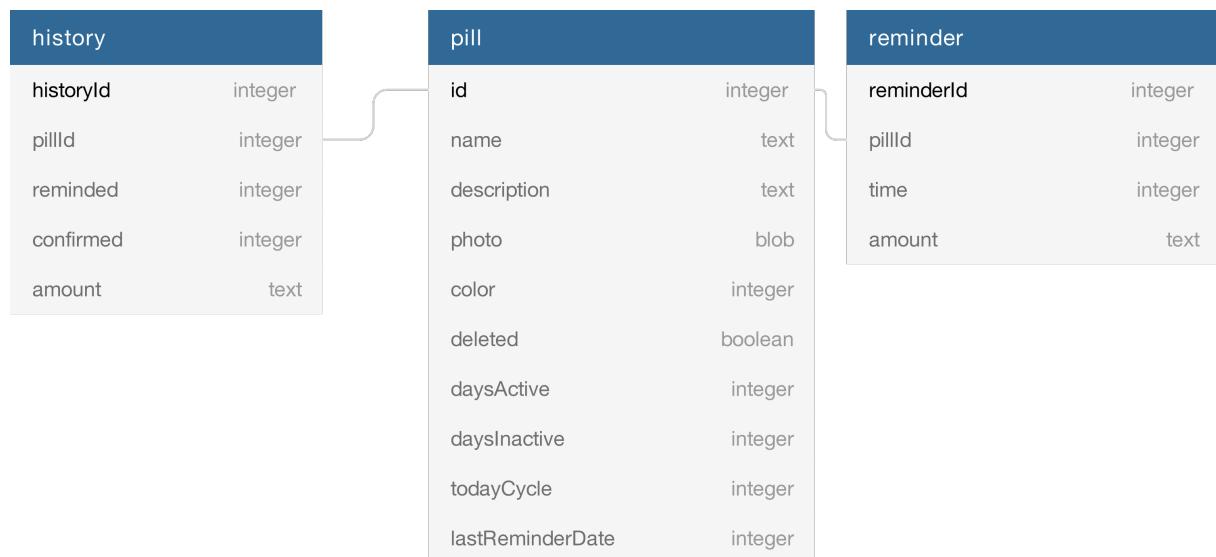
@TypeConverter
fun toCalendar(millis: Long?) = millis?.let {
    Calendar.getInstance().apply { timeInMillis = it }
}

@TypeConverter
fun fromCalendar(calendar: Calendar?) = calendar?.timeInMillis

```

Tímto definujeme jaký formát se uloží do tabulky, v tomto případě se ze třídy `Calendar` stane `Long`, jehož hodnota bude čas vyjádřený v počtu milisekund (pravděpodobně od 1.1.1970, záleží na typu kalednáře/lokalizaci zařízení).

Pro grafické znázornění jsem vytvořil následující digram databáze:



Obrázek 1: Databázový diagram vytvořený pomocí stránky <https://dbdiagram.io/>

2.2.2 Připomínání

Připomínání léků je provedeno standardními oznámeními v systému Android. Pomocí tříd `NotificationCompat` a `NotificationManagerCompat` jsem vytvořil oznámení s nejvyšší možnou prioritou, které obsahuje název léku, popis, říkající jaké množství léku si má uživatel vzít a případnou fotografií. Oznámení má také barvu shodnou s barvou léku. Pod tímto obsahem jsou tlačítka „Potvrdit“ a „Odložit“. Tlačítka mají každé přiřazené svůj `PendingIntent`, který se pošle tzv. *broadcastem* (vysvětleno níže). Při kliknutí na oznámení se uživatel dostane do aplikace, kde bude otevřený lék, jemuž patří daná připomínka. Pokud uživatel připomínku nepotvrdí, aplikace na lék připomene znovu po nastaveném

počtu minut (ve výchozím nastavení 10 minut), maximálně však 6krát (jedno hlavní připomenutí + pět opakování připomenutí).

Broadcasty jsou důležitou součástí celého systému připomínání. V aplikaci existuje celkem 5 tříd, které broadcasty přijímají. Broadcast může být poslán přímo specificky jedné naší třídě (přijímači) nebo jako systémový broadcast (např. spuštění operačního systému). Pro rozesílání broadcastů se používá objekt třídy `PendingIntent`, který v sobě navíc obsahuje `Intent`. Do intentu je možné vložit vlastní hodnoty, které lze v přijímačích získat. Tímto způsobem lze například poslat ID připomínky. Každý z takových to přijímačů musí dědit ze třídy `BroadcastReceiver` a přepsat metodu `onReceive()`. Tato metoda se zavolá pokaždé, když přijímač přijme broadcast. Následuje přehled tříd, které jsou v aplikaci používány na přijímání broadcastů:

<code>BootReceiver</code>	Speciální přijímač, který se zavolá při spuštění operačního systému. Stará se o naplánování připomínek ke všem lékům.
<code>CheckReceiver</code>	Přijímač, který plánuje a zobrazuje oznámení, pokud uživatel připomínu nepotvrdil.
<code>ConfirmReceiver</code>	Přijímač volaný při stisknutí tlačítka <i>Přjmout</i> v oznámení, potvrdí přijmutí léku v danou chvíli.
<code>DelayReceiver</code>	Přijímač volaný při stisknutí tlačítka <i>Odložit</i> v oznámení, odloží oznámení o nastavený počet minut (ve výchozím stavu 30 minut).
<code>ReminderReceiver</code>	Nejdůležitější přijímač, který je volán systémem android v naplánovaný čas. Stará se o zobrazení oznámení, naplánování přijímače <code>CheckReceiver</code> a naplánování sama sebe na další připomínu u léku.

Tabulka 1: Třídy dědící z `BroadcastReceiver`

Pokud bych měl stručně popsat kroky, které vedou k plánování a zobrazení připomínek, budou následující:

1. Při ukládání léku, spuštění operačního systému, nebo spuštění aplikace se zavolá metoda `ReminderManager.planNextPillReminder()`, která plánuje připomínky pro jeden lék.
2. V této metodě:
 - Seřadíme připomínky léku podle času vzestupně.
 - Procházíme jednotlivé připomínky.

- Pokud narazíme na připomínku, která má dnes teprve nastat, naplánujeme ji a metodu ukončíme.
 - Pokud nenařazíme na žádnou připomínku, která nastane ještě dnes, naplánujeme první připomínku zítra a metodu ukončíme.
 - Pokud neexistuje připomínka ani zítra, metodu ukončíme a neplánujeme nic.
3. Pro naplánování používáme metodu `ReminderManager.createAlarm()`. Metoda použije systémovou službu `AlarmManager` a naplánuje s ní „alarm“, který spustí `ReminderReceiver` v čase dané připomínky.
4. V metodě `onReceive` třídy `ReminderReceiver`:
- Pokud toto není první připomenutí léku a zároveň je v daný den připomenutí první, posuneme jeho iteraci cyklu.
 - Zkontrolujeme, zda je lék aktivní (kvůli omezení na počet dní, nebo kvůli cyklu).
 - Pokud lék aktivní je, přidáme do historie toto připomenutí, naplánujeme `CheckReminder` a zobrazíme oznámení.
 - Pokud lék aktivní není a zároveň je omezený na počet dní (tj. už nebude připomínat), odejdeme z metody.
 - Jinak naplánujeme dalsí připomínku a z metody odejdeme.

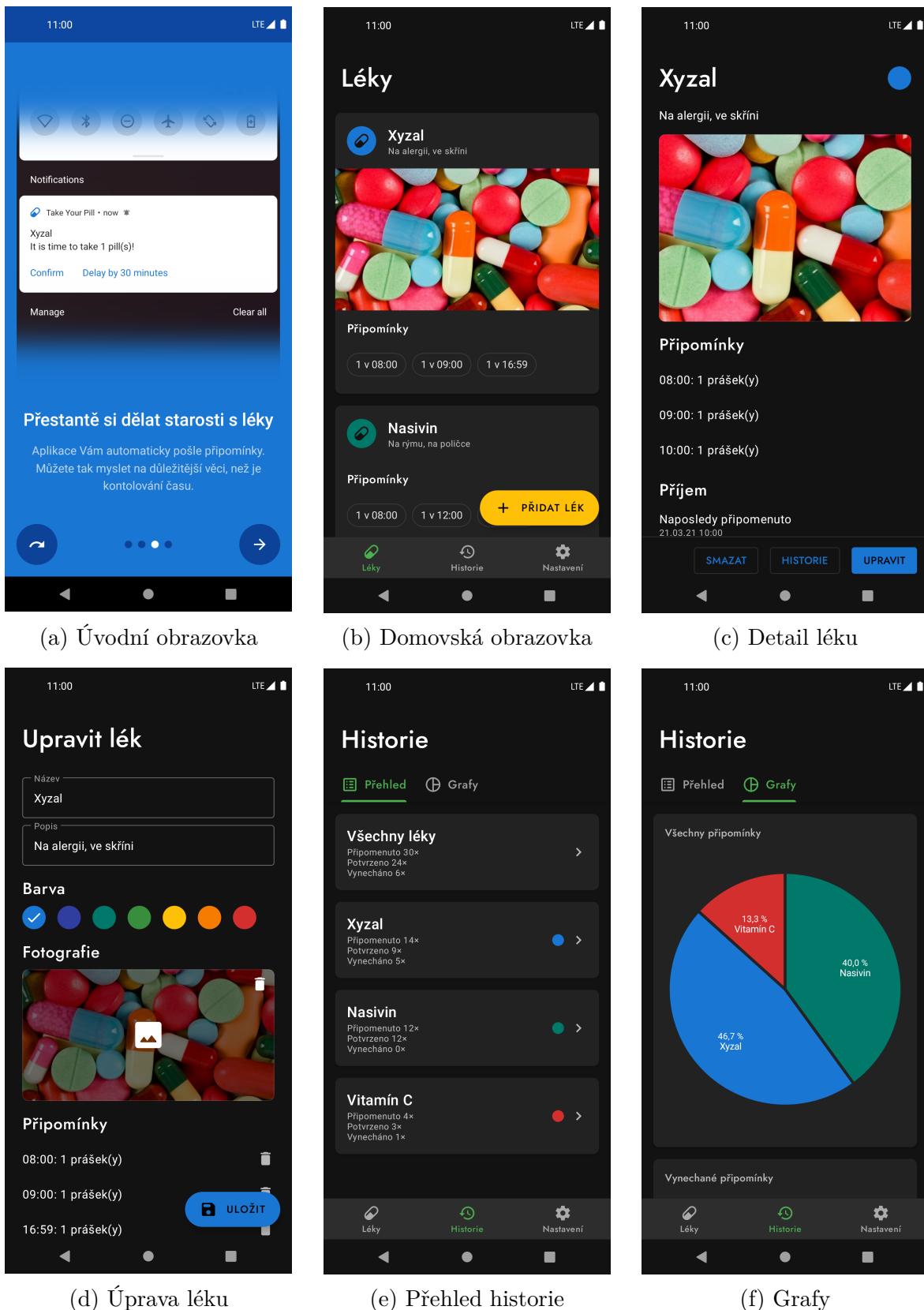
Původně bylo v plánu do aplikace implementovat více varovný režim, kde by se připomínka zobrazila na celou obrazovku. Ukázalo se však, že díky restrikcím, které novější systémy Android obsahují, je toto velice těžko dosažitelné. V aplikaci existují části kódu, které toto zobrazení umožňují, režim jako takový ale není spolehlivý a je proto vypnutý.

Někdy se také může stát, že připomínka dorazí o něco déle, než by měla. V mé testování to bylo maximálně 5 minut. Toto je zapříčiněno uspáním mobilního telefonu, když je obrazovka vypnuta. Android tak automaticky seskupuje *alarmy* a šetří tak baterii, proto se náš broadcast rozešle o něco později. Usuzuji, že je to až moc krátká doba na to, aby

to někomu vadilo, pokud by ale někdo přece jen chtěl, aby připomínky chodili přesně na čas, existuje „řešení“. V informacích o aplikaci v Android nastavení stačí vypnout úsporu baterie pro tuto aplikaci. Má to svůj háček; skoro každý výrobce mobilních telefonů má toto nastavení jinde, někdy jej má dokonce duplicitně, a ne vždy je toto nastavení spolehlivé. Proto jsem do aplikace nedával žádný dialog vyzývající k úpravě tohoto nastavení ve prospěch přesnějšího připomínání.

Závěr

Předložená maturitní práce se skládá za tří částí: mobilní aplikace pro připomínání léků, protokol a informační plakát. Při vlastní práci jsem se snažil dosáhnout toho, aby aplikace byla uživatelsky přívětivá a maximálně spolehlivá. Troufám si říci, že se mi to do uspokojivé míry podařilo. Při vytváření této aplikace jsem se naučil spoustu nových věcí, které jistě využiji ve své profesní kariéře. Překážky, které se při programování vyskytly, jsem se naučil vyřešit a hodně jsem si z nich odnesl. Způsob jakým je aplikace vytvořena umožňuje její rychlou aktualizaci a snadné přidání nových funkcí. Výsledná aplikace může posloužit lidem, kteří musí pamatovat na příjem více či méně léků a také všem, kteří si chtejí tento příjem jednoduše zaznamenávat. Historie v aplikaci může posloužit jako pomůcka při návštěvě lékaře a usnadní tak práci nám, i lékaři.



Obrázek 2: Snímky obrazovky z aplikace

Bibliografie

- [1] Austin Andrews. *pill*. URL: <https://materialdesignicons.com/icon/pill>. [cit. 13. 03. 2021].
- [2] Android Developer. *Android Jetpack*. URL: <https://developer.android.com/jetpack>. [cit. 13. 03. 2021].
- [3] Android Developer. *App resources overview*. URL: <https://developer.android.com/guide/topics/resources/providing-resources>. [cit. 13. 03. 2021].
- [4] Android Developer. *Automated dependency injection*. URL: <https://developer.android.com/training/dependency-injection#automated-di>. [cit. 14. 03. 2021].
- [5] Android Developer. *Dependency injection with Hilt*. URL: <https://developer.android.com/training/dependency-injection/hilt-android>. [cit. 14. 03. 2021].
- [6] Android Developer. *Kotlin flows on Android*. URL: <https://developer.android.com/kotlin/flow>. [cit. 13. 04. 2021].
- [7] Android Developer. *LiveData Overview*. URL: <https://developer.android.com/topic/libraries/architecture/livedata>. [cit. 13. 03. 2021].
- [8] Android Developer. *Navigation*. URL: <https://developer.android.com/guide/navigation>. [cit. 13. 03. 2021].
- [9] Android Developer. *Save data in a local database using Room*. URL: <https://developer.android.com/training/data-storage/room>. [cit. 13. 04. 2021].
- [10] Android Developer. *View Binding*. URL: <https://developer.android.com/topic/libraries/view-binding>. [cit. 13. 03. 2021].

- [11] Google Fonts. *Jost*. URL: <https://fonts.google.com/specimen/Jost>. [cit. 13. 03. 2021].
- [12] Google. *Android - Material Design*. URL: <https://material.io/develop/android>. [cit. 12. 04. 2021].
- [13] Google. *Android Studio*. URL: <https://developer.android.com/studio>. [cit. 13. 03. 2021].
- [14] Google. *Material Design*. URL: <https://material.io/design>. [cit. 13. 03. 2021].
- [15] Google. *Material Design - Icons*. URL: <https://material.io/resources/icons/>. [cit. 13. 03. 2021].
- [16] googlesamples. *EasyPermissions*. URL: <https://github.com/googlesamples/easypPermissions>. [cit. 13. 03. 2021].
- [17] Philipp Jahoda. *MPAAndroidChart*. URL: <https://github.com/PhilJay/MPAndroidChart>. [cit. 14. 03. 2021].
- [18] Dhaval Patel. *Image Picker Library for Android*. URL: <https://github.com/Dhaval2404/ImagePicker>. [cit. 13. 03. 2021].
- [19] PSDev. *LicensesDialog*. URL: <https://github.com/PSDev/LicensesDialog>. [cit. 13. 04. 2021].
- [20] Jan Heinrich Reimer. *material-intro*. URL: <https://github.com/heinrichreimer/material-intro>. [cit. 13. 03. 2021].
- [21] ShimonHoranek. *material-icons*. URL: <https://github.com/ShimonHoranek/material-icons>. [cit. 13. 03. 2021].
- [22] Gabor Varadi. *FragmentViewBindingDelegate-KT*. URL: <https://github.com/Zhuinden/fragmentviewbindingdelegate-kt>. [cit. 14. 03. 2021].
- [23] Gabor Varadi. *Simple one-liner ViewBinding in Fragments and Activities with Kotlin*. URL: <https://zhuinden.medium.com/simple-one-liner-viewbinding-in-fragments-and-activities-with-kotlin-961430c6c07c>. [cit. 14. 03. 2021].
- [24] Yalantis. *uCrop - Image Cropping Library for Android*. URL: <https://github.com/Yalantis/uCrop>. [cit. 13. 03. 2021].