MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

René Gabesam

EasyTrain – tréninková aplikace pro Android

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.
Studijní program: Informatika
Studijní obor: Programování a SW systémy

Praha 2022

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Tímto bych chtěl poděkovat především vedoucímu mé práce Mgr. Pavlu Ježkovi, Ph.D., za jeho čas, pomoc, konstruktivní kritiku, ale i trpělivost při vznikání této práce. Zároveň bych tímto rád poděkoval sourozencům, přátelům a zvláště pak rodičům Mileně a Renému Gabesamovým, kteří mě při mém studiu podporovali.

Název práce: EasyTrain – tréninková aplikace pro Android

Autor: René Gabesam

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Mobilní aplikace dnes hrají velkou roli ve světě osobních tréninkových pomůcek. Zároveň na trhu chybí aplikace specializovaná na práci osobních trenérů a na distribuci cvičení na míru. Cílem této práce je vytvořit mobilní aplikaci na ukládání a sdílení tréninkových aktivit pro zařízení se systémem Android, s využitím frameworku Xamarin Android.

Aplikace jejím uživatelům umožňuje ukládání cviků, tréninků, tréninkových plánů a jejich filtrování pomocí tagů. To je umožněno jak v offline databázi na bázi SQLite, tak v online databázi postavené na Firebase Firestore technologii. Aplikace také umožňuje spustit trénink, aby uživatel mohl dodržet přesné pořadí a množství cviků. Uživatel taktéž může aktivovat tréninkový plán, který ukazuje, v jakém dni plánu se uživatel nachází. Nakonec aplikace přihlášeným uživatelům umožňuje si navzájem posílat zprávy a tréninkové aktivity.

Klíčová slova: Android Xamarin.Android sport trénování Firestore

Title: EasyTrain – Workout Application for Android

Author: René Gabesam

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Nowadays mobile applications are a big part of personal workout accessories. There is a big gap on the market in terms of applications for personal coaches and custom made trainings in general. Goal of this thesis is to create an application for Android devices with Xamarin Android framework meant for saving and distributing training activities.

The application allows users to save their own exercises, trainings, training plans and filter them by tags. This is possible to do locally within the SQLite database as well as online with the Firebase Firestore database. It is also possible to run a training to keep track of exercises and their order and quantity. User also can activate a training plan which keeps track of the day the user is at. Last but not least the application is capable of sending messages and training activities via chat.

Keywords: Android Xamarin.Android sport training Firestore

Obsah

1 Úvod **5**
1.1 Slovník pojmů . . . 5
1.2 Záměr aplikace . . . 6
1.3 Typické využití – scénáře . . . 8
1.4 Rozbor scénářů . . . 13
1.4.1 Datový model . . . 13
1.4.2 Databáze . . . 14
1.4.3 Uživatelský účet . . . 14
1.4.4 Sdílení tréninkových aktivit – chat . . . 15
1.4.5 Spuštění tréninku a tréninkového plánu . . . 16
1.5 Konkurenční řešení . . . 16
1.5.1 Fitify . . . 16
1.5.2 Strava . . . 17
1.5.3 GymKeeper . . . 18
1.5.4 GymRun . . . 19
1.5.5 Shrnutí . . . 20
1.6 Funkční požadavky (Seznam funkcí) . . . 21
1.7 Shrnutí cílů práce . . . 25

2 Analýza, rozbor aplikace **27**
2.1 Framework . . . 27
2.2 Správa uživatelů . . . 28
2.3 Výběr online databáze . . . 28
2.3.1 Azure SQL Database . . . 29
2.3.2 Firebase . . . 29
2.3.3 Výběr online databáze – závěr . . . 30
2.4 Výběr offline databáze . . . 30
2.4.1 Použiti SQLite v naší aplikaci . . . 31
2.5 Fungování Firestore . . . 32
2.6 Modely dat . . . 33
2.6.1 Identifikátory . . . 33
2.6.2 Tag . . . 35
2.6.3 Exercise – cvik . . . 36
2.6.4 Training – trénink . . . 37
2.6.5 Plan – tréninkový plán . . . 39
2.6.6 Spuštěný (aktivní) tréninkový plán . . . 42
2.6.7 Databáze tréninkových aktivit ve Firestoru . . . 43
2.6.8 Chat . . . 43
2.6.9 PersonModel - model uživatele . . . 46
2.7 Bezpečnost uvnitř Firestoru . . . 47
2.8 Uživatelské rozhraní aplikace . . . 49
2.8.1 Rozdělení na obrazovky . . . 49
2.8.2 Dialogy . . . 50
2.8.3 Navigační rozhraní . . . 54
2.8.4 Spravování databáze tagů . . . 55

2.8.5 Prohlížení a editace databáze 56
2.8.6 Spuštěný trénink 61
2.8.7 Spuštěný tréninkový plán 62
2.8.8 Chat, sdílení dat s dalšími uživateli 63

3 Analýza technických řešení 65
3.1 Identifikátory 65
3.2 Cachování tagů 65
3.3 Zacházení se smazanými odkazy 66
3.4 Databáze Firestore 66
3.5 Asynchronní programování 67
3.6 Použití databází 69
3.7 Import dat mezi databázemi 71
3.8 Správa uživatelů 71

4 Implementační dokumentace 73
4.1 Modely 75
4.1.1 TagFilter 78
4.2 Data storage 79
4.2.1 Repositories 79
4.2.2 DatabaseHelpers 80
4.2.3 Online databáze – Firestore 81
4.2.4 Offline databáze – SQLite 83
4.2.5 DatabaseFactory 83
4.3 Services 84
4.4 GUI – Grafické uživatelské rozhraní 84
4.4.1 Aktivity a fragmenty 85
4.4.2 Dialogy 95
4.4.3 Seznamy a AdapterViews 98
4.4.4 Toolbar 103
4.5 Helpers 103
4.5.1 OnClickListenerWrapper 103
4.5.2 OnStartDragListenerWrapper 104
4.5.3 StartDrag events 104
4.5.4 IItemDragHelperAdapter 104
4.5.5 IItemDragAndSwipeHelperAdapter 104
4.5.6 RecyclerView callbacky 105

5 Uživatelská dokumentace 107
5.1 Instalace 107
5.2 Základní práce s databází 108
5.2.1 Cvik 108
5.2.2 Trénink 109
5.2.3 Tréninkový plán 110
5.3 Spouštění tréninku a tréninkového plánu 111
5.4 Registrace – přihlášení 112
5.5 Chat a sdílení tréninků 113
5.6 Ukázková data 114

6 Závěr **115**
6.1 Uživatelské zkušenosti 115
6.2 Zhodnocení cílů 115
6.3 Zhodnocení online databáze 116
6.4 Místo pro rozšíření 116

Seznam použité literatury **119**

Přílohy **123**

1. Úvod

V dnešní době je sport součástí života velké části lidí. Nehledě na sportovní disciplínu, se lidé snaží ustavičně zlepšovat ať už efektivitu tréninku nebo svoje výkony. Tato bakalářská práce se věnuje právě podpoře lidí v jejich zlepšování. Lidé stále hledají nové pomůcky a metody k dosažení jejich cíle. Může jít o cvičební pomůcky, jako například různé posilovací stroje, činky nebo speciálně upravené pomůcky na konkrétní sport. Dále to mohou být organizační pomůcky, například různé aplikace na zapisování statistik a cvičebních plánů, sporttestery na sledování výkonu, nebo si lidé mohou nechat radit od trenérů a vyměňovat zkušenosti s ostatními lidmi. Tato práce si klade za cíl skloubit poslední tři jmenované možnosti. Klade si za cíl zefektivnit organizaci trénování a sdílení zkušeností mezi lidmi.

1.1 Slovník pojmů

V úvodu bylo účelem nastínit potřebu sportujících lidí a v čem bychom jim mohli pomoci. Nicméně pro přesnější stanovení požadavků naší bakalářské práce bychom si potřebovali detailně analyzovat konkrétní potřeby sportovců a lidí pohybujících se v tomto oboru. Abychom tak mohli učinit je potřeba si sjednotit význam některých pojmů, které by čtenář mohl znát, ale nemusel by je chápat ve významu, který zamýšlíme. A proto si zde uveďme krátký slovník pojmů, s tím jak se dané pojmy chápají nejobvykleji a jak je v práci budeme chápat my.

- cvik – anglicky exercise – za cvik považujeme jeden cvičební úkon, například dřep.

- trénink – anglicky training – za trénink považujeme ucelené seskupení cviků v daném pořadí a množství.

- tréninkový plán – anglicky training plan – za tréninkový plán považujeme seskupení dnů, které mohou a nemusí obsahovat nějaké tréninky. Můžeme si to představit jako harmonogram v kalendáři, který říká kdy máme co naplánované a kdy máme naopak volno.

- tag – anglické slovo, česky štítek – tag pro nás bude reprezentovat nějakou vlastnost – přívlastek. Představme si tag jako visačku u nějaké věci, která popisuje čeho se daná věc týká nebo co umí. Tag může říkat, které části těla se cvik týká – např. „posilování zad“ – nebo může označovat, kde se cvik hodí provádět – např. „posilovna“, „fotbalové hřiště“.

- tréninková aktivita – někdy v této práci budeme potřebovat mluvit o cviku, tréninku i tréninkovém plánu najednou. Abychom nemuseli v případě této potřeby dokola psát celý výčet, nazveme cvik, trénink a tréninkový plán souhrnně: „tréninková aktivita“

1.2 Záměr aplikace

Počáteční motivací k vzniku této bakalářské práce bylo, když za námi přišel potenciální zákazník, který dělá florbalového osobního trenéra. Náplní jeho práce je sejít se s jeho svěřencem, provést některé zaměřené cvičení a analyzovat, v čem by se měl daný jedinec zlepšovat. Poté se se svěřencem vidí až za nějaký delší čas, například za měsíc. V tomto měsíci svému svěřenci pravidelně posílá tréninky (každý den, každých pár dní). Doposud si tento trenér všechny cviky a tréninky ukládá do textového souboru nebo dokonce ručně zapisuje na papír. S rostoucím počtem svěřenců mu tento systém přestává vyhovovat, protože pro každého z nich píše všechno znovu. Přišel tedy s tím, že jestli by bylo možné vytvořit mobilní aplikaci na jednoduchou organizaci cviků a tréninků. Chtěl by, aby mu stačilo jednou zapsat cvik a pak ho opakovaně používat v různých trénincích. Zároveň by v takové databázi cviků chtěl mít možnost filtrovat cviky podle jejich vlastností, tedy chtěl mít možnost k cvikům nějaký přívlastek přiřadit – přiřadit tag. S takovým návrhem aplikace by náš zákazník mohl místo každodenního sepisování tréninku, jen sestavit trénink z hotových cviků a poslat ho.

Při tomto rozboru zákazníkova záměru, jsme došli k tomu, že by se pomocí aplikace dala odstranit i nutnost pravidelně posílat tréninky. Chtěli bychom zajistit, aby šlo tréninky posílat například po skupinách nebo ještě lépe přímo naplánovat, kdy má svěřenec jaký trénink odcvičit. Zavedli jsme zde tedy ještě koncept tréninkového plánu, který tento problém řeší.

Dále zde byla otázka distribuce těchto aktivit. První možností byla možnost exportu aktivit do prostého textu a distribuce by probíhala přes jiný komunikační kanál. Trenér by tedy pokaždé musel danou aktivitu přepsat nebo nechat vyexportovat a poslat někde jinde, například jinou chatovací aplikací. Je zde tedy potřeba nějaké další aplikace a to by mohlo odrazovat od užívání naší aplikace. Tento přístup by tedy nebyl moc praktický. Shodli jsme se tedy na druhé možnosti – aplikace by měla obsahovat chat, skz který by trenér mohl tréninkové aktivity rovnou posílat svým svěřencům.

Nakonec jsme ještě přemýšleli nad využíváním aplikace z druhé strany, tedy z pohledu trenérova svěřence. Jeho typické využití by totiž mohlo vypadat tak, že půjde cvičit trenérem naplánovaný trénink a mohl by chtít mít po ruce pomůcku, která mu ukazuje, který cvik je právě na řadě. Shodli jsme se tedy ještě na další funkcionalitě a to sice, že by aplikace měla obsahovat jednoduchou a praktickou možnost k zobrazení právě cvičeného tréninku. Ve stejném duchu jsme ještě přemýšleli nad spouštěním tréninkového plánu. Ten by nebyl jednorázový jako trénink, ale ode dne kdy ho uživatel spustí, by si aplikace pamatovala daný plán a každý den by uživateli ukazovala tréninky na daný den.

Vezmeme-li v potaz podobné mobilní aplikace (viz podkapitola 1.5), nabízí se další funkcionality, především pak nějaké předpřipravené tréninkové aktivity nebo možnost vést si dlouhodobé statistiky o sportovních výkonech. Nicméně naše aplikace je konceptuálně pojatá tak, aby se s její pomocí dal trénovat libovolný sport. Jako předpřipravené, bychom pak mohli vytvořit nějaké aktivity pro sporty, které jsou nejčastější a nejoblíbenější. Je to jistě funkcionalita, kterou by nebylo těžké přidat, ale aplikace je myšlená spíše tak, aby její obsah utvářela komunita. Zavedení statistik v aplikaci by stěžoval právě fakt, že nevíme pro jaký sport statistiku zavádět. Nanejvýš by bylo možné přidat obecnou statistiku, která by

sledovala počet odcvičení daného tréninku nebo cviku a jak dlouho to uživateli trvalo. Tato funkce by aplikaci jistě posunula dál, ale to už považujeme za další úroveň nadstavby. Pro naší práci nepovažujeme předpřipravené cviky nebo vedení statistik za základní funkcionalitu a tedy nebudeme dané funkce implementovat. My se chceme zaměřit na jádro aplikace tvořené organizací tréninkových aktivit a na jejich sdílení v rámci komunity.

Protože cílem aplikace je podpora komunity a nikoliv přidaná hodnota co se samotných tréninkových aktivit týče, dává smysl, aby se aplikace snažila zasáhnout co největší skupinu potenciálních uživatelů. Na základě statistik z webu statcounter [1] o počtu uživatelů jednotlivých operačních systémů, největší množinu potenciálních uživatelů poskytuje operační systém Android, viz Tabulka 1.1. Na základě této statistiky jsme se rozhodli vytvořit aplikaci pro Android. Verzi aplikace pro operační systém iOS považujeme až za další úroveň vývoje a tedy budeme vyvíjet aplikaci pouze pro Android.

Dále bylo potřeba vzít v úvahu i verzi systému Android. Znovu na základě statistik z webu statcounter [2] o poměru používaných verzí Androidu, viz Tabulka 1.2, můžeme nahlédnout, že pokud budeme podporovat verze 8.1 a vyšší, budeme podporovat 84.56 % uživatelů což je pro nás dostačující dosah.

Nakonec je třeba zmínit, že naše práce není určena komerčnímu užití, protože to není práce na zakázku, ale aplikace na podporu sportovní komunity. Aplikace by v začátcích ani nemohla mít žádné zpeněžitelné ambice, protože ji nikdo nebude znát a tedy by nebyl patrně nikdo, kdo by za ní byl ochotný dát nějaké peníze. Zároveň sama aplikace nenabízí žádné nové informace – žádné předpřipravené tréninky nebo revoluční způsoby vedení statistik – tedy by ani z tohoto hlediska nedávalo smysl zkoušet aplikaci jakkoliv prodat. Budeme tedy pro naši práci hledat řešení, která ani nás nebudou stát žádné peníze, protože aplikace si na sebe zatím nevydělá.

	Android	**iOS**
celý svět	75.75 %	28.53 %
Evropa	64.02 %	35.42 %
Česká Republika	71.47 %	27.91 %

Tabulka 1.1: Podíl dvou hlavních telefonních operačních systémů, podle webu statcounter [1].

Android verze	**podíl**
11.0	35.02 %
10.0	28.29 %
9.0	13.84 %
8.1	7.41 %
8.0	3.21 %
7.1	1.94 %

Tabulka 1.2: Podíl verzí systému Android od nejnovější verze, podle webu statcounter [2]

1.3 Typické využití – scénáře

V předchozí podkapitole jsme si ujasnili základní funkce naší aplikace. V rámci těchto funkcí chceme, aby naše aplikace byla pro uživatele co možná nejpřívětivější. Tudíž bychom si chtěli promyslet typické příklady použití naší aplikace uživatelem, na kterého cílíme. Na základě těchto scénářů – příkladů použití – se rozhodneme na přesnější podobě funkcí, respektive si rozmyslíme případné další drobné funkce, které by se uživateli mohli hodit. V této podkapitole je tedy seznam typických scénářů použití, ze kterých bude možné identifikovat konkrétní funkce, které by aplikace měla mít.

1. **Scénář – uložení cviků**

   Představme si pravidelně cvičícího uživatele. Takový uživatel časem sbírá více a více cviků, které cvičí nebo je někdy bude chtít cvičit. V malých počtech cviků je možné si to všechno pamatovat, ale s rostoucím počtem roste i potřeba si všechno někam zapisovat. Uživatel by tedy chtěl mít v aplikaci možnost ukládat cviky. Cvik nemusí nutně být něco obecně známého, např. `dřep`, ale klidně nějaký uživatelem vymyšlený, složitější cvik. S tím vzniká potřeba si o cviku zapsat i nějaký popisek, jak má daný cvik vypadat nebo co k němu budu potřebovat.

2. **Scénář – editace a mazání cviků**

   Uživatel již má v aplikaci vytvořenou rozsáhlou databázi cviků. V rámci komunikace s jiným sportovcem zjistí, že některý z jeho cviků doteď cvičil špatně, bude si tedy chtít upravit popisek daného cviku. Uživatel také může zjistit, že některé cviky jsou dnes už zastaralé a chtěl by si je z databáze smazat. Nebo může chtít starý cvik nahradit jiným, což docílí přejmenováním starého a změnou jeho popisu.

3. **Scénář – tagování cviků – přiřazování vlastností**

   Uživatel má v databázi několik cviků do posilovny a několik cviků na ven. Když pak přijde do posilovny, chtěl by si vytřídit jen ty cviky, které jsou do posilovny. Uživatel by tedy potřeboval systém, který by cvikům dokázala přiřadit vlastnost, např. `posilovna` nebo `na ven`. Jinak řečeno, uživatel typicky chce vytřídit cviky s nějakou danou vlastností. K tomu ale nejdřív potřebuje systém k přiřazení těchto vlastností.

4. **Scénář – třídění cviků podle vlastností – konjunkce**

   Uživatel má databázi cviků ze scénáře 3 a již cviky mají přiřazené všechny vlastnosti. Uživatel si chce najít všechny cviky, které jsou do posilovny a na posilování zádového svalstva. Chce tedy zkombinovat vlastnost `posilovna` a `posilování zad`. Výsledkem takového filtru by měly být ty cviky, které mají **obě** vlastnosti.

5. **Scénář – třídění cviků podle vlastností – disjunkce**

   Uživatel má databázi cviků ze scénáře 3 a již cviky mají přiřazené všechny vlastnosti. Uživatel si chce najít všechny cviky, které jsou na posilování zádového nebo břišního svalstva. Chce tedy zkombinovat vlastnosti

`posilování břicha` a `posilování zad`. Výsledkem takového filtru by měly být ty cviky, které mají **alespoň jednu** z jmenovaných vlastností.

6. **Scénář – uložení tréninků**

   Uživatel má databázi cviků do posilovny a chce vytvořit trénink, který bude pravidelně cvičit, na posílení zádového svalstva. Nechce si to muset pamatovat, ale napíše si to do aplikace. V tréninku by měl v sérii po deseti, vystřídá 3 cviky na různých posilovacích strojích a to celé 3x opakovat. Jinak řečeno, trénink bude chronologicky následující: `10x cvik 1, 10x cvik 2, 10x cvik 3, 10x cvik 1, 10x cvik 2, 10x cvik 3, 10x cvik 1, 10x cvik 2, 10x cvik 3`. Zároveň si k tréninku napsal popisek. Například, že daný trénink se hodí na nabrání velkého objemu svalové hmoty.

7. **Scénář – editace tréninků**

   Uživatel má vytvořený trénink ze scénáře 6 a zjistí, že je příliš náročný. Chce tedy odstranit třetí opakování všech tří cviků a v druhém opakování chce snížit počet cviků na 5. Později si ještě uvědomí, že by bylo lepší, aby cvik 2 předcházel cviku 1 a bude chtít změnit jejich pořadí. Výsledný trénink by měl chronologicky vypadat následovně: `10x cvik 2, 10x cvik 1, 10x cvik 3, 5x cvik 2, 5x cvik 1, 5x cvik 3`.

8. **Scénář – mazání tréninků**

   Uživatel má vytvořený trénink na posilování. Po poradě s trenérem zjistí, že se pro něj daný trénink vůbec nehodí a chce ho tedy ze své databáze smazat aby se mu již nepletl.

9. **Scénář – třídění tréninků podle vlastností – konjunkce**

   Uživatel má databázi tréninků a chce si najít všechny tréninky, které obsahují cviky na posilování zádového a břišního svalstva. Chce tedy zkombinovat vlastnost `posilování břicha` a `posilování zad`. Výsledkem takového filtru by měly být ty tréninky, které obsahují cviky s vlastností `posilování zad` **a** cviky s vlastností `posilování břicha`.

10. **Scénář – třídění tréninků podle vlastností – disjunkce**

    Uživatel má databázi tréninků a chce si najít všechny tréninky, které obsahují cviky na posilování zádového nebo břišního svalstva. Chce tedy zkombinovat vlastnost `posilování břicha` a `posilování zad`. Výsledkem takového filtru by měly být ty tréninky, které obsahují cviky s vlastností `posilování zad` **nebo** cviky s vlastností `posilování břicha`.

11. **Scénář – ukládání tréninkových plánů**

    Uživatel má databázi různých tréninků. Setká se se svým osobním trenérem, který mu poradí jak by si tréninky měl dávkovat. Konkrétně pak vytvoří plán na týden: v pondělí, úterý, čtvrtek a sobotu by měl uživatel odcvičit trénink č.1, ve čtvrtek a pátek by měl odcvičit trénink č.2 a v sobotu a neděli by měl odcvičit trénink č.3. V plánu tedy je i volný den ve středu. Uživatel by chtěl mít možnost si tréninkový plán složit právě z tréninků, které má již v aplikaci uložené.

12. **Scénář – editace tréninkových plánů**

    Uživatel má tréninkový plán ze scénáře 10. Zjistí, že by potřeboval dva volné dny v úterý a čtvrtek, naopak ve středu by chtěl cvičit, ale 2x trénink č.1. Později se uživatel ještě rozhodne zvětšit svůj tréninkový plán na 14 dní z původních 7. Omylem však přidá i 15. den a ten ještě potřebuje dodatečně smazat.

13. **Scénář – mazání tréninkových plánů**

    Uživatel má od svého trenéra vytvořený tréninkový plán na dosažení nějaké určité výkonnosti. Této výkonnosti ovšem už dosáhl a plán je tedy pro něj zastaralý a chtěl by se ho zbavit.

14. **Scénář – třídění tréninkových plánů podle vlastností – konjunkce**

    Uživatel má již databázi tréninkových plánů. A protože v poslední době strávil spoustu času v kanceláři, chce si najít tréninkový plán, který obsahuje cvičení na břišní a zádové svalstvo. Chce si tedy vyfiltrovat všechny plány obsahující takové tréninky, které obsahují cviky na posilování zádového a břišního svalstva. Potřebuje tedy zkombinovat vlastnost `posilování břicha` a `posilování zad`. Výsledkem takového filtru by měly být ty plány obsahující tréninky, které obsahují cviky s vlastností `posilování zad` **a** cviky s vlastností `posilování břicha`.

15. **Scénář – třídění tréninkových plánů podle vlastností – disjunkce**

    Uživatel má již databázi tréninkových plánů. A protože v poslední době vůbec nesportoval, potřeboval by posílit svaly a zlepšit kondici. Chce se ale v jednu chvíli zaměřit pouze na jednu věc. Chce si tedy vyfiltrovat všechny plány obsahující takové tréninky, které obsahují cviky na posilování nebo cviky na zlepšení kondice. Chce tedy zkombinovat vlastnost `posilování` a `kondice`. Výsledkem takového filtru by měly být ty plány obsahující tréninky, které obsahují cviky s vlastností `posilování` **nebo** cviky s vlastností `kondice`.

16. **Scénář – spuštění tréninku**

    Uživatel jde do tělocvičny a chce si nějak jednoduše zobrazit trénink, který má v databázi samostatně nebo je součástí nějakého tréninkového plánu, jenž se snaží dodržovat. V zobrazení daného tréninku by uživatel chtěl možnost nějak jednoduše zobrazit, který cvik je právě na řadě, odškrtávat nebo jinak zaznamenávat, že daný cvik už je hotový a prohlížet si které cviky ho ještě čekají.

17. **Scénář – spuštění tréninkového plánu**

    Uživatel dostane od svého trenéra nebo si sám vytvoří tréninkový plán na posílení zádového a břišního svalstva. Tréninkový plán může být na delší dobu a tedy by uživatel chtěl, když plán spustí, aby si nemusel pamatovat, který den plánu je na řadě. Chtěl by tedy v aplikaci aktivovat tréninkový plán a každý den se jen podívat do aplikace, který trénink (tréninky) je na řadě.

18. **Scénář – offline databáze**

    Všechny předchozí scénáře nevyžadují přihlášení. Může být uživatel, který aplikaci bude používat jen pro sebe, tedy nebude mít potřebu někomu posílat svoje tréninkové aktivity nebo naopak je od někoho přijímat. Pak si uživatel nemusí chtít vytvářet v aplikaci účet a používat aplikaci kompletně offline.

19. **Scénář – registrace**

    Uživatel již má ve své databázi různé tréninkové aktivity a chtěl by se o ně podělit se svými přáteli. Aby tak mohl v rámci aplikace učinit, potřebuje nejdříve mít účet k identifikaci jeho samého. Uživatel si tedy bude chtít vytvořit v aplikaci účet. Aby se mu do účtu nikdo nenaboural a například nesmazal nějaká data, měl by účet být zabezpečen heslem. Zároveň aby byl účet jednoznačně určený měl by být identifikovaný přihlašovacím jménem – např. e-mailem.

20. **Scénář – přihlášení**

    Prerekvizita: být registrován

    Uživatel chce mít svoji databázi přístupnou z různých zařízení nebo chce poslat tréninkové aktivity někomu ze svých přátel. Aby to bylo možné, uživatel se nejprve musí přihlásit do svého účtu pomocí svého přihlašovacího jména a hesla.

21. **Scénář – používání bez nutnosti se pokaždé přihlašovat**

    Prerekvizita: být přihlášen

    Uživatel je přihlášený a zavře aplikaci. Později, když potřebuje aplikaci znovu použít, zapne ji a rád by se vyhnul opakovanému přihlašování. Uživatel by po znovuotevření aplikace chtěl být stále přihlášen k účtu, ke kterému byl přihlášen při posledním použití aplikace.

22. **Scénář – online databáze**

    Prerekvizita: být přihlášen

    Uživatel má v aplikaci účet, protože chce mít svoje tréninkové aktivity spojené se svým účtem, nikoliv s mobilním zařízením. Chce mít tedy tréninkové aktivity zálohované a dostupné z libovolného zařízení. Uživatel chce aby se toto dělo automaticky, když je přihlášený.

23. **Scénář – možnost importu dat z offline databáze do online databáze po registraci**

    Prerekvizita: být registrován

    Uživatel může dlouho používat aplikaci pouze offline a mít zde již rozsáhlou databázi tréninkových aktivit. Když se z nějakého důvodu (viz scénáře 19, 20) rozhodne vytvořit si v aplikaci účet, bude chtít i nadále mít svoji databázi tréninkových aktivit z dřívějšího užívání aplikace. Zároveň zde může být uživatel, který chce oddělit soukromou databázi tréninkových aktivit a databázi tréninkových aktivit, které chce posílat dalším lidem. Takový uživatel naopak po registraci bude chtít mít prázdnou online databázi. Uživatel

se po registraci bude chtít rozhodnout, jestli má aplikace spojit uživatelovy offline data s uživatelovým novým účtem.

24. **Scénář – možnost importu dat z offline databáze do online databáze – dodatečně**

    Prerekvizita: být přihlášen

    Uživatel může mít oddělenou offline databázi a online databázi (viz scénář 22). Může ale chtít mít některé tréninkové aktivity dostupné i z jiných zařízení nebo je mít zálohované. Aby se tak stalo, potřebuje být přihlášen a mít je uložené v online databázi a mít je tedy spojené se svým účtem. Tyto tréninkové aktivity ale momentálně má pouze v offline databázi. Uživatel by tedy chtěl importovat libovolnou tréninkovou aktivitu z offline databáze do online databáze.

25. **Scénář – nalezení jiného uživatele**

    Prerekvizita: být přihlášen

    Uživatel chce jinému uživateli poslat jednu ze svých tréninkových aktivit. V případě, že si tito dva uživatelé navzájem zatím nic neposlali, potřebuje nejdříve druhého uživatele najít v seznamu všech uživatelů. V seznamu všech uživatelů by chtěl vyhledávat podle jména.

26. **Scénář – poslání libovolné tréninkové aktivity nebo textové zprávy jinému uživateli**

    Prerekvizita: být přihlášen

    Uživatel chce poslat jinému uživateli jednu ze svých tréninkových aktivit. Najde chat s daným uživatelem, otevře ho a pošle danou tréninkovou aktivitu. K odeslané aktivitě může ještě chtít poslat druhému uživateli komentář k aktivitě, co mu právě poslal – chce tedy poslat textovou zprávu.

27. **Scénář – uložení libovolné tréninkové aktivity z chatu s jiným uživatelem k sobě do databáze**

    Prerekvizita: být přihlášen

    Uživatel v chatu obdrží od svého přítele tréninkovou aktivitu. Chce si ji následně uložit k sobě do databáze aby ji mohl použít při cvičení.

28. **Scénář – odhlášení**

    Prerekvizita: být přihlášen

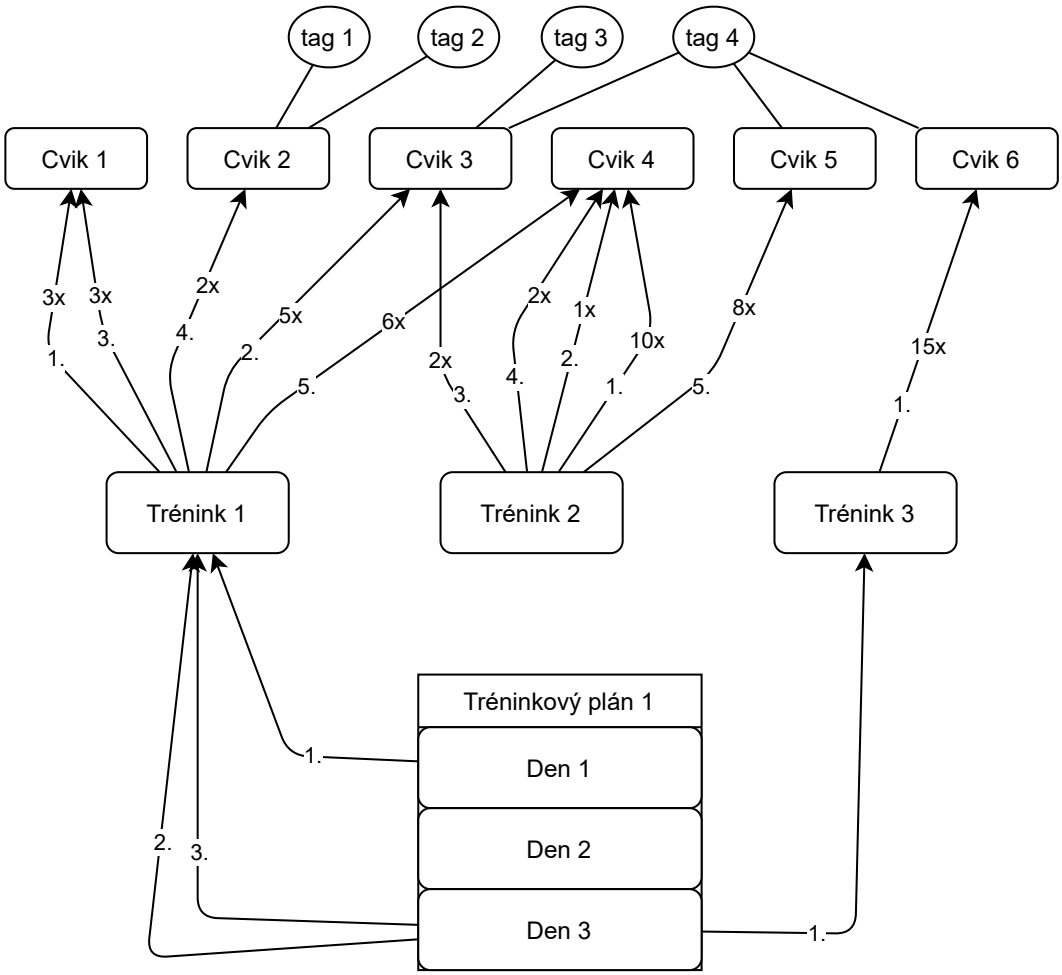
    Uživatel se může setkat s dalším sportovcem používajícím naši aplikaci. Druhý sportovec zapomněl svůj mobilní telefon, ale rád by našemu uživateli ukázal něco ze své databáze. Pokud používá aplikaci jako registrovaný uživatel, jsou po přihlášení jeho cviky dostupné na libovolném zařízení. Náš uživatel se chce na svém zařízení odhlásit, aby se druhý sportovec mohl přihlásit.

1.4 Rozbor scénářů

Na základě podkapitol 1.2 a 1.3, si detailněji rozmyslíme, jak by se aplikace z pohledu uživatele měla chovat.

1.4.1 Datový model

Hlavním pilířem naší práce bude datový model tréninkových aktivit, s jakým bude uživatel operovat. Schéma je znázorněno na obrázku 1.1. Vidíme zde vytvořené cviky 1–6. Cviky mají každý libovolný počet tagů, reprezentující nějakou vlastnost daného cviku. Nedává smysl, aby se vlastnost mohla u cviku opakovat, ale může jich být libovolný počet. Na jeden cvik se může odkazovat libovolný počet tréninků v libovolném množství. Máme zde *Trénink 1*, který obsahuje 2x3x *Cvik 1*, 2x *Cvik 2*, 5x *Cvik 3* a 6x *Cvik 4*. Ve schématu je naznačeno pořadí cviků v daném tréninku, tedy na pořadí cviků v tréninku záleží. *Trénink 1* ze schématu je posloupnost cviků: *3x Cvik 1, 5x Cvik 3, 3x Cvik 1, 2x Cvik 2, 6x Cvik 4*. Dále na obrázku máme jediný tréninkový plán. Tréninkový plán se skládá ze dnů, kde každý den se může odkazovat na libovolný počet tréninků. Analogicky k tréninku a cvikům i zde na pořadí tréninků v daném dnu záleží. Tréninkový plán může obsahovat i den bez tréninku – volný den.



Obrázek 1.1: Datový model z pohledu uživatele

Chceme, když uživatel aktualizuje nějaký cvik, aby se daný cvik aktualizoval i ve všech trénincích, ve kterých se nachází. Stejně to platí pro tréninky a tréninkové plány. To znamená, že tréninky i tréninkové plány by měli používat odkazy na aktivity které agregují. Tato funkcionalita dovoluje uživateli jednoduše nahradit např zastaralý cvik v tréninku (viz scénář 2), obecně aktualizovat cvik/trénink a nechat změnu propsat do tréninku/tréninkového plánu. Tento model má i svoje negativa a to sice, když uživatel smaže cvik, v tréninku který daný cvik obsahuje, se najednou zneplatní reference a tedy se cvik z tréninku odstraní. Analogicky pak trénink a tréninkový plán.

Vlastnost cviku je sice něco co už existuje, tedy to nevytváří uživatel. Mohlo by tedy vypadat, že nebude potřeba udržovat databázi tagů, tedy vlastností. Nicméně abychom mohli cviky podle vlastností třídit, musí mít uživatel množinu vlastností, ze které bude vybírat. Vlastností je totiž jinak potenciálně nekonečno. Je tedy potřeba udržovat i databázi těchto vlastností – tagů. Uživateli by mělo být umožněno ke cviku přiřadit i tag, který ve své databázi zatím nemá. Tím se tag vloží do uživatelovy databáze tagů a zároveň se přidá k danému cviku. Počet tagů takto může docela rychle růst a uživatel by někdy mohl některý existující tag smazat. Není to klasický scénář, ale spíše následek scénáře 3. Smazání tagu může mít za následek, analogicky ke cvikům, odebrání daného tagu od všech cvičení v databázi.

1.4.2 Databáze

Důležitou částí naší práce jsou databáze. Aplikace by měla nabídnout lokální – offline databázi těm uživatelům, kteří by aplikaci používali pouze pro uspořádání svých dat a neměli by žádnou potřebu data zálohovat nebo sdílet s kýmkoliv jiným. Není zde ale žádný důvod, aby uživatel musel s aplikací pracovat jinak, než kdyby používal online databázi. Aplikace by tedy měla umět použít správnou databázi nehledě na to zda-li je uživatel přihlášený nebo ne a chovat se navenek stejně. Registrovaný a přihlášený uživatel pak pouze získá víc možností jak aplikaci využívat. Aplikace přihlášenému uživateli umožní posílat a přijímat všechny tréninkové aktivity skrze chat a umožní uživateli přihlásit se na jiném zařízení a mít k dispozici celou svoji databázi.

V databázi (online i offline), by uživatel měl být schopen procházet seznam všech tréninkových aktivit. Cviky, tréninky i tréninkové plány by ale měly být pro přehlednost oddělené. Dále by v této databázi měl být uživatel schopen filtrovat libovolnou aktivitu podle tagů – viz scénáře 4, 5, 9, 10, 14, 15.

1.4.3 Uživatelský účet

Účet v aplikaci by měl sloužit pouze jako identita daného uživatele. Účet je v aplikaci důležitý pouze kvůli identifikaci dat náležejících k danému účtu a k identifikaci uživatelů navzájem. Není tedy potřeba aby účet nabízel rozšířené možnosti nastavení účtu, jako například profilový obrázek, detaily o uživateli apod. Z hlediska aplikace je důležité aby se alespoň základně bránila zahlcení „fake“ účty. Každý účet jistě bude vytvářet nějaká data, která budou muset být uložena v online databázi. Zároveň s rostoucím množstvím dat nebo počtem čtení a zápisů do/z online databáze, mohou růst i náklady na provoz online databáze.

Pravděpodobně nejsnazším a dnes velmi rozšířeným způsobem, jak zabránit vytváření „fake“ účtů, je povinnost svázat účet s e-mailovou adresou. Dnešní uživatelé jsou na tento postup již určitě navyklí. Aplikace by tedy měla umožňovat vytvořit účet pomocí e-mailu. Druhou variantou by bylo spojení účtu v aplikaci s jiným, již existujícím, účtem u nějaké důvěryhodné společnosti, například Google účet, Facebook účet apod. To je funkce, kterou dnes nabízí velká část aplikací na trhu. Na základě vybrané technologie (viz 2.3), by nebylo těžké tuto funkci přidat, nicméně je to funkce navíc, o kterou nám momentálně nejde a tedy ji v základu nebudeme implementovat.

1.4.4 Sdílení tréninkových aktivit – chat

Pro sdílení tréninkových aktivit jistě existuje více způsobů řešení. Když se podíváme na již existující řešení sdílení libovolných dat, nabízí se například e-mail – oddělené zprávy s přílohou i textovou zprávou nebo různé sdílené uložiště jako například Google Drive, kde lze nahrát libovolný soubor a tedy i textový soubor. Nicméně pro uživatele mobilních aplikací je dnes pravděpodobně nejpohodlnější forma chatu, kde základ tvoří textové zprávy, ale dají se zde většinou posílat i různé soubory. U chatovacích aplikací jistě není nejpoužívanější funkcí dohledávání starých zpráv a nahlížení do historie, ale občas to uživatel chce a tyto aplikace to i umožňují. Stejně tak můžeme předpokládat u uživatelů naší aplikace, že dohledávání starých zpráv a poslaných tréninkových aktivit nebude to hlavní, ačkoliv dostupné by to zde mělo být. Uživatel typicky přijme nějakou tréninkovou aktivitu a uloží si ji k sobě do databáze a pravděpodobně se znova k dané aktivitě v chatu vracet nebude.

Uživatel jistě bude předpokládat, že tagy by se měli dát použít i na vyhledávání tréninkové aktivity, kterou chce poslat. Také by se tréninková aktivita mohla dát vyhledávat podle jména. Dále z hlediska uživatele, když chce odeslat libovolnou tréninkovou aktivitu, jistě bude předpokládat, že se zároveň s tím pošlou i všechny obsažené tréninkové aktivity. Konkrétně, když bude chtít uživatel poslat trénink, bude předpokládat, že se zároveň s tréninkem pošlou i všechny cviky obsažené v tréninku a zároveň všechny tagy, které jsou použity danými cviky. Dále můžeme předpokládat, že by se chtěl vyhnout posílání duplicit. Tedy například v situaci, kdy pošle trénink s dvěma identickými cviky, nechce aby druhý uživatel, po uložení do své databáze, měl cvik, který je zdvojený v tréninku, zdvojený i v jeho databázi. Ukážeme si to na příkladu z obrázku 1.1. Uživatel chce poslat `Tréninkový plán 1`. Kromě samotného plánu se pošlou tréninky 1, 3 a tedy se pošlou i všechny cviky z obou tréninků, tzn. všechny cviky kromě cviku 5. Většina cviků má přiřazenou i nějakou vlastnost – tag – a tedy se pošlou i dané tagy. Vše se ovšem pošle jen v jedné kopii, včetně tagů. Navíc by se zde dalo předpokládat, že pokud se zaslaný tag jmenuje stejně jako nějaký tag u příjemce v databázi, mají i stejný význam. Zde vycházíme jednoduše z toho, že pojmenování vlastností nebývá něco, co by si uživatel vymyslel, ale slovo s ustáleným významem. Nakonec je třeba zmínit, že posílání samostatných tagů nedává z pohledu uživatele smysl. Jak bylo řečeno, tag je zamýšlen jako vlastnost. Vlastnost dává smysl jen když je k něčemu přiřazená a tedy dává smysl ji posílat jen s něčím.

1.4.5 Spuštění tréninku a tréninkového plánu

Aplikace by krom organizace a sdílení dat, měla být nápomocná i při samotném trénování. Když uživatel obdrží od jiného uživatele trénink, pravděpodobně si nebude hned pamatovat všechny cviky, jak jdou v tréninku za sebou a v jakém jsou množství. Aplikace by kvůli tomuto problému měla poskytovat možnost spuštění tréninku, když jde uživatel cvičit. Tato funkce pak uživateli bude ukazovat cvik, který je právě na řadě ve spuštěném tréninku. Uživateli se zde zobrazí všechny dostupné detaily o daném cviku a kolikrát ho má opakovat. Když uživatel daný cvik odcvičí, posune se na další v pořadí.

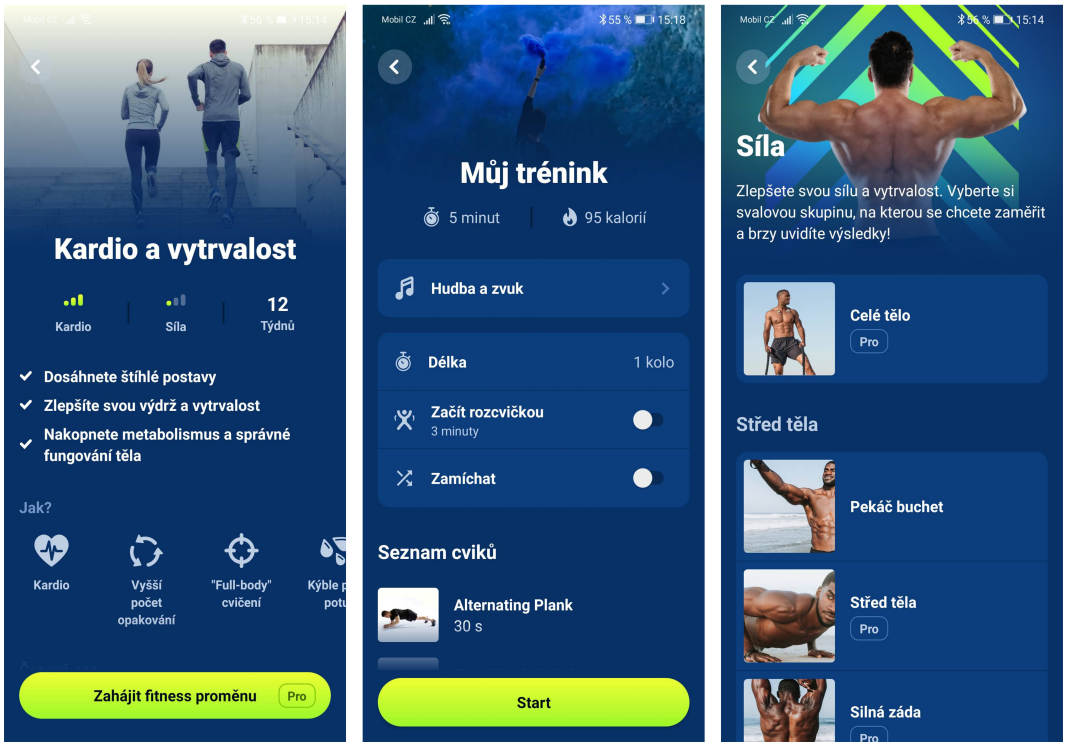
Ještě těžší pro uživatele bude pamatovat si celý tréninkový plán včetně tréninků v něm. Aplikace by tedy pro tento účel měla poskytnout možnost spustit tréninkový plán, aby uživateli stačilo sledovat v aplikaci, který den a které tréninky jsou v daném tréninkovém plánu právě na řadě.

1.5 Konkurenční řešení

V této podkapitole se podíváme na mobilní aplikace, které se také věnují sportu. Porovnáme základní uživatelské funkce, které jsme si vypsali v předchozí podkapitole, s funkcemi vybraných aplikací. Ve výběru figurují aplikace, které jsou na trhu nejpoužívanější nebo ty, které se nejvíce blíží myšlence naší práce.

1.5.1 Fitify

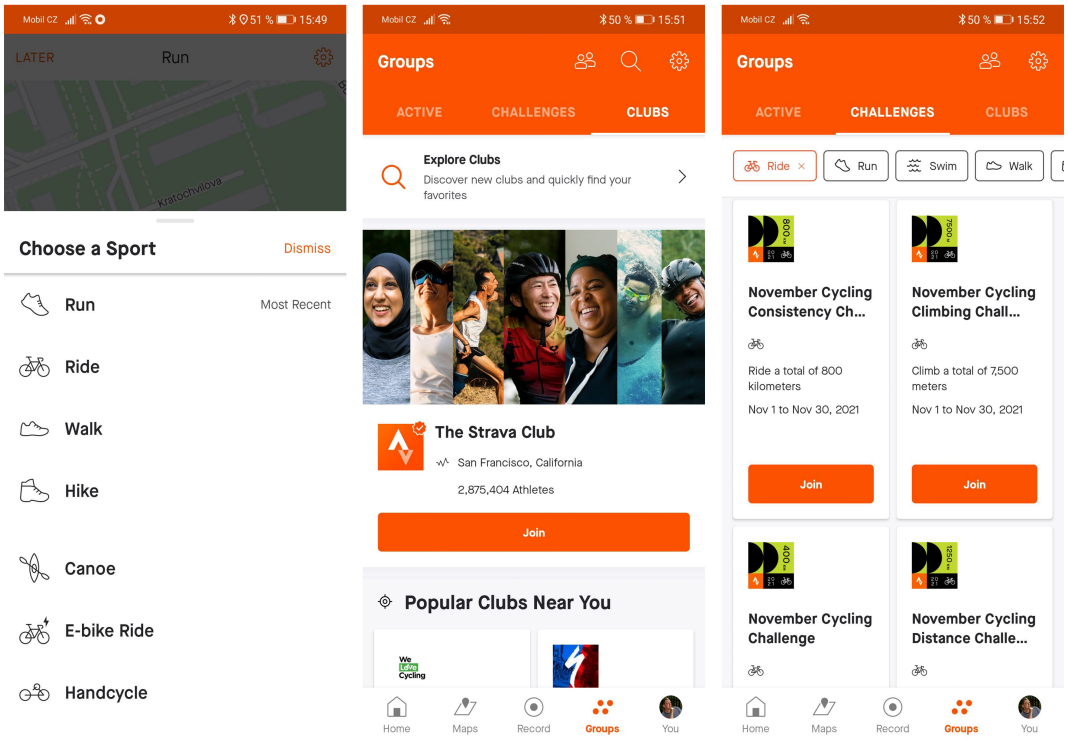
Fitify je jedna z nejvíce používaných tréninkových aplikací na trhu. Fitify je ovšem založena na docela jiné myšlence. Fitify nabízí tréninky a tréninkové plány na míru nebo rovnou předpřipravené univerzální tréninkové plány. Naprostá většina těchto služeb je placená formou předplatného, takže aplikace zde má vlastně suplovat osobního trenéra. Pro porovnání s naší prací, Fitify nabízí alespoň sestavování vlastních tréninků, které ale skládáte z databáze předpřipravených cviků. Sestavování tréninků, až na jeden zkušební trénink, je znovu placená funkce. Fitify má velkou sadu již hotových tréninkových aktivit a ty se snaží prodat, kdežto naše práce má za cíl nechat tuto databázi růst komunitně (viz Záměr aplikace). Nakonec Fitify se striktně zaobírá fitness aktivitami – cardio tréninky, posilování, hubnutí apod., zatímco naše práce by měla umožnit libovolnou sportovní aktivitu.



Obrázek 1.2: Screenshoty z aplikace Fitity

1.5.2 Strava

Aplikace Strava se stejně jako Fitify řadí mezi jednu z hojně používaných aplikací na trhu. Zatímco aplikace Fitify se zaměřovala na cvičení stejně jako naše aplikace, Strava je spíše něco na pomezí sociální sítě a aplikací jako např. Fitify. Jedna ze základních funkcí Stravy je sběr dat o sportovních aktivitách uživatele. Například důležitá funkcionalita je sledování polohy, měření rychlosti a nastoupaných metrů, pomocí GPS modulu v telefonu, během běhu nebo jízdě na kole. Strava pak shromažďuje výsledky, které tam uživatel zadá ručně nebo je pomocí telefonu, během výkonu sama naměří. Strava dále pomáhá lidem se společným sportovním zájmem shlukovat do skupin a mezi sebou navzájem sdílet svoje úspěchy a výkony. V těchto skupinách i mimo ně nabízí koncept výzev v daném sportovním odvětví. Výzva nabádá uživatele k dosažení jistého výkonu v daném sportu, může vypadat například následovně: „Uběhni za měsíc 150 km“. Svoje výsledky pak uživatel porovnává s dalšími uživateli. V placené verzi, formou předplatného, pak Strava nabízí tréninkové plány, které jsou předpřipravené a zaměřené vždy na konkrétní cíl, například: „uběhni maraton“. Strava tedy funguje spíše jako databáze vašich výsledků a sociální síť pro sdílení výsledků s dalšími uživateli. V placené verzi pak zaměřené tréninky, pouze pro daného uživatele.

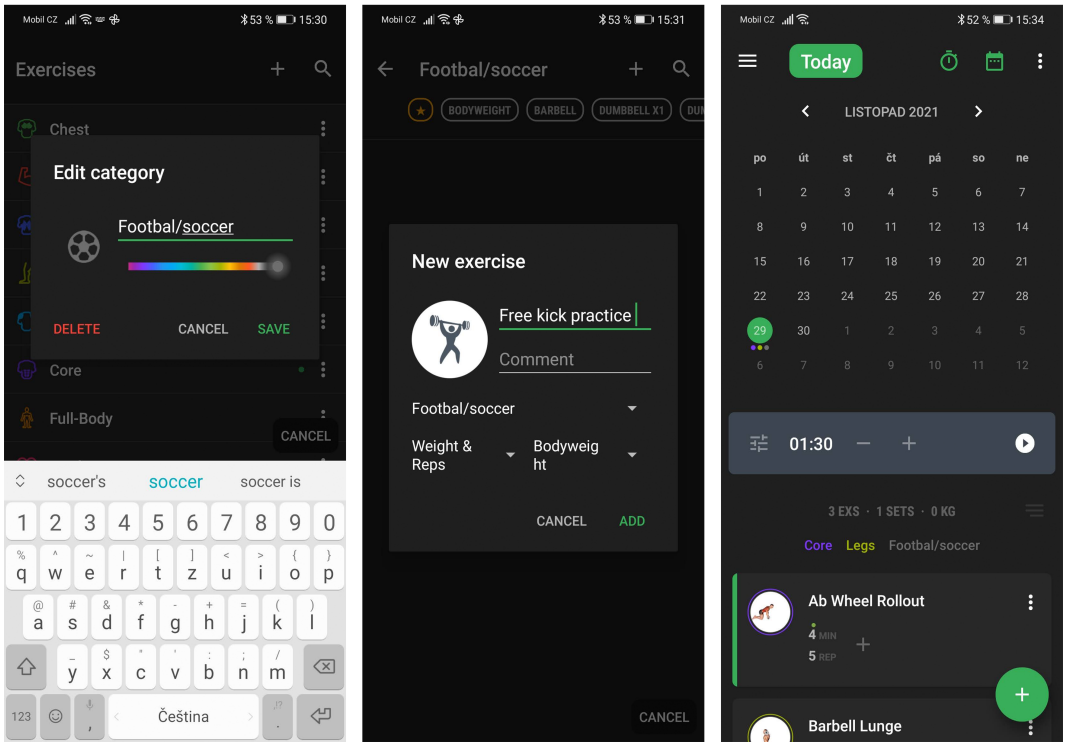


Obrázek 1.3: Screenshoty z aplikace Strava

1.5.3 GymKeeper

Aplikace GymKeeper je docela podobná nápadu naší práce. V aplikaci lze vytvářet cviky a vkládat je do kategorií, které uživatel také může vytvářet sám. Tyto kategorie mají podobný význam, jako tagy z naší práce. Nicméně naše práce si klade za cíl přiřadit libovolný počet vlastností každému cviku, zatímco GymKeeper vkládá cviky do jedné kategorie. Můžeme se ti představit jako vkládání do šuplíku. Pokud danou věc nemáme víckrát nemůže být ve více šuplících najednou. Zatímco naše aplikace by na věci měla věšet visačky s danou vlastností, těch lze pověsit libovolně mnoho. Narozdíl od naší práce má aplikace GymKeeper rozsáhlou databázi již vytvořených cvičení a kategorií Znovu se ale jedná o tréninkové aktivity v oblasti fitness. Bohužel aplikace sice nabízí volnou ruku ve vytváření cviků a dalších v aktivit, ale je připravena spíše na trénování fitness, posilování, apod. Při vytváření cviku tedy nabízí parametry, které se hodí k posilování, ale už ne tolik například na trénování fotbalu (viz obrázek). Dále GymKeeper funguje jako jeden velký kalendář, kam si vkládáte cviky. K nim si pak uživatel může přidat statistiku, jak dlouho mu daný cvik v daný den trval a jak výkonný byl. Do statistik si uživatel může například uložit jakou váhu při posilování zvedl.

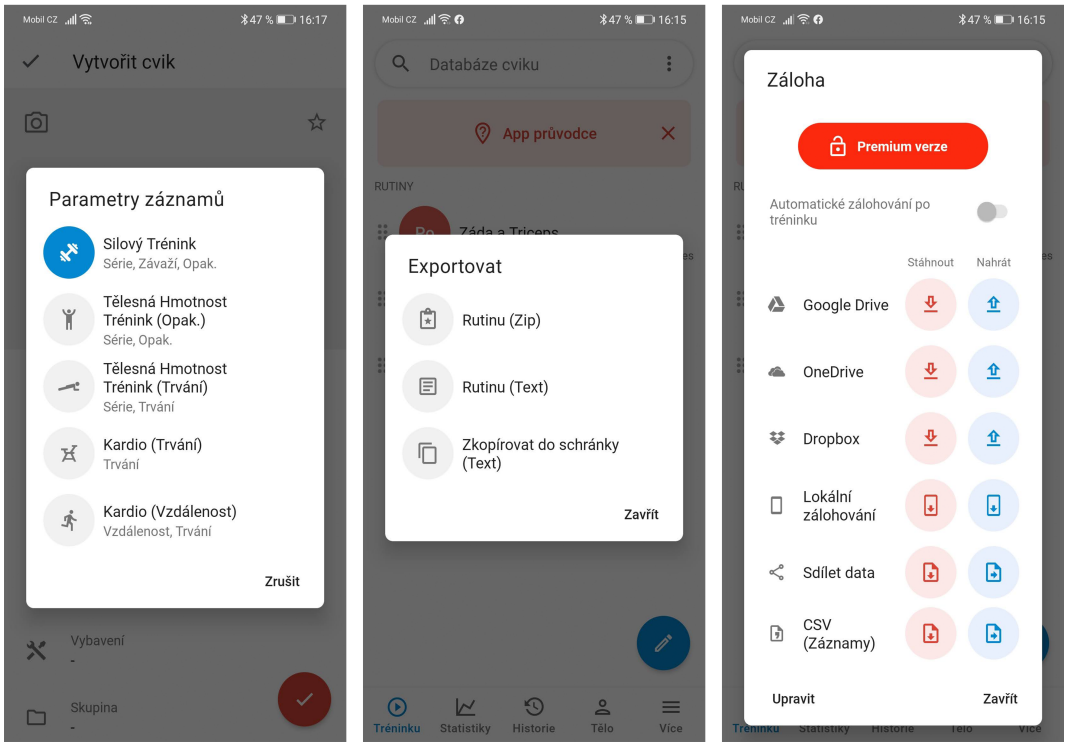
Narozdíl od naší práce, zde absolutně chybí možnost své tréninky poslat jiným uživatelům. Aplikace GymKeeper totiž slouží spíše jako osobní kalendář vašeho cvičení. Aplikace GymKeeper je znovu placená, tentokrát se neplatí předplatné, ale platí se jednorázově za software.



Obrázek 1.4: Screenshoty z aplikace GymKeeper

1.5.4 GymRun

Aplikace GymRun nabízí plánování rutin na různé dny. Do rutiny uživatel může skládat cviky, tudíž rutina zde odpovídá jednomu tréninku v naší práci. Dále tu máte dvě možnosti ekvivalentu tréninkového plánu z naší práce. První možnosti je přiřazování rutin (tréninků) na jednotlivé dny v týdnu, tedy to je jako jeden týdenní tréninkový plán. Druhou možností je plánování do normálního kalendáře, stejně jako aplikace GymKeeper (viz sekce GymKeeper). Dále GymRun obsahuje, stejně jako GymKeeper , rozsáhlou databázi již vytvořených cviků a tréninků a zároveň umožňuje uživateli vytvořit vlastní cvik i trénink. Tréninky lze zařazovat do skupin, podobně jako v aplikaci GymKeeper. Tedy znovu se zde opakuje koncept zařazování cviků do skupin, narozdíl od konceptu naší práce, kde cvikům přiřazujeme vlastnosti. Stejně jako GymKeeper nebo Fitify je aplikace orientována spíše na fitness a tomu odpovídá i seznam předpřipravených atributů které lze k cvikům a tréninkům dát. GymRun obsahuje i statistiku proběhlých tréninků nebo prohlížení aktuálně cvičeného tréninku/cviku – podobně jako tomu chceme i v naší práci. V aplikaci je možné exportovat rutiny (tréninky) nebo je nějak sdílet pomocí zálohy (viz obrázek), tato možnost ale nikde není blíže popsaná a vyzkoušet ji lze až po zaplacení předplatného, na kterém je tato aplikace založená. Je zde možné sdílení pomocí exportu a posílání přes jiný komunikační kanál, např. e-mail, chatovací aplikace apod. Aplikace GymRun se asi nejvíce blíží nápadu naší práce, nicméně je těžce orientovaná na fitness, cviky zařazuje do skupin a ne naopak jak tomu chceme my a nemá pohodlný a praktický systém na sdílení tréninkových aktivit s dalšími uživateli.



Obrázek 1.5: Screenshoty z aplikace gymRun

1.5.5 Shrnutí

Většina aplikací, které jsme v rámci práce vyzkoušeli, byla orientovaná dost konkrétně na posilování, fitness, běh a další podobné individuální aktivity. Záměrem naší aplikace je podporovat organizaci tréninkových aktivit bez ohledu na sport. Stejně dobře jako fitness chceme podporovat i týmové sporty např. florbal nebo fotbal (viz podkapitola 1.2). Přestože existují aplikace, ve kterých by nějak bylo možné zařídit podporu i pro jiné sporty, ačkoliv tak nejsou prvotně myšlené, např. GymRun, v žádné z nich se nám nepodařilo najít funkcionalitu, kterou zamýšlíme. Žádná z aplikací, které jsme si vyzkoušeli, nenabízela systém tagů. Nejpodobnější bylo zařazování cviků do skupin. Dále žádná z těchto aplikací nenabízela přímý chat s dalšími uživateli, který by umožňoval přímo posílat tréninkové aktivity.

Na druhou stranu všechny aplikace mají společné to, že je používají velké počty uživatelů. Pomineme-li funkce aplikací, domníváme se, že v dnešní době je část úspěchu libovolné aplikace založena na jejím grafickém uživatelském rozhraní. Chtěli bychom se tedy po vzoru výše uvedených aplikací zaměřit na praktické a uživatelsky přívětivé ovládací prvky. Mezi tyto prvky může například patřit navigační panel, pomocí kterého se uživatel naviguje skrze jednotlivé funkce či obrazovky aplikace. Příklad takového panelu můžeme vidět na prostředním a pravém screenshotu z aplikace Strava (viz Obrázek 1.3). Navigační panel je zde umístěn pevně na horním panelu, nicméně navigační panel můžeme mít i na spodní části obrazovky nebo například jako schované postranní menu. Dalším výrazným prvkem jsou ikonky doplňující text, které pomáhají rychlé orientaci v aplikaci. Můžeme si takových ikonek všimnout u všech aplikací, které jsme pro-

hlíželi. Třetím hojně používaným prvkem jsou vyskakovací dialogy[1] (viz. všechny tři screenshoty na obrázku 1.5). Tyto vyskakovací dialogy se často používají pro detailní pohled nebo detailní úpravy nějaké věci. Tyto tři ovládací prvky jsou dnes běžnou součástí většiny aplikací, můžeme tedy říct, že jsou již praxí prověřené. V naší aplikaci, tam kde nám to bude dávat smysl, budeme tyto ovládací prvky také používat.

1.6 Funkční požadavky (Seznam funkcí)

V předchozích částech jsme si definovali čeho se naše aplikace bude týkat, jaké jsou typické scénáře použití naší aplikace a jak by mohli vypadat základní funkce aplikace z pohledu uživatele. Nyní bychom chtěli na základě těchto informací vytvořit strukturovaný seznam funkcí, které aplikace musí mít, aby odpovídala našim představám. Tyto funkce budou přímo vycházet z podkapitol 1.2, 1.3, 1.4. Jen tedy všechny funkce, které jsme si v daných podkapitolách řekli, že by aplikace měla mít, shrneme do seznamu. Abychom se mohli později v práci na tyto definice jednoduše odkazovat, uděláme seznam číslovaný.

1. Uživatel musí být schopný vytvořit v databázi nový cvik.
   1.1. Součástí cviku je volitelný popisek.
2. Uživatel musí být schopný libovolný, již vytvořený, cvik editovat.
3. Uživatel musí být schopný smazat libovolný cvik.
   3.1. Uživatel by při mazání cviku měl být upozorněn, že odstranění cviku má za následek odebrání daného cviku ze všech tréninků.
4. Uživatel musí být schopný libovolnému cviku v databázi přiřadit tag z databáze uživatelových tagů.
5. Uživatel musí být schopný libovolnému cviku v databázi přiřadit nový tag.
   5.1. Přiřazení nového tagu musí uložit daný nový tag do databáze uživatelových tagů.
6. Uživatel musí být schopný zobrazit seznam všech cviků v databázi.
7. Uživatel musí být schopný třídit seznam cviků podle tagů.
   7.1. Uživatel musí být schopný zadat množinu tagů, které musí **každý** výsledný cvik obsahovat.
   7.2. Uživatel musí být schopný zadat množinu tagů, kde každý výsledný cvik, musí obsahovat **alespoň jeden** tag ze zadané množiny.
8. Uživatel musí být schopný vytvořit v databázi nový trénink.
   8.1. Součástí tréninku je volitelný popisek.

[1] vyskakovacím dialogem se myslí okno, které je menší než obrazovka a překreslí se přes momentální obrazovku aplikace, uživatel tedy neztratí kontakt s původní obrazovkou.

9. Uživatel musí být schopný libovolný, již vytvořený, trénink editovat.
    9.1. Uživatel musí být schopný editovat jméno a popisek tréninku.
    9.2. Uživatel musí být schopný přidat do tréninku libovolný, i duplicitní, cvik.
    9.3. Uživatel musí být schopný editovat množství a pořadí cviků v daném tréninku.
10. Uživatel musí být schopný smazat libovolný trénink.
    10.1. Uživatel by při mazání tréninku měl být upozorněn, že odstranění tréninku má za následek odebrání daného tréninku ze všech tréninkových plánů.
11. Uživatel musí být schopný zobrazit seznam všech tréninků v databázi.
12. Uživatel musí být schopný třídit seznam tréninků podle tagů.
    12.1. Uživatel musí být schopný zadat množinu tagů, kde každý výsledný trénink obsahuje alespoň jeden cvik, který musí mít přiřazen **každý** tag ze zadané množiny.
    12.2. Uživatel musí být schopný zadat množinu tagů, kde každý výsledný trénink obsahuje alespoň jeden cvik, který musí mít **alespoň jeden** tag ze zadané množiny.
13. Uživatel musí být schopný vytvořit v databázi nový tréninkový plán.
14. Uživatel musí být schopný libovolný, již vytvořený, tréninkový plán editovat.
    14.1. Uživatel musí být schopný editovat jméno tréninkového plánu.
    14.2. Uživatel musí být schopný přidat do plánu libovolný počet dní.
    14.3. Uživatel musí být schopný editovat pořadí dní v daném plánu.
    14.4. Uživatel musí být schopný smazat libovolný den v plánu.
    14.5. Uživatel musí být schopný editovat konkrétní den v rámci tréninkového plánu.
        14.5.1. Uživatel musí být schopný přidat do plánu libovolný počet tréninků.
        14.5.2. Uživatel musí být schopný odstranit trénink ze dne.
        14.5.3. Uživatel musí být schopný měnit pořadí tréninků v rámci dne.
15. Uživatel musí být schopný smazat libovolný tréninkový plán.
16. Uživatel musí být schopný zobrazit seznam všech tréninkových plánů v databázi.
17. Uživatel musí být schopný třídit seznam tréninkových plánů podle tagů.
    17.1. Uživatel musí být schopný zadat množinu tagů, kde každý výsledný tréninkový plán tranzitivně obsahuje (obsahuje trénink, který obsahuje cvik) alespoň jeden cvik, který musí mít přiřazen **každý** tag ze zadané množiny.

17.2. Uživatel musí být schopný zadat množinu tagů, kde každý výsledný tréninkový plán tranzitivně obsahuje **alespoň jeden** cvik, který musí mít alespoň jeden tag ze zadané množiny.

18. Uživatel musí být schopný spustit trénink.

    18.1. Aplikace musí v rámci spuštění tréninku ukazovat cvik, které je v tréninku na řadě se všemi detaily cviku.

    18.2. Uživatel musí být schopný, při spuštěném tréninku a zobrazeném aktuálním cviku, označit cvik jako hotový.

    18.3. Uživatel musí být schopný, v rámci spuštěného tréninku, prohlížet i cviky, které následují.

19. Uživatel musí být schopný smazat libovolný tag ze své databáze.

    19.1. Uživatel by při mazání tagu měl být upozorněn, že odstranění tagu má za následek odebrání daného tagu všem cvičením.

20. Aplikace musí být schopna splnit požadavky 1–19, když uživatel není přihlášený.

21. Uživatel musí být schopný vytvořit nový účet, tzn. registrovat se v aplikaci, pokud právě není přihlášený.

22. Uživatel musí být schopný přihlásit se do aplikace existujícím účtem, pokud není žádný uživatel přihlášený.

23. Přihlášený uživatel musí být schopný se z aplikace odhlásit.

24. Aplikace musí být schopna splnit požadavky 1–19, když je uživatel přihlášený.

25. Data v databázi přihlášeného uživatele musí být po přihlášení dostupná z kteréhokoliv zařízení

26. Aplikace musí po registraci nabídnout uživateli import dat z lokální databáze do online databáze.

27. Přihlášený uživatel musí být schopný importovat data z lokální databáze do online databáze.

28. Přihlášený uživatel musí být schopný najít libovolného jiného registrovaného uživatele.

29. Přihlášený uživatel musí být schopný vést chat s libovolným jiným registrovaným uživatelem.

30. Přihlášený uživatel musí být schopný poslat v chatu s jiným uživatelem libovolný cvik.

    30.1. Aplikace musí cvik poslat včetně všech tagů daného cviku.

31. Přihlášený uživatel musí být schopný poslat v chatu s jiným uživatelem libovolný trénink.

31.1. Aplikace musí trénink poslat včetně všech cviků obsažených v daném tréninku (viz požadavek 30).

31.2. Aplikace musí poslat cviky z daného tréninku bez duplikací.

32. Přihlášený uživatel musí být schopný poslat v chatu s jiným uživatelem libovolný tréninkový plán.

32.1. Aplikace musí tréninkový plán poslat včetně všech tréninků obsažených v daném tréninkovém plánu (viz požadavek 31).

32.2. Aplikace musí poslat tréninky z daného tréninkového plánu bez duplikací.

33. Aplikace musí vizuálně odlišit textovou zprávu a zprávu každé z tréninkových aktivit.

34. Přihlášený uživatel musí být schopný v chatu otevřít náhled poslané tréninkové aktivity.

35. Přihlášený uživatel musí být schopný uložit zaslaný cvik z chatu k sobě do databáze.

35.1. Aplikace musí cvik uložit včetně všech tagů daného cviku.

35.2. Aplikace musí použít tag z uživatelovy databáze, pokud se shoduje ve jménu s tagem přiřazeným ke cviku.

36. Přihlášený uživatel musí být schopný uložit zaslaný trénink z chatu k sobě do databáze.

36.1. Aplikace musí trénink uložit včetně všech cviků obsažených v daném tréninku (viz požadavek 35).

36.2. Aplikace musí uložit cviky z daného tréninku bez duplikací.

37. Přihlášený uživatel musí být schopný uložit zaslaný tréninkový plán z chatu k sobě do databáze.

37.1. Aplikace musí tréninkový plán uložit včetně všech tréninků obsažených v daném tréninkovém plánu (viz požadavek 36).

37.2. Aplikace musí uložit tréninky z daného tréninkového plánu bez duplikací.

38. Přihlášený uživatel musí být schopný v chatu poslat textovou zprávu.

39. Aplikace musí zprávy v chatu mít seřazené podle času odeslání.

40. Přihlášený i ne-přihlášený uživatel musí být schopen spustit tréninkový plán.

40.1. Aplikace si musí pamatovat všechny spuštěné plány.

40.1.1. Pokud je uživatel přihlášený, musí spuštěné plány být dostupné z libovolného zařízení.

40.1.2. Pokud není uživatel přihlášený, daný spuštěný plán musí být dostupný jen na daném zařízení.

40.2. Aplikace musí v libovolný reálný den ukázat, jaký den ve spuštěném tréninkovém plánu je na řadě.

1.7 Shrnutí cílů práce

Na závěr úvodní kapitoly si shrneme cíle naší bakalářské práce. Cíle jsou následující:

1. Vytvořit mobilní aplikaci pro systém Android.

   1.1. Naše aplikace má podporovat minimálně verzi 8.1.

2. Naše aplikace má alespoň prozatím používat technologie, které jsou zdarma.

3. Naše aplikace má být uživatelsky přívětivá a má kopírovat trendy uživatelského rozhraní dnešních úspěšných aplikacích.

4. Aplikace má splnit seznam funkčních požadavků (viz podkapitola 1.6).

2. Analýza, rozbor aplikace

Nyní již máme aplikaci popsanou z hlediska uživatele, máme seznam funkčních požadavků a chceme najít nejlepší řešení pro každý problém, který nám vyvstane. Budeme se tedy snažit analyzovat různá řešení, které by naplnili funkční požadavky (viz podkapitola 1.6).

2.1 Framework

První problém, který je třeba vyřešit, je prostředí, jazyk a framework, ve kterém budeme aplikaci vyvíjet. V předchozí kapitole jsme si stanovili, že aplikace má být pro Android, uděláme si tedy výčet základních možností, jak tohoto cíle dosáhnout.

1. Pro nativní vývoj pro Android se používají jazyky Java a Kotlin. Java se používá, protože Android je postavený na JDK (Java Development Kit) a Kotlin se používá díky tomu, že běží v Java Virtual Machine. Jako vývojové prostředí bychom pak používali Android Studio nebo IntelliJ IDEA.

2. Druhou možností by mohl být framework Flutter, který umožňuje multiplatformní vývoj, tedy pro Android i pro iOS najednou. Flutter je technologie od Googlu, ve které se používá programovací jazyk Dart. Aplikace se ve Flutteru nakonec zkompiluje do nativního kódu pro Android respektive iOS. Znovu bychom pravděpodobně použili vývojové prostředí Android Studio nebo IntelliJ IDEA.

3. Podobná možnost jako Flutter je i Xamarin Forms. Xamarin Forms je multiplatformní vývojový framework od společnosti Xamarin. Chová se stejně jako Flutter, tedy umožňuje vývojářům vyvíjet nativní aplikace pro jednotlivé platformy (Android, iOS, aj.) a používat při tom jeden programovací jazyk, zde konkrétně C#, a jedno vývojové prostředí, zde Visual Studio.

4. Xamarin nabízí možnost nativního vývoj i pouze pro Android – Xamarin Android. Ten se téměř neliší od nativního vývoje v Javě. Jediným rozdílem je práce v programovací jazyce C#. Narozdíl od vývoje v Xamarin Forms zde odpadá potřeba psát separátně kód společný pro všechny platformy a pak kód specifický pro danou platformu.

Autor této aplikace se dnes zaměřuje především na programovací jazyk C# a rodinu technologií .NET. Tedy nebudeme dále uvažovat vývoj nativní aplikace v Jave nebo Kotlinu, ani multiplatformní vývoj ve Flutteru. Dále autor této aplikace má z předchozích let drobné zkušenosti s vývojem nativní aplikace pro Android a zároveň nám cíle práce stanoví, že chceme aplikaci na Android. Vezmeme-li v potaz tyto aspekty, vychází nám ze zbylých jako nejvhodnější platforma Xamarin Android. Nakonec i kdybychom chtěli aplikaci rozšířit na iOS, stačilo by migrovat projekt na Xamarin Forms a použít velkou část kódu z původního projektu.

2.2 Správa uživatelů

Pro naši aplikaci bude třeba se zamyslet, jakým způsobem budeme zavádět správu uživatelů. Možnost registrace a přihlášení se je totiž součástí funkčních požadavků (viz 1.6, požadavek č. 21, 22). Zároveň jsme si z pohledu uživatele řekli (viz sekce 1.4.3), že by dávalo smysl, pokud by se uživatel mohl do aplikace registrovat a přihlásit pomocí e-mailu nebo pomocí jiné služby jako např. Google nebo Facebook. Nyní z našeho hlediska dávají smysl obě možnosti, ale pro jednoduchost nám stačí zatím jen přihlášení pomocí e-mailu. Nicméně možnost přihlášení pomocí jiných služeb nechceme zavrhnout úplně, rádi bychom tuto možnost v budoucnu měli.

Tradiční způsob návrhu v oblasti webových nebo online mobilních aplikací odpovídá třívrstvé architektuře, kde uživatel pracuje s první vrstvou, tedy s uživatelskou aplikací u sebe na zařízení. Tato aplikace dále komunikuje s druhou vrstvou, tedy se serverovou aplikací, která pak dále komunikuje, respektive obsluhuje třetí vrstvu – databázi. Tímto způsobem se tradičně navrhuje i správa uživatelů. Tedy že součástí serverové aplikace a databáze je i správa uživatelů. V aplikacích, kde je potřeba vytvořit správa uživatelů tímto způsobem, to s sebou přináší více pracnosti. Nejenže je to kus softwaru, který musí někdo navrhnout, implementovat, adt., ale je zde vždy třeba i dbát bezpečnosti, například co se ukládání hesel týče.

V dnešní době se na trh dostává v této oblasti nového přístupu, kdy součástí online databázových služeb je zároveň služba pro správu uživatelů. Taková služba může být například knihovna pro obsluhu databáze nebo klidně služba, která nabízí vše od ukládání dat o uživateli, přes bezpečnost, až po komunikaci s dalšími službami a propojováním účtů u jiných společností či aplikací. Pokud bychom pro naši aplikaci našli takovou technologii a službu, která nám pomůže se správou uživatelů, jistě by nám to výrazně ulehčilo práci. Musíme ale stále brát v potaz otázku peněz, protože chceme aby naše aplikace používala (alespoň zatím) prostředky zdarma (viz. podkapitola 1.7).

2.3 Výběr online databáze

Jak jsme si stanovili v úvodní kapitole, naše aplikace má mít offline databázi pro nepřihlášené uživatele a zároveň online databázi pro ty přihlášené (viz podkapitola 1.6 – požadavek 20, 24, 25). Navíc potřebujeme správu uživatelů (viz podkapitola 2.2). Zaměříme se nejprve na online databázi, jakožto složitější část. Chceme tedy pro online databázi najít vhodnou službu. Při výběru vhodné technologie a služby pro databáze je nutno vzít v potaz především strukturu dat, které budeme používat, správu uživatelů v naší aplikaci a v neposlední řadě cenu, která pro nás může nakonec být určující. V rámci těchto požadavků nám vyšly dvě velké služby – Google Firebase a Azure SQL Database. Obě tyto služby poskytují online databázi pro mobilní aplikace, je možné s nimi pracovat v Xamarin Android a poskytují podporu správy uživatelů.

2.3.1 Azure SQL Database

Platforma Azure je od společnosti Microsoft. V rámci této platformy existuje i služba SQL databáze a provoz serverových aplikací například právě pro obsluhu databází. Kombinaci SQL databáze a serverové aplikace bychom zde potřebovali použít. S Azure SQL Database bychom museli implementovat klasický třívrstvý model, jaký jsme zmínili v podkapitole 2.2. Pracovali bychom zde s SQL databázi ve spojení s Entitiy Frameworkem, který by nám poskytl nadstavbu pro klasický SQL[1]. Pomocí Entity Frameworku bychom se vyhnuli například definici tabulek do databáze nebo bychom se mohli vyhnout klasickému dotazování pomocí SQL. Navíc bychom s jeho pomocí mohli najednou implementovat online i offline databázi, poněvadž bychom mohli použít modely používané Entity Frameworkem pro obě databáze. Toto by nám dále dalo ještě jednu výhodu ve chvíli, kdy budeme chtít implementovat import dat z offline databáze do online databáze (viz podkapitola1.6 – požadavky 26, 27), protože by formát dat byl úplně identický. Nakonec Azure nám nabízí i správu uživatelů ve formě služby Azure Active Directory.

Nicméně Azure SQL Database má pro nás i mnoho nevýhod. První je, že bychom kromě samotné Azure SQL Database potřebovali právě i serverovou aplikaci. To by se ovšem dalo čekat i u jiných technologií. Další je cena produktů Azure. Azure nenabízí svoje služby zdarma a pokud už ano, je to jen velmi omezená služba na vyzkoušení dané technologie. To platí i pro službu správy uživatelů – Azure Active Directory – která je též placená nebo se poskytuje v rámci jiných placených služeb. V úvodní kapitole jsme si stanovili, že aplikace není komerčně orientovaná, tedy potřebujeme technologie, které jsou zdarma. Poslední nevýhodou, kterou jsme vzali v potaz, je složitost Entity Frameworku ve spolupráci s Azure. Autor této aplikace si práci s Entity Frameworkem pro účely vytvoření této aplikace vyzkoušel a vyvodil, že plné pochopení tohoto frameworku by vyžadovalo nemalé úsilí.

2.3.2 Firebase

Na druhé straně zde máme Google Firebase (dále jen Firebase). Firebase poskytuje taky placené služby, nicméně má štědrou nabídku služeb zdarma (viz Firebase – pricing [4]). Dále Firebase poskytuje správu uživatelů, kde zároveň dává na výběr přihlašování skrze jiné služby – Google, Facebook, Twitter aj. – a také nabízí přihlašování pomocí e-mailu. V rámci správy uživatelů Firebase poskytuje i zabezpečení databáze pomocí jednoduchých pravidel psaných přímo ve webovém uživatelském rozhraní. Tyto pravidla umožňují například některá data jen číst nebo umožňují upravovat data jen pokud je uživatel přihlášený atd. (viz Firebase documentation [5]). Tato správa uživatelů je zdarma ve velkém rozsahu, rozhodně postačujícímu pro zdarma poskytovanou databázi. Další výhodou Firebase je absence serverové aplikace, která by databázi ovládala. Respektive tato serverová aplikace je už v technologii integrovaná a vývojáři se jí nemusí zabývat. Nakonec pro užívání Firebase v aplikaci, je potřeba pouze konfigurační soubor pro danou aplikaci a knihovna pro ovládání databáze.

V rámci Firebase jsou k dispozici dva další typy databáze. Google zde rozlišuje

[1]SQL je zkratka (anglicky Structured Query Language) pro standardizovaný strukturovaný dotazovací jazyk pro práci s daty v relačních databázích [3].

Firestore a Realtime Database. Realtime Database je strukturovaná jako JSON strom, zatímco Firestore je tvořena z dokumentů a kolekcí, kde dokumenty jsou v kolekci, ale součástí každého dokumentu mohou být i další kolekce. V rámci služeb zdarma oba typy databáze nabízí 1GB dat a správu uživatel mají společnou. Struktura Firestore přibližně odpovídá modelu dat, jaký jsme si nastínili v podkapitole Rozbor scénářů (viz sekce 1.4.1). Měli bychom totiž kolekci cviků, tréninků a plánů. V kolekci cviků bychom měli pro každý cvik jeden dokument s podrobnostmi o cviku, zatímco například v kolekci plánů bychom měli pro každý plán jeden dokument, ten by dále obsahoval kolekci dnů a každý den by pak obsahoval kolekci tréninků. V databázi Firestore by tedy data například dvou libovolných plánů byla striktně oddělena dokumentem definující daný plán. Kdybychom si tedy vybrali Firebase pro online databázi, používali bychom Firestore.

Firestore i Realtime Database navíc poskytují podporu offline cachování dat. Zařízení, na kterém běží aplikace tedy nemusí být připojené k internetu, ale stále může pracovat s online databází. Změny v databázi se v zařízení cachují a po připojení k internetu se synchronizují s online databází. Tato možnost je pro oba typy databáze volitelná, ale ve výchozím nastavení je povolená.

2.3.3 Výběr online databáze – závěr

Vzhledem ke složitosti Entity Frameworku a především pak ceně služeb Azure, jsme se Azure služby rozhodli nepoužít. Nicméně pokud by aplikace byla od počátku výdělečná a zároveň pokud bychom dobře znali Entity Framework, byla by to pravděpodobně jasná volba. Azure totiž poskytuje velkou škálovatelnost pro jejich řešení, tedy by se například velmi snadno dali přidávat prostředky s rostoucím počtem uživatelů. Dále Azure i Xamarin jsou od společnosti Microsoft a tedy je zde potenciál pro zlepšování jejich kooperace, ale i kooperace s dalšími technologiemi a službami od Microsoftu. Vzhledem k jmenovaným výhodám Firebase, jsme se rozhodli použít tuto službu. Pro začátek poskytuje pro naši aplikaci dostatek úložného prostoru a dostačující správu uživatelů zdarma. Jako velmi příjemný bonus shledáváme i offline synchronizovanou databázi. Zároveň nám použitím Firebase odpadá povinnost vytvářet serverovou aplikaci spravující databázi.

2.4 Výběr offline databáze

Pro offline databázi jsme pak chtěli vybrat technologii co nejjednodušší na používání. Jako první možnost se nabízelo použít Firebase, abychom měli stejnou technologii jako online databáze. Firebase bohužel zatím nenabízí žádnou offline alternativu jejich databází. V rámci Firebase se ještě nabízelo použít vlastnosti Firestoru – offline synchronizovanou databázi. Tato vlastnost ale není myšlena jako samotná offline databáze. Z cache určené pro cachování dat offline se po čase mohou data mazat a zároveň se nikde nepíše, jaký by toto mohlo mít vliv na výkonost. Firebase tedy použít nelze.

Online databázi jsme vybírali především na základě nabídky služeb. Ačkoliv jsme vybrali ne-relační databázi, našemu datovému modelu by relační databáze vyhovovala. Máme totiž jednoduché struktury, mezi kterými pak ukládáme

vztahy. Každou entitu, ze které se nějaký model skládá, můžeme ukládat v samostatné tabulce a napříč tabulkami se odkazovat pomocí identifikátorů. Protože vyvíjíme mobilní aplikaci, chceme pro offline databázi použít jednoduchý systém relační databáze. Chceme se vyhnout použití těžkotonážních databázových systémů, kde například databáze musí běžet jako separátní proces. Rádi bychom měli databázi ve formě knihovny s jednoduchým rozhraním. Toto nám poskytuje SQLite databáze (viz Wikipedia – SQLite [6]), která je distribuovaná jako knihovna, databáze se ukládá do jednoho souboru a hlavně běží v procesu aplikace a nepotřebuje vlastní proces. Pro mobilní aplikace na Android jde o běžnou volbu, protože SQLite je již v systému Android zabudovaná.

2.4.1 Použiti SQLite v naší aplikaci

SQLite jakožto relační databáze, je založena na tabulkách, kde se ukládají záznamy stejného typu. Jeden řádek odpovídá jednomu záznamu a jeden sloupec odpovídá jedné informaci. Důležitým sloupcem (nebo více sloupci) je takový, který daný záznam jednoznačně určí – tedy identifikátor (klíč). Obecně se klíč v relačních databázích může skládat z vícero sloupců. Klíči jež se skládá z více sloupců se říká *složený klíč*.

Způsob ukládání v tabulkách by mohl na první pohled tvořit překážku při ukládání informací, které jsou v datovém modelu v seznamech. V naší aplikaci je takový příklad seznam tagů u cviku. Existuje ovšem standardní jednoduché řešení tohoto problému (viz normalizace databáze – Wikipedia [7]) – vytvoří se další tabulka, která neodpovídá žádné entitě z datového modelu, ale odpovídá nějakému vztahu. Na našem příkladě by vznikla tabulka říkající, že cvik má daný tag. Záznam v takové vztahové tabulce je jednoznačně určený identifikátory entit, kterých se tento vztah týká (v našem případě identifikátorem cviku a identifikátorem tagu). Právě z tohoto důvodu v relačních databázích existují složené klíče. Složené klíče umožňují nevkládat do tabulky uměle vytvořený klíč, ale lze se orientovat právě pomocí více hodnot, které by v tabulce stejně byly.

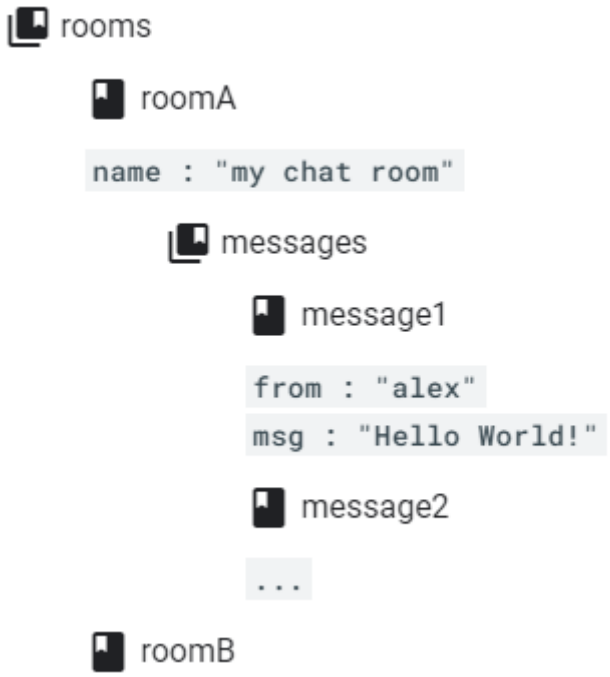
SQLite můžeme v naší aplikaci použít pomocí knihovny přímo od Androidu nebo můžeme použít knihovnu „sqlite-net-pcl“, která v .NET sloužící jako wrapper SQLite (viz adapter pattern – Wikipedia [8]). Knihovna „sqlite-net-pcl“ má oproti vestavěné knihovně od Androidu jednu výhodu a jednu nevýhodu. Knihovna „sqlite-net-pcl“ bohužel nepodporuje složené klíče, což by pro nás znamenalo, použít umělý generovaný klíč pro každou entitu, kde bychom normálně použili složený klíč. To není na závadu, jen to zvýší spotřebu perzistentního uložiště. Na druhé straně nám „sqlite-net-pcl“ umožňuje zjednodušené používání SQLite jen pomocí C# tříd, LINQu [9] a C# atributů [10]. Nepotřebujeme tedy vůbec používat SQL [3]. Vyhnuli bychom se psaní SQL dotazů, které by byly textové, tedy by v nich šlo udělat jednoduše chybu a místo toho by všechno zacházení z daty podléhalo typové bezpečnosti, jako zbytek kódu. Toto by nám zjednodušilo vývoj, zlepšilo by to následnou čitelnost kódu a udržitelnost aplikace. Z tohoto důvodu jsme si vybrali právě „sqlite-net-pcl“.

2.5 Fungování Firestore

Když máme vybrané databáze, chtěli bychom začít modelovat data a ukázat si jak se v námi zvolených databázích budou ukládat. Nicméně k tomu je třeba si ještě detailněji uvodit, jak funguje Firestore databáze, protože oproti ne-relačním databázím má neobvyklý způsob používání.

Jak již bylo řečeno (viz sekce 2.3.2), Firestore databáze organizuje data do kolekcí a dokumentů. V první úrovni jsou kolekce. Každá kolekce má své jméno a obsahuje dokumenty, které nemají jméno, ale unikátní identifikátor. Identifikátory dokumentů můžeme dodávat dokumentům sami, ale jednodušší je toto přenechat Firestoru, který to umí vyřešit za nás. V dokumentu lze uložit klasická data (řetězce, čísla, datum/čas, bool hodnotu), ale také seznam nebo mapu[2]. Všechna data uložená v dokumentu musí být označena klíčovým slovem popisující daný atribut – podobně jako ve formátu JSON. Dále pro daný dokument můžeme vytvořit další vnořené kolekce. Strukturu Firestoru si tedy můžeme představit jako větvící se strom, kde v každém prvním uzlu jsou kolekce a v každém druhém jsou dokumenty. Když budeme chtít k libovolným datům přistupovat, definujeme cestu pomocí názvů kolekcí, identifikátorů dokumentů a případně klíčovými slovy vedoucími ke konkrétním datům daného dokumentu.

Pro představu si můžeme strukturu Firestoru ukázat na příkladu na Obrázku 2.1. Na obrázku máme kolekci `rooms` obsahující dva dokumenty: `roomA` a `roomB`, v realitě bychom místo názvů pokojů měli dlouhé náhodné řetězce jako identifikátory. Dokument `roomA` dále obsahuje položku `name` a kolekci `messages`. V kolekci `messages` pak vidíme další dva dokumenty `message1` a `message2`, kde první jmenovaný obsahuje položky `from` a `msg`.



Obrázek 2.1: Ukázka struktury Firestore databáze, obrázek z dokumentace Firestoru [5]

[2]mapou se myslí seznam dvojic klíč : hodnota

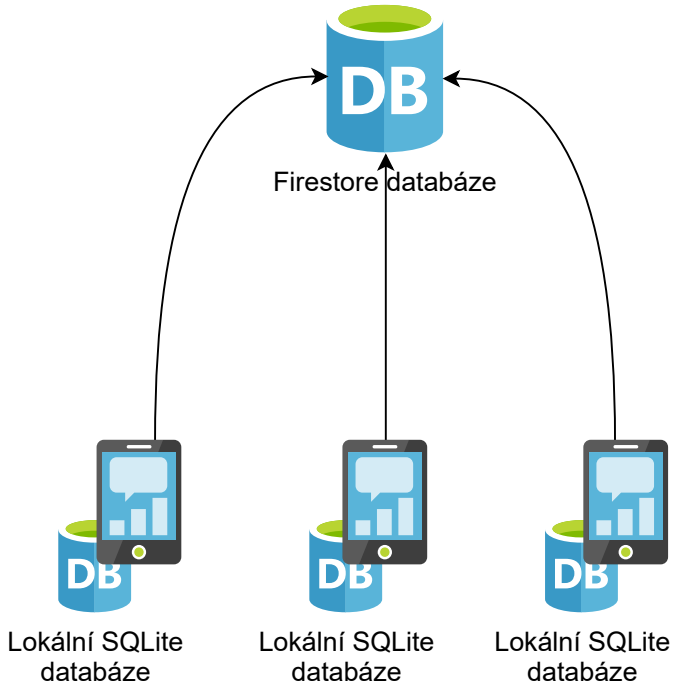
2.6 Modely dat

Protože naše aplikace svojí ideou staví především na modelu dat, pro vývoj aplikace bude klíčové si rozmyslet, jak bude model dat vypadat konkrétně. V návrhu budeme vycházet z podkapitol Funkční požadavky 1.6 a Rozbor scénářů 1.3, kde jsme již vytvořili model, se kterým bude v aplikaci pracovat uživatel. Vzhledem k tomu, že jsme si vybrali dvě odlišné technologie pro online a offline databázi, budeme mít i dva různé způsoby ukládání dat, tedy i modely dat pro databáze. Zároveň kdybychom v aplikaci pracovali s dvěma modely, přineslo by to další pracnost, protože bychom museli aplikaci přizpůsobit tak, aby byla schopna pracovat s oběma modely a zároveň bychom museli přepínat modely podle používané databáze. Dále by to přineslo problémy v udržitelnosti a rozšířitelnosti aplikace. Vytvoříme tedy jeden datový model, který se bude v aplikaci používat a databázím ho přizpůsobíme až při načítaní/ukládání dat. V této podkapitole si tedy rozebereme všechny potřebné detaily modelu, ty které jsou dané modelem z pohledu uživatele (viz sekce 1.4.1), ale i ty které nebudou uživateli přístupné, ale jsou pro fungování modelu nezbytné. Zároveň si rovnou ukážeme, jak budeme mít data uložená v databázích.

2.6.1 Identifikátory

Už víme jak fungují databáze, které jsme vybrali, ale ještě je třeba si vysvětlit detail týkající se všech entit. Když budeme data ukládat do databáze, ztrácíme výhodu odkazů (referencí) z běžícího programu. Kvůli hledání, editaci apod. nebo kvůli odkazování se záznamů mezi sebou, potřebujeme jednotlivé záznamy stále nějak identifikovat. Pro takové záznamy budeme tedy potřebovat unikátní identifikátor. U entit, na které se nebudeme potřebovat explicitně odkazovat, identifikátory nebudou potřeba.

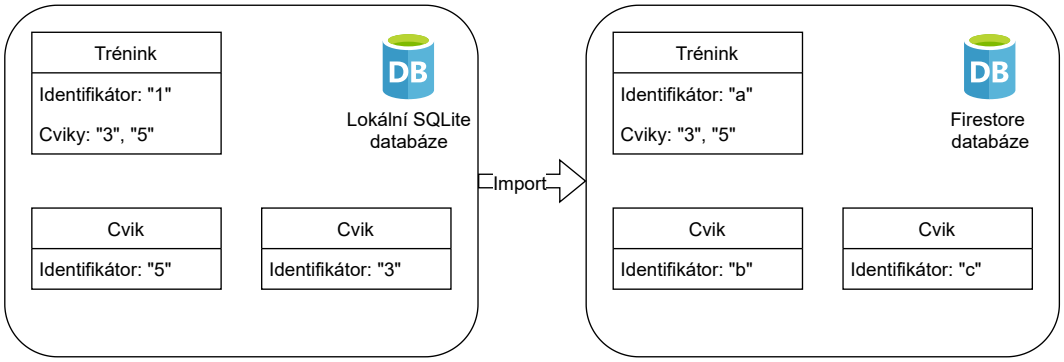
V aplikaci budeme používat kombinaci SQLite databáze a Firestore databáze (viz podkapitoly 2.3, 2.4), kde SQLite databáze obsahuje podporu pro práci s identifikátory a Firestore databáze dokonce vynucuje pro každý dokument identifikátor (identifikátor totiž definuje cestu k záznamu, viz sekce 2.3.2). Ve funkčních požadavcích si také definujeme, že aplikace musí umožnit import tréninkových aktivit z offline databáze do online databáze (viz podkapitola 1.6 – požadavky 26, 27). To znamená, že budeme potřebovat uložit záznam z SQLite databáze, který bude mít celočíselný identifikátor, do Firestore databáze, která má textové identifikátory. Mohli bychom jednoduše vzít danou aktivitu a vložit ji do Firestore databáze jak je, jen bychom převedli celočíselné identifikátory na textové. Nicméně protože SQLite je pro nás lokální databáze, nebere zřetel na jiné zařízení a tedy pokud by se ze dvou různých zařízení importovaly aktivity se stejným identifikátorem (viz Obrázek 2.2) a ve Firestore databázi by skončily ve stejné kolekci, nešlo by dané záznamy od sebe rozeznat a tedy i odkazy na ně by nebyly jednoznačné. Stejný případ by se mohl stát když by uživatel importoval z lokální do online databáze stejnou aktivitu dvakrát, jen třeba nějak upravenou.



Obrázek 2.2: Koncept lokálních databází vs online databáze

Tento problém vyřešíme tím, že při importu dat do Firestore databáze necháme vytvořit úplně nový záznam a tedy i nový identifikátor. Firestore už nám totiž zaručí jedinečnost identifikátoru v rámci databáze. Nicméně změna identifikátoru dané aktivity nám zavádí ještě jeden problém.

Aktivity odkazující se na jiné aktivity (například trénink se odkazuje na cvičení), k těmto odkazům používají právě identifikátory. Tedy když například když tréninku vytvoříme ve Firestoru nový identifikátor a následně vytvoříme nové identifikátory i každému cviku, který v tréninku je, zneplatníme odkazy uvnitř tréninku (viz Obrázek 2.3). Na příkladu máme trénink s identifikátorem `1`, který obsahuje cviky s identifikátory `3, 5`. Při importu do Firestoru se jim změní identifikátory, ale nezmění se seznam identifikátorů odkazující se na cviky. Cviky mají identifikátory `b, c`, ale trénink se stále odkazuje na cviky s identifikátory `3, 5`. Musíme tedy pamatovat na to, že při přesunu dat mezi databázemi, musíme nejen vytvořit nové identifikátory, ale i opravit všechny odkazy uvnitř aktivit, aby používali nově vytvořené identifikátory.



Obrázek 2.3: Chyba v referencích mezi aktivitami

2.6.2 Tag

Nyní již máme dostatek informací, abychom si mohli navrhnout model dat. Začneme tedy úplně nejjednodušší entitou v našem datovém modelu – tagem. Tag má reprezentovat nějakou vlastnost a sám se na žádnou další entitu neodkazovat. Naopak na něj se budou odkazovat cviky. U tagu tedy budeme potřebovat jeho jméno a identifikátor (viz sekce 2.6.1), abychom se na něj mohli odkazovat.

Tag v SQLite

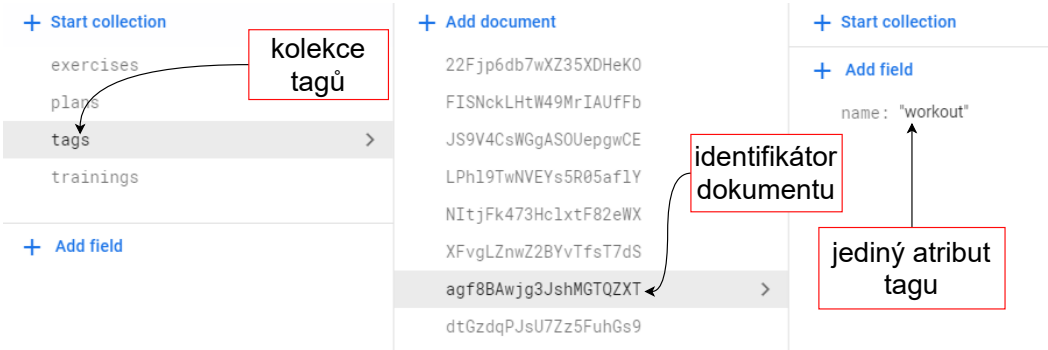
Uložení tagů v SQLite databázi vzhledem k modelu tagu bude naprosto jednoduché. Databáze bude pro tagy obsahovat jednu tabulku, kde bude pouze název tagu a identifikátor tagu, viz příklad (Tabulka 2.1) kde můžeme vidět výsek tabulky tagů se třemi konkrétními tagy: `Do posilovny`, `Posilování břicha`, `Posilování zad`. Sloupec Name reprezentuje jméno tagu a sloupec `Id` reprezentuje jeho identifikátor. `Name` je typu `text` a `Id` je typu `int`.

Id	**Name**
int	**text**
⋮	⋮
50	Do posilovny
51	Posilování břicha
52	Posilování zad
⋮	⋮

Tabulka 2.1: SQL tabulka pro Tagy, s příkladem záznamů.

Tag ve Firestoru

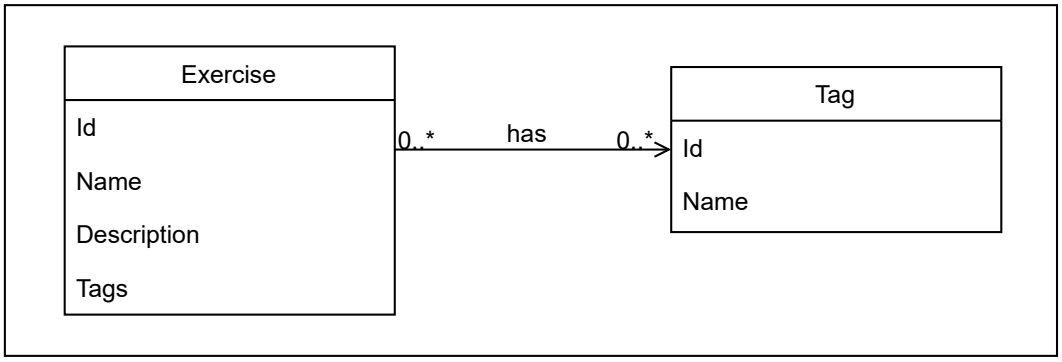
Ve Firestoru budeme potřebovat pouze kolekci tagů, kde každý dokument v této kolekci bude jeden uložený tag. Tag zde bude mít jediný atribut a to sice jméno. Identifikátor tagu bude identifikátor (respektive jméno) daného dokumentu. Na příkladu (viz Obrázek 2.4) vidíme tag `workout`, jehož identifikátor, tedy identifikátor dokumentu, je `agf8BA...`.



Obrázek 2.4: Struktura Firestore databáze pro data v chatu

2.6.3 Exercise – cvik

Cvik bude v aplikaci základní datová třída. Model bude určitě obsahovat jméno a popis cviku jako jeho základní atributy. Uživatelský datový model je naplánovaný tak, že jeden cvik může být použitý v libovolně mnoha trénincích. Právě trénink si bude pamatovat odkazy na cviky, proto cvik potřebuje identifikátor (viz Obrázek 2.5 – `Id`). Cvik si tedy sám nepotřebuje pamatovat, kterých tréninků je součástí. Tato informace by nám pomohla ve chvíli, když uživatel bude chtít daný cvik smazat. Tehdy se cvik musí smazat i ze všech tréninků a my bychom rovnou věděli, v kterých trénincích máme odstraňovat. Protože cvik je základní kámen našeho modelu, ze kterého se vše dále skládá, předpokládáme, že cviků bude mít uživatel uložených hodně a spíš by to tedy vedlo ke zbytečně velkému množství dat navíc. Cvik se ovšem potřebuje odkazovat na tag. Každý cvik může mít libovolně mnoho vlastností, ale nedává smysl aby se nějaká opakovala, tedy nepotřebujeme si pamatovat počty a stačí nám tedy obyčejný seznam tagů (viz Obrázek 2.5 – `Tags`).



Obrázek 2.5: Datový model cviku a tagu

Cvik v SQLite

Základem uloženého cviku v SQLite databázi bude určitě jednoduchá tabulka, podobně jako pro tagy, obsahující pouze atributy cviku. Budou v ní sloupce `Name` pro jméno, `Description` pro popisek a `Id` pro identifikátor, abychom se na cvik mohli odkazovat. Na příkladu vidíme (viz Tabulka 2.2) tabulku s dvěma cviky: `speciální klik, dřep`. Abychom uložili odkazy cviku na tagy, budeme potřebovat ještě druhou tabulku pro reprezentaci vztahu z Obrázku 2.5 – „cvik má tag“. Tato tabulka bude obsahovat sloupec `TagId` pro identifikátor tagu, kterého se vztah týká, sloupec `ExerciseId` pro identifikátor cviku, kterého se to týka a ještě sloupec `Id` pro vlastní identifikátor v rámci tabulky. Například když budeme potřebovat cviku `speciální klik` uložit do databáze dvě vlastnosti, bude to vypadat jako v Tabulce 2.3. Pro oba záznamy bude `ExerciseId` = 1 a `TagId` bude 50 a 52, což odpovídá identifikátorům z tabulky tagů (viz sekce 2.6.2).

Id	Name	Description
int	**text**	**text**
1	speciální klik	lokty podél těla
⋮	⋮	⋮
94	dřep	obyčejný dřep
⋮	⋮	⋮

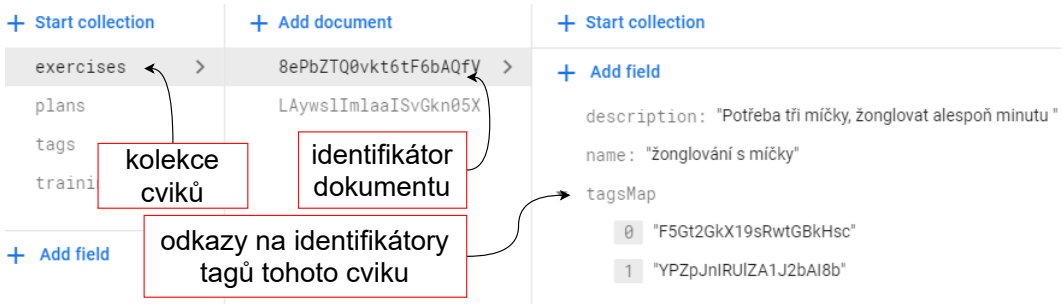
Tabulka 2.2: SQL tabulka pro cviky, s příkladem záznamů.

Id	ExerciseId	TagId
int	**int**	**int**
⋮	⋮	⋮
19	1	50
20	1	52
⋮	⋮	⋮

Tabulka 2.3: SQL tabulka pro vztah „cvik má tag“, s příkladem záznamů.

Cvik ve Firestoru

Ve Firestoru budeme mít samostatnou kolekci pro cviky. V kolekci bude jeden dokument pro každý cvik v databázi a každý cvik bude obsahovat atribut pro jméno, popis a seznam všech tagů ve formě odkazů. Seznam všech tagů se tedy bude skládat z identifikátorů daných tagů. Na příkladu (viz Obrázek 2.6) můžeme vidět cvik `Dřep`, který má dva tagy s identifikátory: `F5Gt...` a `YPZp....` Zároveň v databázi z příkladu jsou vidět v prostředním sloupci 2 různé dokumenty, kde každý je pro jiný cvik.

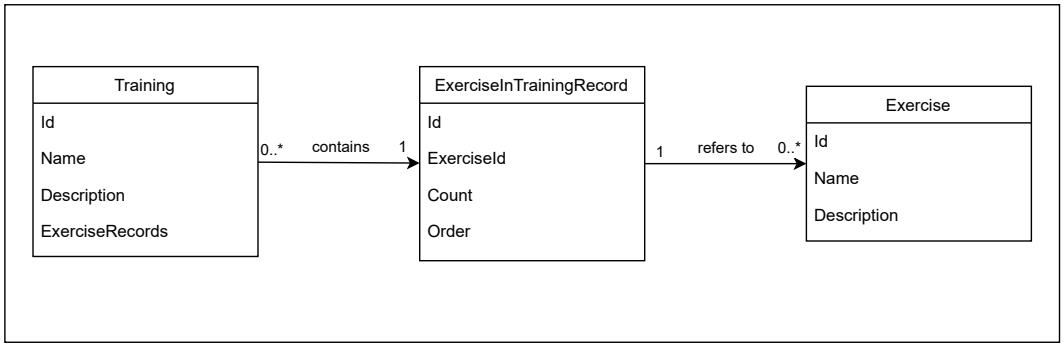


Obrázek 2.6: Strukutra Firestore databáze pro cviky

2.6.4 Training – trénink

Model tréninku bude obsahovat jméno, popis jako základní atributy a identifikátor abychom se na něj mohli odkazovat z tréninkových plánů. Dále již víme, že trénink se bude odkazovat na cviky, ale už nám nebude stačit odkazovat se přímo na cvik, protože cvik se v tréninku může nacházet v libovolném množství a pořadí, tedy pro každý výskyt cviku si potřebujeme pamatovat

jeho množství a jeho pořadí v rámci tréninku. Pro tuto potřebu vytvoříme samostatnou entitu `ExerciseInTrainingRecord` (záznam cviku v tréninku). Entita `ExerciseInTrainingRecord` si bude pamatovat odkaz na daný cvik, jeho počet, na schématu (viz Obrázek 2.7) jako `Count` a pořadí jako `Order`, a protože se na ni bude odkazovat trénink, bude mít také identifikátor. Trénink bude mít seznam těchto entit a pro každý jeden výskyt cviku, zde bude jedna instance `ExerciseInTrainingRecord` – seznam je na Obrázku 2.7 pojmenovaný jako `ExerciseRecords`.



Obrázek 2.7: Datový model tréninku

Trénink v SQLite

V SQLite databázi bude základní tabulka pro trénink stejná jako pro cvik, bude zde znovu: jméno, popis a identifikátor. Na příkladu vidíme (viz Tabulka 2.4) tabulku se dvěma tréninky: `Kardio, Vytrvalostní`. Pak budeme potřebovat tabulku pro záznamy o cvicích. Druhá tabulka bude tedy mít sloupce dle atributů entity `ExerciseInTrainingRecord`, pouze potřebujeme do tabulky přidat informaci o tom, kterému tréninku daný záznam patří, tedy se odkázat na trénink. Tabulka tedy bude obsahovat: identifikátor tréninku – `TrainingId`, identifikátor cviku – `ExerciseId`, počet opakování cviku – `Count`, pořadí v rámci tréninku – `Order` a nakonec vlastní identifikátor – `Id`. Na příkladu vidíme (viz Tabulka 2.5), že jsou v tabulce tři záznamy a všechny patří k tréninku s id = 44. Tedy trénink `Vytrvalostní` má minimálně tyto tři cviky, přičemž cvik s id = 1 je v tréninku alespoň dvakrát.

Id	**Name**	**Description**
int	**text**	**text**
⋮	⋮	⋮
43	Kardio	trénink na zlepšení fyzičky
44	Vytrvalostní	trénink na vytrvalost
⋮	⋮	⋮

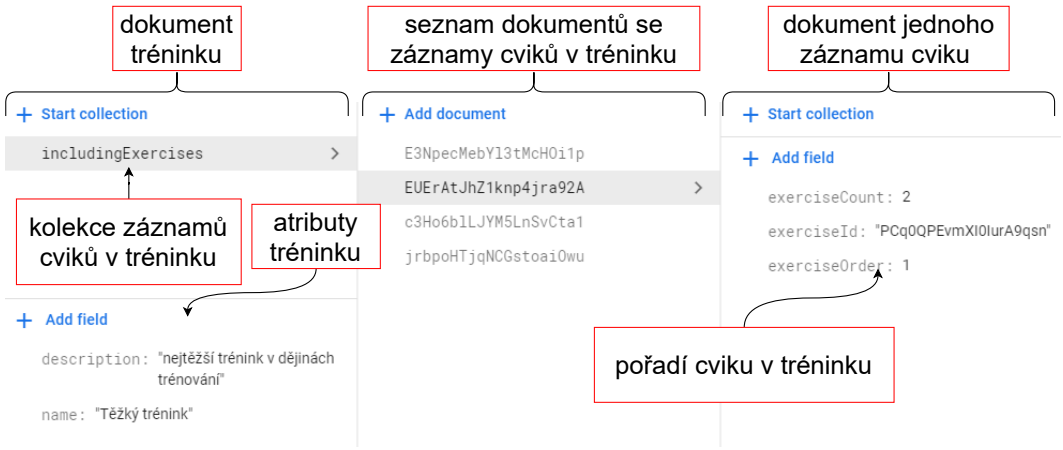
Tabulka 2.4: SQL tabulka pro Tréninky, s příkladem záznamů.

Id	TrainingId	ExerciseId	Count	Order
int	int	int	int	int
⋮	⋮	⋮	⋮	⋮
13	44	1	20	0
14	44	94	15	1
15	44	1	10	2
⋮	⋮	⋮	⋮	⋮

Tabulka 2.5: SQL tabulka pro vlastnost „trénink obsahuje cvik“, s příkladem záznamů.

Trénink ve Firestoru

Pro Firestore databázi vytvoříme kolekci tréninků, kde pro každý trénink bude jeden dokument. V dokumentu budou atributy: jméno a popis. V modelu, který jsme navrhli (viz sekce 2.6.4) má trénink dále seznam záznamů o cviku, tedy každý dokument reprezentující trénink bude mít další vnořenou kolekci. V kolekci bude pro každý záznam o cviku (`ExerciseInTrainingRecord`) další dokument. Tento dokument bude mít jméno (identifikátor) shodné s identifikátorem záznamu a bude obsahovat všechny další atributy daného záznamu tedy: identifikátor cviku, na který se odkazuje, počet a pořadí. Na příkladu (viz Obrázek 2.8) v levé části vidíme dokument tréninku `Těžký trénink` a 4 vnořené záznamy o cviku. Dále na příkladu vpravo je otevřený dokument jednoho záznamu o cviku, který je v daném tréninku 2. v pořadí (číslováno od 0) a cvik se má cvičit v počtu dvou.

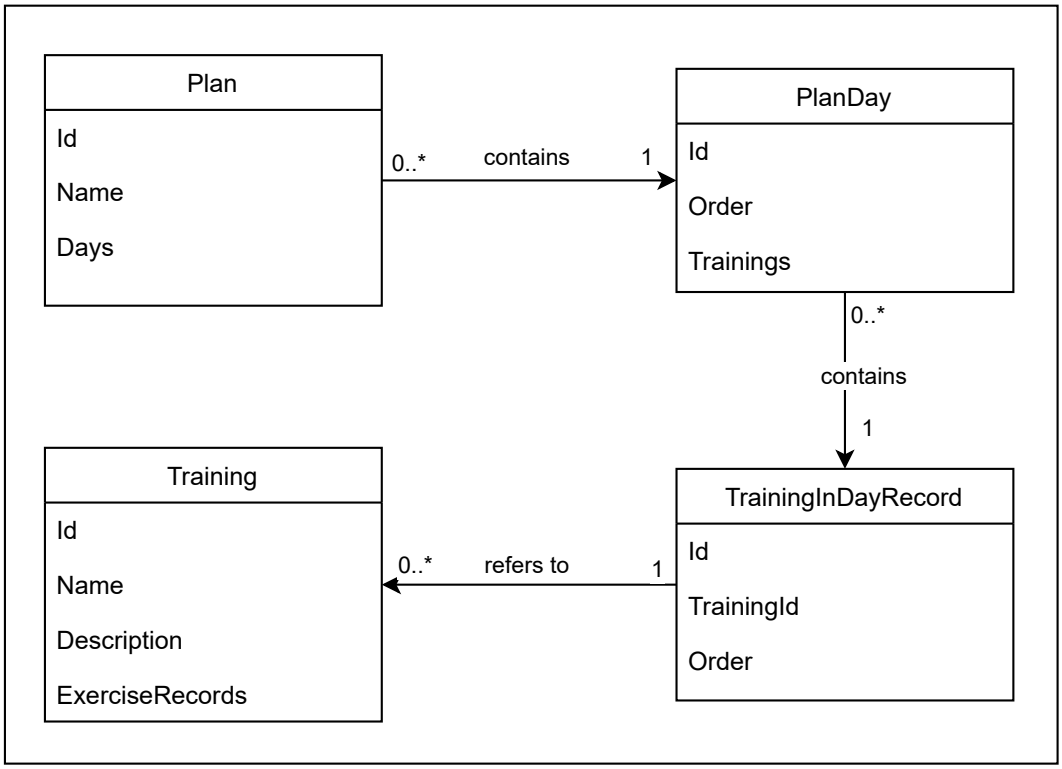


Obrázek 2.8: Strukutra Firestore databáze pro trénink

2.6.5 Plan – tréninkový plán

Tréninkový plán má jediný atribut, svoje jméno. Když na plánu něco změníme, potřebujeme se odkázat do databáze, kde chceme změnit záznam o plánu. I zde tedy potřebujeme identifikátor. Dále pak už jen udržuje reference na dny, které ovšem existují jen v rámci tréninkového plánu – na schématu vidíme, že den je obsažen právě v jednom plánu, tedy samostatně neexistuje a naopak plán obsahuje libovolně mnoho dnů. Entita plánu tedy bude mít tři datové položky: jméno,

identifikátor a seznam dnů. Pro den v plánu vytvoříme entitu `PlanDay`. Den (viz Obrázek 2.9 – `PlanDay`) obsahuje identifikátor, abychom se na něj mohli odkazovat, dále informaci kolikátý v pořadí je a seznam záznamů tréninků (`Trainings`). Podobně, jako jsme vytvořili entitu `ExerciseInTrainingRecord` (viz 2.6.4), i zde potřebujeme vytvořit další entitu – `TrainingInDayRecord` – která reprezentuje jeden trénink v daném dnu. Jak je ve schématu vidět, `PlanDay` může obsahovat libovolně mnoho tréninkových záznamů, ale každý tréninkový záznam je v právě jednom dni, tedy taky nemůže existovat samostatně. Tréninkový záznam, kromě svého identifikátoru, obsahuje identifikátor tréninku na který se odkazuje – `TrainingId` a pořadí – `Order`. Identifikátor tréninku obsahuje pouze jeden, protože reprezentuje pouze jeden konkrétní trénink.



Obrázek 2.9: Datový model tréninkového plánu

Tréninkový plán v SQLite

Pro tréninkové plány nám tabulky budou téměř odpovídat tomu, jak máme model navržený. První tabulka bude pro plán samotný, kde bude pouze jméno a identifikátor plánu. Na příkladu (viz Tabulka 2.6) vidíme tabulku se dvěma plány: `Na posílení nohou, Na hubnutí`. Vzhledem k tomu, že jeden `PlanDay` (viz sekce 2.6.5) může náležet pouze jednomu tréninkovému plánu, není potřeba abychom zde vytvořili samostatnou tabulku pro vztah „plán obsahuje den“ a vytvoříme tedy přímo tabulku pro tuto entitu, která nám tento vztah bude reprezentovat. Ke každému záznamu pouze přidáme identifikátor plánu, kam daný den náleží. Na příkladu (viz Tabulka 2.7) vidíme dva dny patřící plánu s identifikátorem 36 a jeden den patřící plánu s identifikátorem 37. Dále potřebujeme ještě třetí tabulku pro `TrainingInDayRecord`. Entita totiž simuluje vlastnost „den obsahuje trénink“, kde navíc ukládá pořadí v rámci dne. Do tabulky mu-

síme oproti entitě ještě přidat identifikátor dne, ke kterému daný záznam patří. Na příkladu (viz Tabulka 2.8) jsou pro den s identifikátorem 11 dva tréninky a jen jeden trénink pro den s identifikátorem 12.

Id	**Name**
int	**text**
⋮	⋮
36	Na posílení nohou
37	Na hubnutí
⋮	⋮

Tabulka 2.6: SQL tabulka pro plány, s příkladem záznamů.

Id	**PlanId**	**Order**
int	**int**	**int**
⋮	⋮	⋮
11	36	0
12	36	1
13	37	0
⋮	⋮	⋮

Tabulka 2.7: SQL tabulka pro dny, s příkladem záznamů.

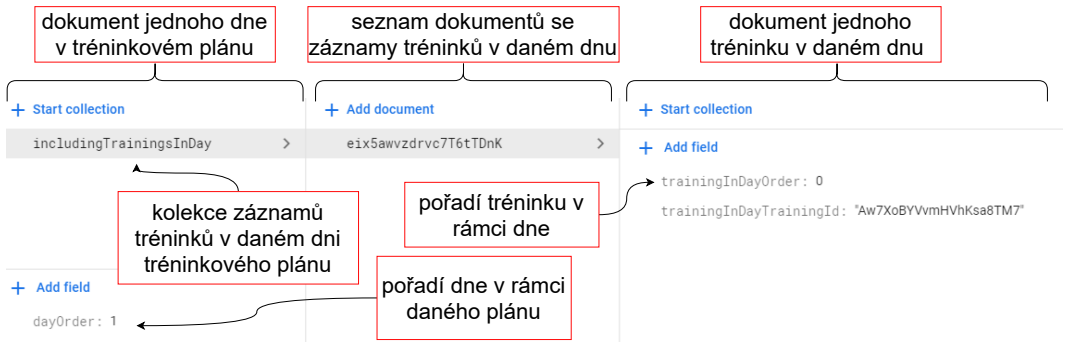
Id	**DayId**	**TrainingId**	**Order**
int	**int**	**int**	**int**
⋮	⋮	⋮	⋮
38	11	43	1
39	11	44	0
40	12	43	0
⋮	⋮	⋮	⋮

Tabulka 2.8: SQL tabulka pro vlastnost „den obsahuje trénink“, s příkladem záznamů.

Tréninkový plán ve Firestoru

Pro tréninkové plány si ve Firestoru uděláme znovu samostatnou kolekci. Pro každý plán bude právě jeden dokument, kde identifikátor dokumentu bude identifikátor plánu. Tento dokument bude obsahovat pouze atribut pro jméno plánu a vnořenou kolekci pro dny z tréninkového plánu. Každý dokument reprezentující jeden den bude mít pouze atribut pořadí. Příklad dokumentu pro den v tréninkovém plánu můžeme vidět v levém sloupci na obrázku (viz Obrázek 2.10). Dokument pro den dále bude obsahovat kolekci pro záznamy o tréninku, respektive

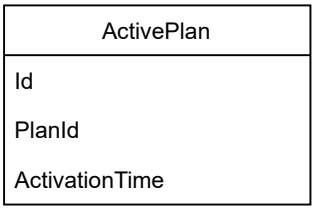
`TrainingInDayRecord` (viz sekce 2.6.5). Každý tento záznam bude mít znovu svůj dokument a bude obsahovat následující atributy: identifikátor tréninku a pořadí tréninku v rámci dne. Na příkladu vidíme otevřený záznam dne, obsahující jediný trénink s identifikátorem `Aw7Xo...`, pro který též vidíme otevřený tréninkový záznam.



Obrázek 2.10: Strukutra Firestore databáze pro den v tréninkovém plánu

2.6.6 Spuštěný (aktivní) tréninkový plán

Pro účely splnění funkčního požadavku č. 40 (viz 1.6), potřebujeme model i pro aktivní tréninkové plány. Protože požadavky definují, že aplikace si musí pamatovat aktivní tréninkové plány, půjde o model, který se jistě bude ukládat do databáze. V rámci databáze ho budeme potřebovat identifikovat, abychom ho mohli po skončení smazat, proto budeme potřebovat identifikátor. Dále se budeme potřebovat odkazovat na daný plán, budeme tedy potřebovat identifikátor daného plánu – `PlanId`. Nakonec si budeme potřebovat pamatovat, kdy byl plán aktivován, abychom věděli, který den plánu je. Model tedy bude mít atribut času – `ActivationTime` (viz Obrázek 2.11).



Obrázek 2.11: Datový model aktivního tréninkového plánu

Spuštěný tréninkový plán v SQLite

Protože jde o jednoduchou strukturu, bez žádných mapování, žádných složitějších struktur, půjde pro model aktivního plánu vytvořit tabulka v SQLite přesně odpovídající modelu. Tedy tabulka bude mít tři sloupky (viz Tabulka 2.9) – `Id`, `PlanId` a `ActivationTime` typu `datetime`. Na příkladu je tabulka s dvěma aktivními tréninkovými plány, kde první má v databázi id = 1 a byl spuštěn 15. prosince 2021 v 08:52 hodin.

Id	PlanId	ActivationTime
int	**text**	**datetime**
1	1	2021-12-15 08:52:25
2	2	2021-12-23 15:01:19

Tabulka 2.9: SQL tabulka pro aktivní plány, s příkladem záznamů.

Spuštěný tréninkový plán ve Firestoru

Ve Firestoru budou aktivní tréninkové plány pouhou kolekcí dokumentů, které budou mít právě ty dva atributy, které jsme si definovali při definici modelu – tedy čas aktivace tréninku a identifikátor tréninku. Otázkou zůstává, kam danou kolekci umístíme. Aktivní tréninkové plány se úzce týkají tréninkové databáze, nechceme aby k tomu měl přístup někdo jiný, respektive jsou to data specifická pro uživatele, můžeme tedy tuto kolekci přidružit ke kolekcím tréninkových aktivit (viz sekce 2.6.7).

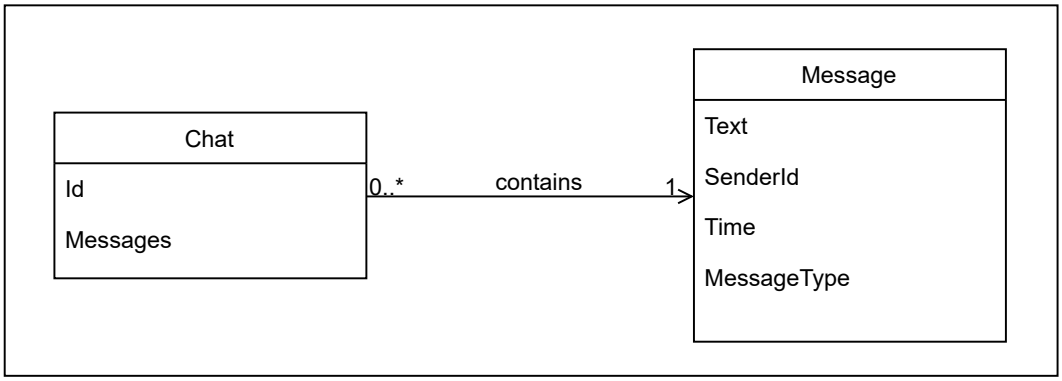
2.6.7 Databáze tréninkových aktivit ve Firestoru

Nakonec je ještě potřeba si říct, kam ve Firestore databázi umístíme kolekce pro tréninkové aktivity, které jsme si právě namodelovali a navrhli jejich strukturu ve Firestore. Mohli bychom kolekce aktivit dát do počátečního dokumentu, tedy jako výchozí kolekce naší databáze. Kdybychom tak učinili, tak s tím jak je ukládání ve Firestoru momentálně navrženo, bychom neměli jak rozeznat data jednotlivých uživatelů od sebe navzájem. Vytvoříme tedy kolekci se stejnou vnitřní strukturou, pro každého uživatele. Vytvoříme kolekci pro tréninkové aktivity, kde bude pro každého uživatele jeden dokument s identifikátorem shodným s identifikátorem daného uživatele a až tento dokument bude obsahovat kolekce pro jednotlivé tréninkové aktivity. Tyto kolekce pak pro každého uživatele budou mít stejnou strukturu.

2.6.8 Chat

Pro entitu chatu nám stačí přemýšlet jen v kontextu přihlášeného uživatele, tedy v kontextu online databáze. Z pohledu uživatele jde jen o seznam zpráv, my v rámci databáze potřebujeme jednotlivé chaty rozlišovat, tedy potřebujeme identifikátor. Dále potřebujeme entitu pro zprávu(viz Obrázek 2.12). Zpráva si z pohledu uživatele jistě potřebuje pamatovat text, čas odeslání a autora zprávy. Čas odeslání ani nemusí být uživateli dostupný, ale hraje pro něj roli, aby měl zprávy seřazené podle času odeslání. Autor zprávy ve většině chatovacích aplikací není explicitně nadepsaný, ale nějak se například odlišuje barva nebo umístění zprávy na obrazovce podle toho kdo ji odeslal. Podobně budeme postupovat při návrhu grafického rozhraní (viz sekce 2.8.8). Tedy i odesílatel zprávy je pro uživatele důležitý. Z našeho pohledu si stačí pamatovat identifikátor autora a porovnávat ho s aktuálním uživatelem – tedy tím, který právě aplikaci používá (viz Obrázek 2.12 – `SenderId`). Identifikátor pro model zprávy nepotřebujeme. Budeme totiž vždy načítat všechny zprávy a nebudeme potřebovat je navzájem odlišit, ale pouze seřadit podle času odeslání.

Protože v rámci chatu chceme posílat i tréninkové aktivity, chceme je nějak odlišit od zbytku zpráv, jednak vizuálně (viz 1.6. funkční požadavek č. 33), ale také v tom jak se s danou zprávou nakládá. Přidáme si do zprávy textové pole `MessageType`, kam pro každou zprávu vložíme předdefinovanou konstantu, co daná zpráva obsahuje. V tomto bodě nepotřebujeme žádnou další podporu pro zasílání tréninkových aktivit, protože nám stačí identifikátor dané aktivity, který můžeme ve zprávě vložit do pole pro text. Funkční požadavky nám nedefinují, že bychom chtěli mít možnost odeslat text a tréninkovou aktivitu v jedné zprávě, tedy si to můžeme dovolit. Zároveň nastavíme patřičně typ zprávy – textová zpráva, zpráva s cvikem, zpráva s tréninkem, zpráva s tréninkovým plánem.



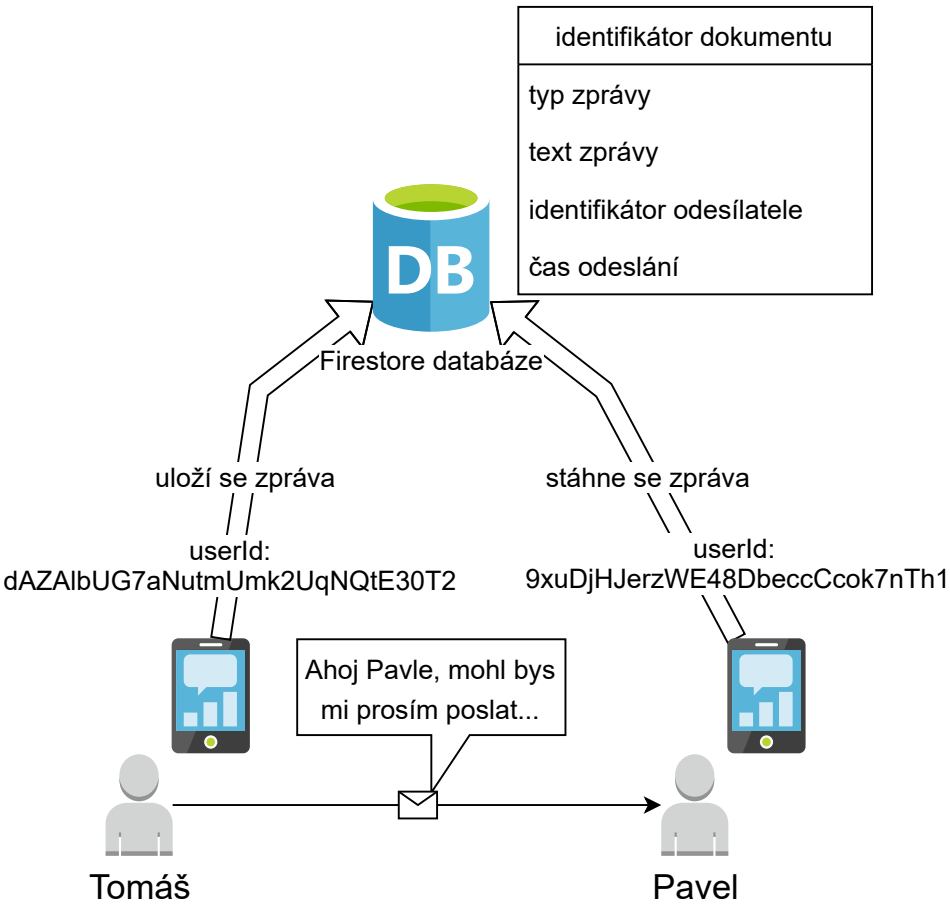
Obrázek 2.12: Datový model chatu

Chat ve Firestoru

Pro chaty si vytvoříme kolekci v počátečním dokumentu. V této kolekci chatů pak bude každý dokument představovat právě jeden chat. Tento dokument bude odrážet strukturu modelu, který jsme si navrhli (viz sekce 2.6.8), tedy zde bude zanořená kolekce zpráv. Dokument s uloženou zprávou pak bude obsahovat atributy: text zprávy, typ zprávy, čas odeslání a identifikátor odesílatele. Model zpráv v chatu jsme navrhli tak, že pokud chceme poslat nějakou tréninkovou aktivitu, změníme typ zprávy a do textu zprávy vložíme identifikátor dané tréninkové aktivity (viz sekce 2.6.8). Tento identifikátor odkazuje na dokument, ve kterém je aktivita uložená. Ovšem my chceme aby daná aktivita nebyla uložená v databázi uživatele, ale v databázi chatu, protože z osobní databáze může uživatel danou aktivitu smazat nebo změnit. Chceme tedy pro každý chat ještě definovat databázi tréninkových aktivit, tedy každý chat bude mít kromě kolekce zpráv, ještě kolekci cviků, tréninků a tréninkových plánů. Tyto tři kolekce budou přesně kopírovat strukturu databáze tréninkových aktivit (viz sekce 2.6.3, 2.6.4, 2.6.5). Záměrně jsme vynechali kolekci tagů, poněvadž v rámci chatu nepotřebujeme tagy spravovat, tudíž nám stačí, když jsou uloženy v rámci datového modelu cviku (viz sekce 2.6.3). V modelu cviku jsou ovšem tagy uloženy pouze jako seznam identifikátorů. Pro potřeby chatu nám naopak bude stačit seznam jmen tagů, který nahradí seznam identifikátorů.

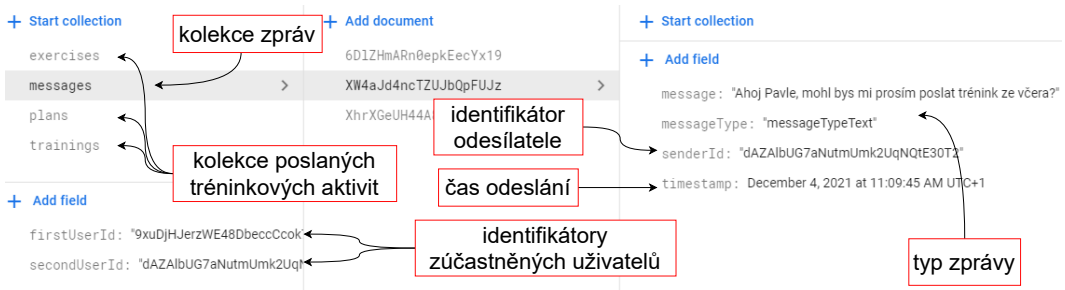
Nakonec přidáme chatu ještě atributy s identifikátory uživatelů, kterých se chat týká. Identifikátory uživatelů si zde potřebujeme pamatovat, protože chceme dovolit čtení a zápis do dokumentu daného chatu jen těm uživatelům, kterých se to týká (viz podkapitola 2.7).

Ukážeme si, co proběhne když si dva uživatelé budou chtít poslat zprávu. Na schématu (viz Obrázek 2.13) vidíme uživatele *Tomáše*, který chce poslat textovou zprávu *Pavlovi*. Zpráva se ve Firestoru nejprve uloží jako dokument, kde se uloží informace o tom, že jde o textovou zprávu, kdy jí dotyčný poslal, odesílatelův identifikátor a text zprávy. Potom si ji uživatel *Pavel* může načíst ve svém telefonu.



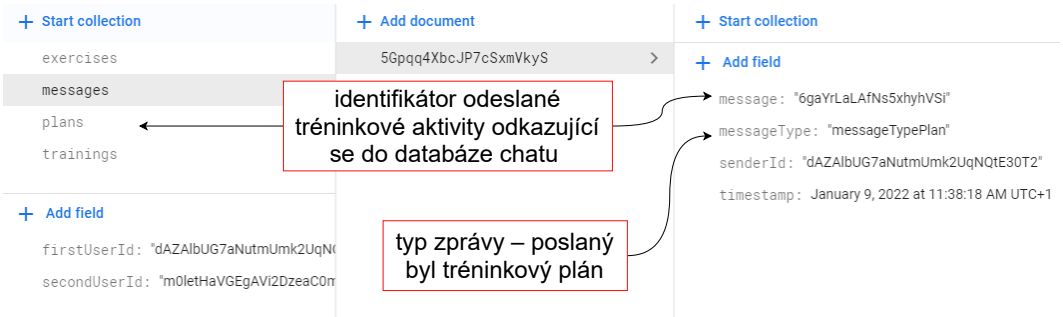
Obrázek 2.13: Proces posílaní zpráv v chatu

Na příkladu (viz Obrázek 2.14) vidíme textovou zprávu (v pravém sloupci), kterou jsme si naznačili na schématu výše (viz Obrázek 2.13). V dokumentu je vidět, že zpráva byla odeslána 4. prosince, obsahuje náš text a obsahuje informace kdo jí poslal a typ zprávy. Z prostředního sloupce pak můžeme vyčíst, že v chatu mezi uživateli z identifikátory `9xuDj...` a `dAZAlb...` jsou zatím jen tři zprávy.



Obrázek 2.14: Strukutra Firestore databáze pro data v chatu

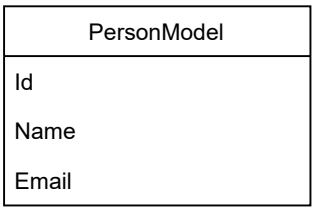
Ukážeme si ještě jeden příklad, odeslání tréninkové aktivity v chatu. Na obrázku níže (viz Obrázek 2.15) máme dokument jednoho chatu, s jedinou zprávou. Zpráva již není textová, nýbrž je typu `messageTypePlan`, což definuje zprávu pro tréninkový plán. Dále atributy `senderId` a `timestamp` obsahují stejný typ hodnot, jako textové zprávy (viz Obrázek 2.14). Posledním atributem je `message`, který v tomoto případě obsahuje identifikátor odeslaného plánu. Identifikátor se neodkazuje do odesílatelovy databáze, ale do databáze tohoto chatu. V tomto případě protože jde o tréninkový plán, identifikátor se odkazuje do kolekce `plans`, kterou můžeme vidět v levé části obrázku. Všechny další aktivity, které daná tréninková aktivita obsahuje (odkazuje se na ně), se také uloží do databáze chatu, do příslušných kolekcí `plans, trainings, exercises`. Při odeslání tréninku nebo tréninkového plánu se navíc musí odpovídajícím způsobem ošetřit odkazy v rámci dané aktivity tak, aby odpovídali identifikátorům aktivit, na než se odkazují, v databázi chatu (viz sekce 2.6.1).



Obrázek 2.15: Strukutra Firestore databáze tréninkovou aktivitu v chatu

2.6.9 PersonModel - model uživatele

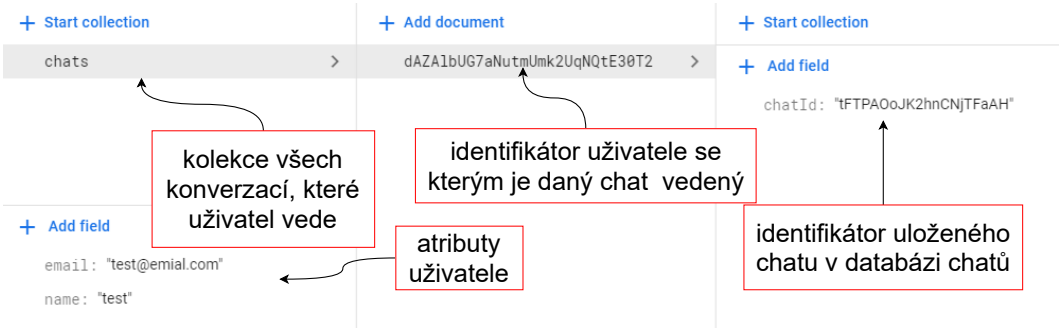
Ačkoliv nechceme ukládat žádná speciální data o uživatelích (data narození, profilové obrázky apod.), potřebujeme pro uživatele model. Součástí funkčních požadavků je totiž i možnost vyhledávání v uživatelích a tedy tyto data budeme ukládat do databáze a v aplikaci něco z nich načítat, když uživatel bude hledat jiného uživatele. Uživatele potřebujeme v databázi rozlišovat i se na ně odkazovat, tudíž potřebujeme identifikátor. O uživateli nám pak už stačí si pamatovat jen jméno (přezdívku) a jeho email, abychom mohli uživatele identifikovat i z pohledu jiného uživatele a tedy abychom splnili funkční požadavek o vyhledáváni uživatelů (viz podkapitola 1.6 – požadavek 28).



Obrázek 2.16: Datový model pro data o uživateli

Uživatelé ve Firestoru

Uživatelský model můžeme uvažovat jen v rámci Firestoru, protože v offline režimu aplikace nepoužíváme koncept uživatele, protože ho není potřeba. V rámci databáze uživatelů chceme ukládat data z modelu, který jsme si pro uživatele vytvořili (viz sekce 2.6.9). Vytvoříme si tedy kolekci uživatelů a použijeme zde znovu koncept, že pro každého uživatele zde bude jeden dokument s identifikátorem shodným s identifikátorem daného uživatele. V tomto dokumentu rovnou uložíme všechna data modelu (již bez identifikátoru – ten je jako identifikátor dokumentu), tedy zde budou atributy: jméno a e-mail. Součástí uživatelových dat bychom chtěli udělat i seznam všech jeho chatů, abychom mohli uživateli rychle zobrazit všechny jeho chaty a s kým daný chat uživatel vede. Vyhneme se tím procházení celé kolekce chatů, kde může být až `N nad 2` dokumentů při N uživatelích a vybírání jen těch, které jsou daného uživatele. Vytvoříme tedy vnořenou kolekci chatů, kde pro každý uživatelův chat bude jeden dokument, jehož identifikátor bude shodný s identifikátorem uživatele, se kterým si náš uživatel chatuje a dokument bude obsahovat pouze identifikátor chatu – tedy odkaz do databáze chatů. Na příkladu (viz Obrázek 2.17) vidíme uživatele se jménem `test`, který vede jediný chat s uživatelem, jehož identifikátor je `dAZA1...`. Chat bychom pak našli v databázi chatů pod identifikátorem `tFTPA...`.



Obrázek 2.17: Strukutra Firestore databáze pro data uživatele

2.7 Bezpečnost uvnitř Firestoru

Na rozdíl od offline databáze, ve Firestoru je potřeba zabezpečit data. Bezpečností dat zde myslíme, aby nebylo možné dostat se k datům jiného uživatele nebo se dokonce dostat k datům bez přihlášení. Pro přístup do Firestore databáze se používá knihovna ve spojení s konfiguračním souborem. Mohlo by se tedy stát, že by někdo daný soubor získal a snažil by se data v databázi nějak poškodit či změnit. Pro řešení tohoto problému Firestore nabízí jednoduchá pravidla, která blokují a zakazují přistup k datům. Pomocí těchto pravidel, které je možno editovat přímo ve webovém rozhraní Firestoru, lze odlišit čtení a zápis dat, tudíž lze například globálně povolit čtení ale zakázat zápis. Pravidla se skládají z cesty k datům, jakou činnost nad daty chceme povolit nebo zakázat a za jaké podmínky tak chceme učinit. Na příkladu (viz kód níže) je pravidlo, které povoluje číst a editovat data v dokumentu s detaily o uživateli, jen jeho vlastníkovi. Pravidlo nejdříve definuje cestu, zároveň v cestě definuje proměnnou `userId` – máme tedy dokument, jehož identifikátor máme v proměnné `userId`.

V rámci definované cesty pak pravidlo říká co povoluje (zakazuje) a nakonec za jaké podmínky. V našem případě se povoluje čtení a zápis (`allow read, write`) pokud: `request.auth` není `null`, to znamená že musí být někdo na daném zařízení přihlášený, a zároveň pokud se identifikátor přihlášeného uživatele shoduje s identifikátorem daného dokumentu.

```
//Ukázka bezpečnostního pravidla ve Firestoru
match /users/{userId}{
    allow read, write: if request.auth != null
    && request.auth.uid == userId
}
```

Těmito pravidly bychom chtěli popsat všechna data, která v databázi jsou, aby je nemohl číst a upravovat někdo jiný než uživatel naší aplikace. Zároveň chceme mít taková pravidla, aby jeden uživatel mohl číst a editovat jen svoje záznamy. Tedy celou databázi tréninkových aktivit chceme pro čtení a úpravy zpřístupnit pouze danému uživateli. To pro nás nebude problém, protože celá databáze tréninkových aktivit jednoho uživatele je v souboru s identifikátorem shodným s uživatelovým identifikátorem. Definujeme tedy cestu k tomuto souboru a pak již povolíme vše co je zanořené v daném dokumentu.

Dále chceme nastavit pravidla pro databázi chatů, která musí být přístupná všem účastníkům chatu, ale ostatním uživatelům nikoliv. Uživatelé, kterých se chat týká musí být schopni číst všechny položky daného chatu a vytvářet zde nové záznamy. Chceme tak učinit v kolekci zpráv i v kolekcích tréninkových aktivit. K tomu jsme si již připravili atributy v dokumentu chatu – v každém chatu jsou identifikátory těch dvou uživatelů, kterých se chat týká. A protože tyto pravidla umějí i porovnávat data v dokumentu (viz dokumentace Firebase [5]) a nejen identifikátory dokumentů, můžeme identifikátor uživatele porovnat s oběma identifikátory uloženými v chatu. Pravidla také nabízí rozlišit libovolný zápis (vytvoření, editace, smazání záznamu) a pouhé vytvoření záznamu, pomocí klíčových slov `write` a `create`. Protože ve funkčních požadavcích (viz podkapitola 1.6) jsme si nedefinovali možnost editace zpráv, nebudeme ji zatím vůbec podporovat. V chatu tedy povolíme pouze vytváření nových záznamů. V případě, že bychom chtěli v budoucnu povolit editaci zpráv nebo odeslaných tréninkových aktivit, vytvořili bychom nové pravidlo s klíčovým slovem `write`, které by povolilo editaci tomu uživateli, který je původce dané zprávy nebo aktivity. Tímto bychom zabránili například vzájemné editaci zpráv mezi uživateli.

Pro databázi uživatelů, bychom chtěli dát práva měnit data uživateli jen o sobě (viz příklad výše), ale číst (nikoliv editovat) data i o ostatních uživatelích, abychom mohli v uživatelích vyhledávat (viz 1.6, požadavek 28). Vzhledem k tomu, jak jsme si navrhli strukturu dat uživatele (viz sekce 2.6.9) ve Firestoru, nám vyvstává ještě jeden problém. Součástí těchto dat je i seznam chatů, kterých se uživatel účastní. Když tedy někomu poprvé napíše, vytvoří se nový dokument. Potřebujeme ale, aby vznikl tento záznam i druhému uživateli, tedy v datech jiného uživatele, než který inicioval danou konverzaci. Chceme tedy ještě povolit libovolnému přihlášenému uživateli vytvářet nové záznamy v kolekci chatů libovolného uživatele.

2.8 Uživatelské rozhraní aplikace

Protože vyvíjíme mobilní aplikaci pro Android, přirozeně chceme, aby se v našem grafickém rozhraní uživatel snadno vyznal a aby aplikace byla k uživateli přívětivá (viz cíl práce 3). V této podkapitole se tedy zaměříme na návrh takového grafického rozhraní, které budeme považovat za přehledné a přívětivé. Budeme se při tom snažit držet již fungujících řešení, na kterých dnes staví hodně úspěšných aplikací (viz sekce 1.5.5).

2.8.1 Rozdělení na obrazovky

Z pohledu uživatele dává smysl, abychom aplikaci rozdělili na funkční celky, které budou jasně viditelně odděleny od zbytku. Tedy například chceme jasně oddělit obrazovku chatu od obrazovky se spuštěným tréninkem. Definujeme si zde obrazovky, které chceme navzájem jasně oddělit a u těch vizuálně jednodušších si i popíšeme jejich vzhled. Naopak pro ty složitější si vyhradíme samostatnou sekci. Konkrétně tedy budeme chtít vizuálně izolovat následující funkce a obrazovky:

- **Domovská obrazovka**

  Ve funkčních požadavcích ani nikde jinde nemáme specifikovanou hlavní (domovskou) obrazovku, ani nevíme co by měla být její funkce. Nicméně zamyslíme se, co se uživateli zobrazí když aplikaci zapne. Není důvod proč by se měla rovnou zobrazit nějaká konkrétní funkce. Vytvoříme tedy domovskou obrazovku, kam v případě potřeby můžeme umístit libovolnou věc, například průvodce aplikací, odkazy na poslední použité funkce apod. Úvodní obrazovku tedy zatím ponecháme prázdnou pro případ potřeby.

- **Chat**

  Chatem zde myslíme jednu konkrétní konverzaci (viz podkapitola 1.6 – požadavek 29). Tato obrazovka bude podobná jiným aplikacím s chatem, více viz sekce 2.8.8

- **Seznam konverzací – chatů**

  Stejně jako je tomu v jiných chatovacích aplikacích, chceme mít přehledně seznam všech započatých konverzací. Položky seznamu by měli být nadepsané jménem uživatele, se kterým je konverzace vedena. Nic víc na této obrazovce nepotřebujeme.

- **Procházení databáze tréninkových aktivit**

  Na této obrazovce chceme uživateli umožnit zobrazit seznam všech jeho tréninkových aktivit v databázi (viz podkapitola 1.6 – požadavky 6, 11, 16), ideálně odděleně cviky, tréninky a plány. Dále zde bude možnost filtrování pomocí tagů. Více viz sekce 2.8.5.

  - **Zobrazení detailu konkrétní tréninkové aktivity**

    Protože je hodně věcí co je třeba v rámci aktivit zobrazit (u plánů může jít o dlouhý seznam dní) a zároveň na těchto obrazovkách chceme umožnit editaci aktivit, chceme jejich detail jako samotnou obrazovku. Více viz sekce 2.8.5.

- **Spravování databáze tagů**

  Databázi tagů chceme vizuálně oddělit od databáze tréninkových aktivit, protože se nejedná o tréninkovou aktivitu, nýbrž pouze o databázi používaných vlastností. Zde bude uživateli umožněno zobrazit seznam všech jeho tagů a případně některý smazat (viz podkapitola 1.6 – požadavek 19). Více viz sekce 2.8.4.

- **Spuštění tréninku**

  Tato obrazovka má za cíl zobrazit vybraný trénink, který uživatel chce zrovna cvičit (viz podkapitola 1.6 – požadavek 18). Více v sekci 2.8.6.

- **Všechny spuštěné tréninkové plány**

  Tato obrazovka má za cíl zobrazit všechny momentálně aktivní tréninkové plány, nic víc.

- **Konkrétní spuštěný tréninkový plán**

  Tato obrazovka má za cíl zobrazit jeden vybraný aktivní tréninkový plán, včetně informace, který den v plánu právě je. Více v sekci 2.8.7.

- **Zobrazení nebo editace konkrétní tréninkové aktivity**

  V rámci procházení databáze aplikace musí umožnit uživateli i detailně zobrazit nebo upravit konkrétní tréninkovou aktivitu (viz podkapitola 1.6 – požadavky 2, 3, 9, 10, 14, 15). Více v sekci 2.8.5.

- **Import dat**

  Importem dat se myslí import tréninkových aktivit z offline databáze do online databáze (viz podkapitola 1.6 – požadavek 27). Jediný vstup od uživatele, který potřebujeme pro import nějaké tréninkové aktivity, je vybrat danou aktivitu. K tomuto účelu použijeme dialogy (viz sekce 2.8.2 – Výběr tréninkové aktivity). Více grafické rozhraní nemusí nabídnout.

- **Vyhledávání uživatelů** Aby uživatel mohl s jiným uživatelem začít konverzaci, potřebuje nejdříve daného uživatele vyhledat, potřebujeme tedy seznam všech uživatelů, ve kterém můžeme vyhledávat (viz podkapitola 1.6 – požadavek 28). Obrazovka tedy bude obyčejný textový seznam, reprezentující uživatele.

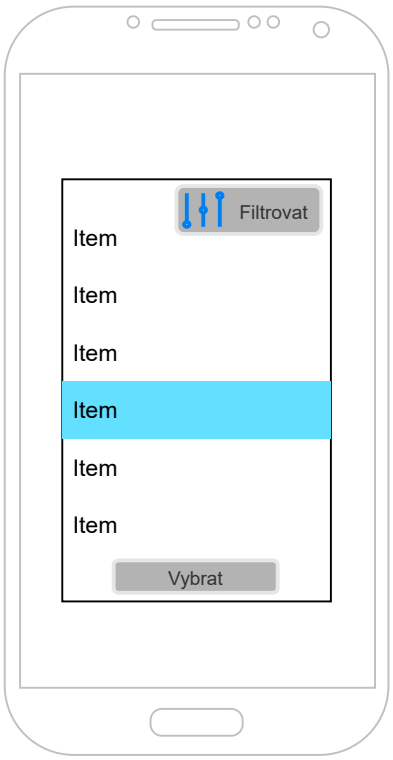
2.8.2 Dialogy

V rámci shrnutí konkurenčních řešení (viz sekce 1.5.5) jsme si na jiných aplikacích ukázali použití vyskakovacích dialogů. Tyto dialogy mají široké spektrum použití. Můžeme je použít ve formě „are you sure“ dialogů, kdy se například před provedením operace, která něco nenávratně změní, zeptáme uživatele, zda-li si opravdu přeje operaci provést. Další možností je použití pro zobrazení detailu nějaké položky. V naší aplikaci budeme dialogy používat tam, kde chceme dodat nějakou pomocnou funkci jiné funkce, například když bychom při procházení databáze chtěli tréninkové aktivity filtrovat podle tagů (viz 2.8.2 – Dialog pro filtrování podle tagů). Nespornou výhodou dialogů je zaprvé možnost jich mít

více přes sebe, respektive že můžeme uživateli zobrazit jeden dialog a na základě jeho vstupu ještě další, a zadruhé můžeme ten stejný dialog použít na více místech, například zmíněný dialog na filtrování podle tagů můžeme použít dále i při výběru tréninkové aktivitu, kterou chceme poslat někomu jinému.

Dialog pro výběr tréninkové aktivity

Pokud uživatel bude chtít poslat tréninkovou aktivitu někomu jinému (viz podkapitola 1.6 – požadavky 30, 31, 32), bude ji potřebovat vybrat z databáze. Stejně tak pokud bude chtít importovat z offline databáze dodatečně jen konkrétní tréninkovou aktivitu (viz podkapitola 1.6 – požadavek 27), bude potřeba vybrat kterou aktivitu uživatel chce. Zde se nám tedy bude hodit dialog pro výběr tréninkové aktivity. V dialogu by se měl zobrazit seznam všech aktivit daného typu (cviky, tréninky, plány). Navíc, protože každá z aktivit má být filtrovatelná pomocí tagů (viz podkapitola 1.6 – požadavky 7, 12, 17), měla by součástí tohoto dialogu být i možnost filtrování pomocí tagů. K tomuto účelu použijeme dialog pro filtrování (viz 2.8.2 – Dialog pro filtrování podle tagů). Na obrázku (viz Obrázek 2.18) vidíme návrh takového dialogu. Dialog zobrazuje seznam všech položek. Po stlačení tlačítka `Vybrat` se označená položka vybere a pošle k dalšímu zpracování. Dialog také obsahuje tlačítko s ikonkou pro filtrování položek.



Obrázek 2.18: Návrh vzhledu dialogu pro výběr tréninkové aktivity

Dialog pro filtrování podle tagů

Každá tréninková aktivita má být filtrovatelná podle tagů (viz podkapitola 1.6 – požadavky 7, 12, 17) a této vlastnosti se dá využít na více místech v aplikaci. Lze využít pro procházení databáze a také pro výběr nějaké položky z databáze ve

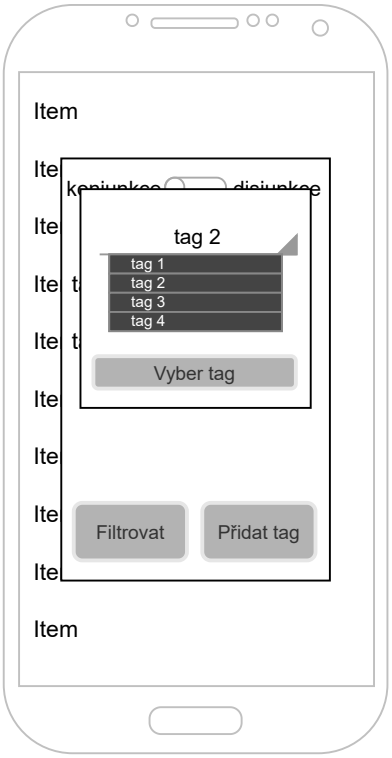
vyskakovacím dialogu (viz 2.8.2 – Dialog pro výběr tréninkové aktivity). Pro tuto funkci si zvolíme také dialog, protože je to funkce navázaná na jinou funkci (např. prohlížení databáze). Na příkladu vidíme (viz Obrázek 2.19) jak by takový dialog mohl vypadat. Funkce tohoto dialogu se přímo odvíjí od funkčních požadavků pro filtrování tréninkových aktivit (viz podkapitola 1.6 – požadavky 7, 12, 17). Dialog musí být schopný přidávat další tagy k filtrování a musí nabídnout možnost filtrovat konjunkci tagů nebo disjunkci tagů. Zároveň můžeme předpokládat, že běžný uživatel by nemusel znát slova *konjunkce* a *disjunkce*, zvolíme zde proto slova *a zároveň, nebo*. Pro přidávání dalších tagů použijeme další dialog (viz 2.8.2 – Dialog pro výběr tagu) Na našem příkladu máme vybrané dva tagy a chceme aby filtrované položky měly oba tagy.

Obrázek 2.19: Návrh vzhledu dialogu pro filtrování pomocí tagů

Dialog pro výběr tagu

Když uživatel potřebuje vybrat tag, je to od uživatele drobný vstup, který ovlivní něco dalšího. V našem případě použijeme dialog pro výběr tagu jednak v dialogu pro filtrování (viz 2.8.2 – Dialog pro filtrování podle tagů) a zadruhé v detailu cviku (viz 2.8.5 – Detail cviku), kde uživatel může chtít cviku přiřadit další tag. Na obrázku (viz Obrázek 2.20) je návrh takového dialogu. Dialog obsahuje pouze spinner[3] pro výběr požadovaného tagu a tlačítko pro jeho přidání.

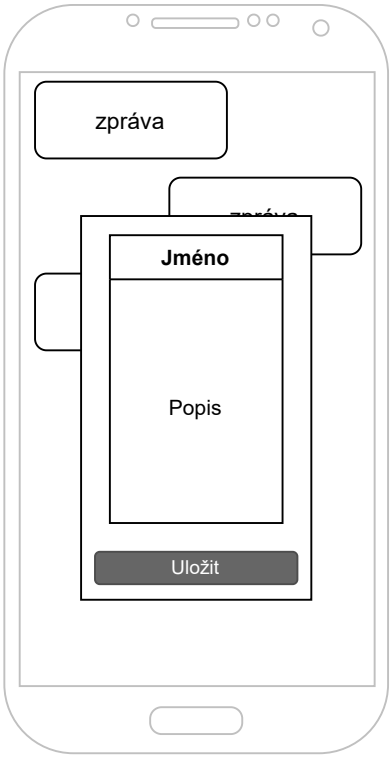
[3]spinner je widget pro Android, který po kliknutí rozbalí položky v něm a umožní uživateli nějakou vybrat – viz Xamarin Android dokumentace – Spinner [11]



Obrázek 2.20: Návrh vzhledu dialogu pro vybrání tagu

Dialog pro zobrazení tréninkové aktivity v chatu

V rámci chatu aplikace umožní uživateli posílat tréninkové aktivity. Na druhé straně je uživatel musí být také schopný přijmout – respektive zobrazit danou aktivitu přímo v chatu a případně ji uložit k sobě do databáze (viz podkapitola 1.6 – požadavky 34, 35, 36, 37). Aby uživatel neztratil kontakt s obrazovkou chatu, vytvoříme pro tuto funkci dialog, který zobrazí danou aktivitu a nabídne uživateli ji uložit k sobě do databáze. Na návrhu (viz Obrázek 2.21) vidíme dialog vykreslený přes chat, který ukáže jméno aktivity a její popis. V případě tréninkového plánu nezobrazí popis, protože plán ho nemá, ale může zobrazit jiný detail nebo nic. Nakonec je zde tlačítko pro uložení dané aktivity k uživateli do databáze.



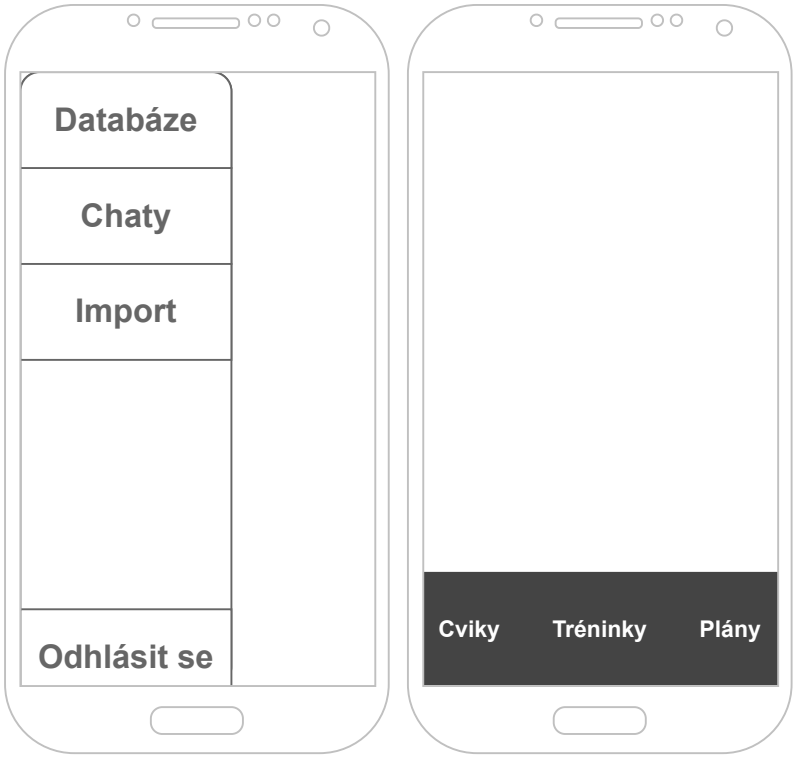
Obrázek 2.21: Návrh vzhledu dialogu pro zobrazení detailu tréninkové aktivitu v chatu

Dialogy závěrem

V této sekci jsme si navrhli jak by měli vypadat dialogy, které nějak ovlivňují funkcionalitu. Nezmínili jsme zde již dialogy, které považujeme za primitivní, ale přece by našli v aplikaci uplatnění. Mezi takové dialogy řadíme například již zmíněný „are you sure“ dialog (viz sekce 2.8.2) nebo například dialog, který pouze zobrazí textovou informaci uživateli – například dialog s nápovědou.

2.8.3 Navigační rozhraní

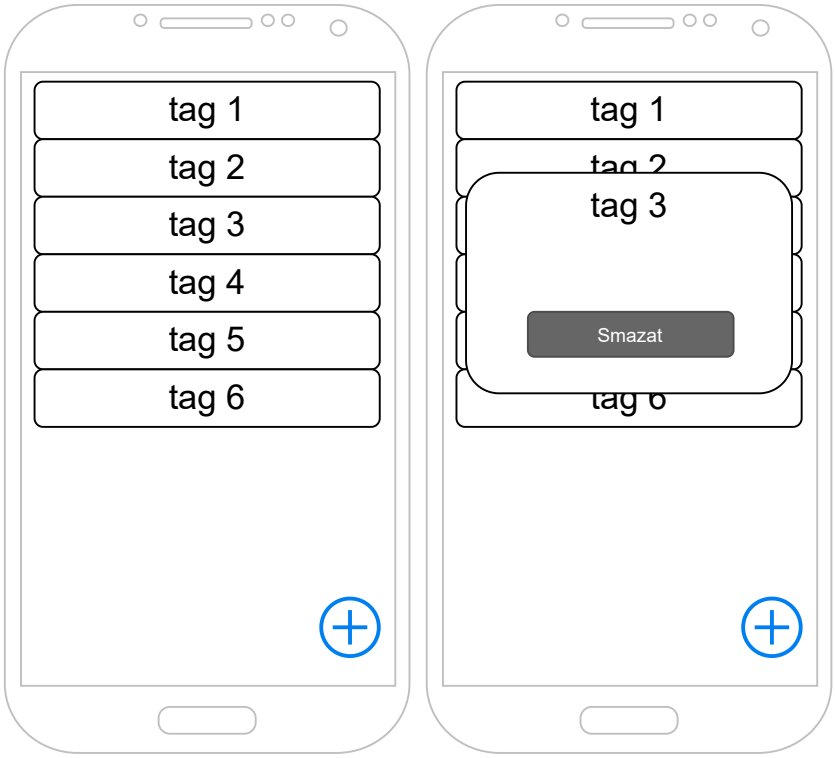
Základem velké většiny všech možných aplikací je dnes navigační rozhraní. Může jít o menu vyjíždějící ze strany nebo navigační panel na spodku nebo na vrchu obrazovky. Navigační rozhraní se používá pro orientaci mezi různými obrazovkami a funkcemi a i k tomuto účelu ho použijeme. V našem případě použijeme pro orientaci mezi obrazovkami (viz sekce 2.8.1) menu, které bude vyjíždějící ze strany obrazovky, takže nebude normálně vidět (viz Obrázek 2.22 – vlevo). V rámci prohlížení databáze tréninkových aktivit budeme chtít snadno přepínat mezi jednotlivými databázemi (cviků, tréninků a tréninkových plánů) (viz sekce 2.8.5), použijeme tedy rozhraní ve formě panelu na spodku obrazovky (viz Obrázek 2.22 – vpravo).



Obrázek 2.22: Návrh menu pro základní obrazovku vlevo a navigačního panelu pro prohlížení databáze vpravo

2.8.4 Spravování databáze tagů

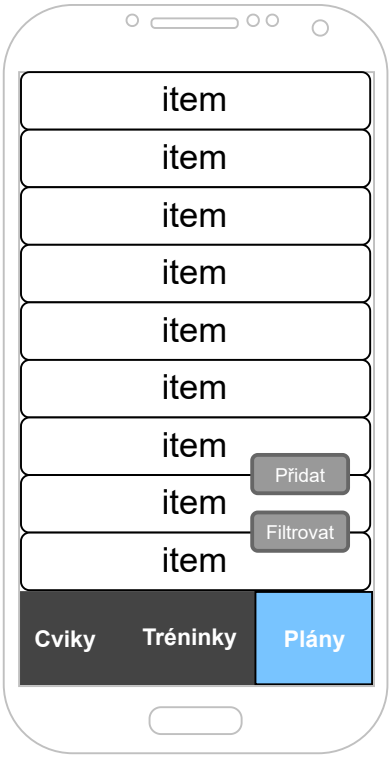
Pro obrazovku správy databáze tagů nám stačí seznam všech tagů v databázi, tlačítko pro přidání tagu (viz Obrázek 2.23) a možnost jak tag smazat. Navíc bychom chtěli uživatele upozoronit, že smazáním tagu se tag odstraní i ze všech cviků, kterým byl přiřazený (viz podkapitola 1.6 – požadavek 19.1). Toto bychom mohli provést „are you sure“ dialogem (viz sekce 2.8.2).



Obrázek 2.23: Návrh pro obrazovku správy databáze tagů

2.8.5 Prohlížení a editace databáze

Procházení databáze jsme si definovali jako samostatnou obrazovku (viz sekce 2.8.1), protože tato funkce tvoří jeden z pilířů naší aplikace a chceme ji jasně oddělit od zbytku aplikace. Databáze se skládá z databází cviků, tréninků a tréninkových plánů, tedy z pohledu uživatele se skládá ze tří seznamů. Obrazovka pro procházení těchto seznamů by měla tedy být jednoduchým seznamem položek, ale jen jednoho typu v jeden moment. K přepínání mezi těmito seznamy nám poslouží navigační panel na spodku obrazovky (viz sekce 2.8.3). Součástí grafického rozhraní by mělo být i znázornění, v jakém seznamu uživatel v daný moment listuje (tj. jaký seznam je zobrazen). K tomu stačí barevně označit dané tlačítko na navigačním panelu (viz Obrázek 2.24). K seznamu věcí a navigačnímu panelu budeme potřebovat ještě dvě další tlačítka – pro filtrování a pro přidání nové položky (viz Obrázek 2.24), abychom splnili funkční požadavky pro filtrování a přidávání tréninkových aktivit (viz podkapitola 1.6 – požadavky 1, 7, 8, 12, 13, 17).



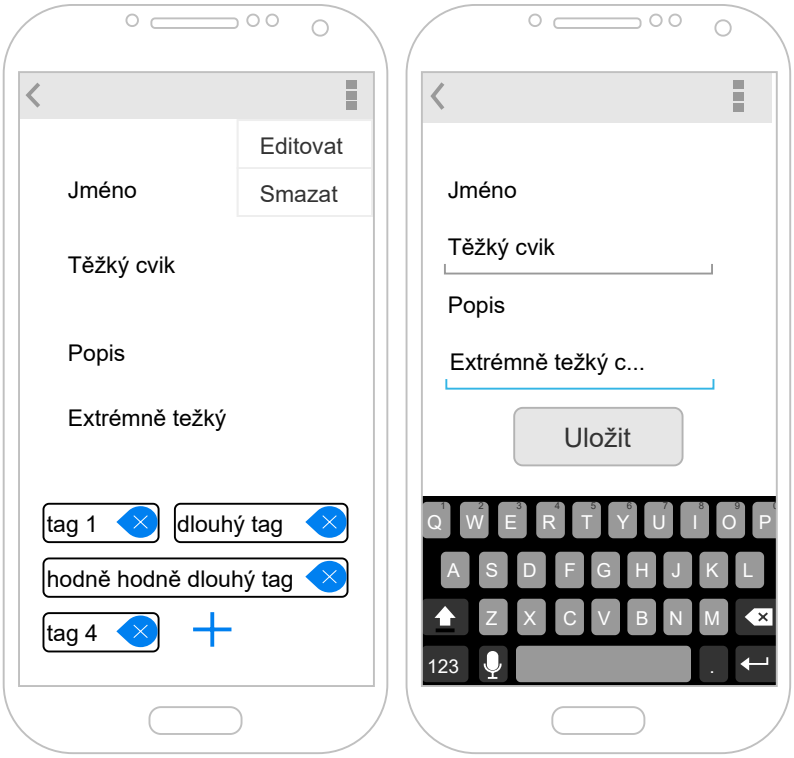
Obrázek 2.24: Návrh menu pro obrazovku procházení databáze

Detail cviku

V rámci procházení databáze potřebujeme umožnit uživateli otevřít konkrétní cvik, aby se mohl podívat na detaily a případně cvik upravit či smazat. Chceme uživateli jednoduše zobrazit textové atributy cviku a k tomu seznam všech tagů. Dále chceme uživateli umožnit editaci a smazání. Toto můžeme nabídnout uživateli skrze popup menu[4]. V režimu editace se pak uživateli umožní změnit popisek a jméno cviku. Nakonec potřebujeme seznam tagů. Chtěli bychom, aby seznam tagů byl co nejkompaktnější (viz Obrázek 2.25 – vlevo). Na příkladu máme tagy uzavřené do bublin, které se nafukují podle potřeby. Tuto funkci pro nás může udělat flexbox layout[5]. Pro každy tag bychom pak chtěli tlačítko, kterým ho okamžitě odstraníme ze cviku. Nakonec i tlačítko pro přidání nového tagu.

[4]Popup menu je menu, které se ukotví k jinému grafickému prvku v androidu a při definované akci (např. zmáčknutí tlačítka) se rozbalí – viz Xamarin Android dokumentace – Popup menu [12].

[5]Layout je obal grafických prvků na obrazovce. Flexbox layout je pak layout, který přizpůsobuje velikost míst pro grafické prvky podle potřeby – viz Google github – Flexbox layout [13].



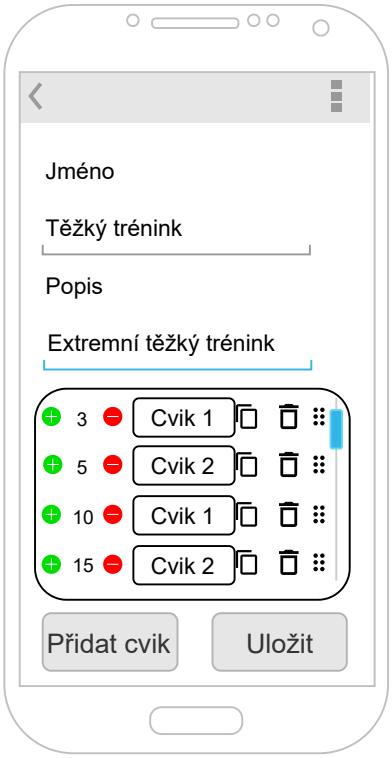
Obrázek 2.25: Návrh obrazovky pro detail a editaci cviku

Detail tréninku

Obrazovka pro detail tréninku bude v základu podobná detailu cviku (viz 2.8.5 – Detail cviku). Bude zde popup menu pro nabídku editace a smazaní tréninku a textová pole pro jméno a popisek daného tréninku. Na rozdíl od cviku zde nebudeme mít zobrazené tagy, ale cviky z daného tréninku včetně jejich četností (viz Obrázek 2.26 – vlevo). Při editaci tréninku potřebujeme uživateli umožnit nejen měnit atributy tréninku (jméno popisek), ale i počty jednotlivých cviků a jejich pořadí. Jako první intuitivní možnost grafického rozhraní pro editaci jsme navrhli a implementovali následující (viz Obrázek 2.26 – vpravo). Počet jednoho cviku bude možné měnit tlačítky plus a mínus. Pro pořadí jsme zvolili přímočaré řešení v podobě tlačítek se šipkami, které ukazují směr, kterým se daný cvik posune .

Obrázek 2.26: Návrh obrazovky pro detail a editaci tréninku

Pro editaci tréninkového plánu taktéž potřebujeme možnost měnit pořadí, ale na rozdíl od tréninku jsme se pro plán rozhodli použít systém „drag and drop“ (viz sekce 2.8.5 – Detail tréninkových plánů). Po tom co jsme implementovali první návrh pro editaci tréninku i plánu, porovnali jsme tyto dva rozdílné přístupy ke změně pořadí položek v seznamu a rozhodli se i u tréninku použít systém „drag and drop“, jakožto uživatelsky přívětivější. Šipky na změnu pořadí nahradila ikonka šesti teček (viz Obrázek 2.27 – ikonka u položky v seznamu nejvíce vpravo), která i ve spoustě jiných aplikací značí právě posouvání položky v seznamu. Nakonec jsme se ke každé položce rozhodli přidat tlačítko pro zdvojení položky (ikonka pro kopírování), které danou položku přesně zkopíruje a vloží hned za kopírovanou položku a tlačítko pro odstranění položky.

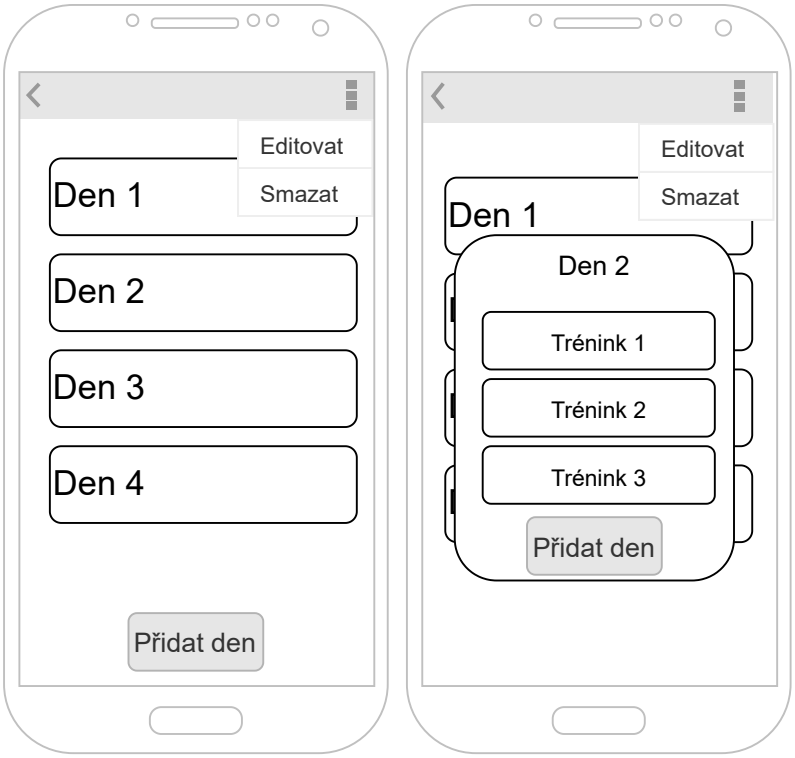


Obrázek 2.27: Nový návrh obrazovky pro editaci tréninku

Detail tréninkových plánů

Vzhledem k tomu, že tréninkový plán je pouze o tom uspořádat dny a tréninky a krom jména neobsahuje žádná další data, chtěli bychom zvolit k editaci přímočarý přístup. Popup menu pro editaci a smazání zůstane stejné, ale mód editace po kliknutí na tlačítko `editovat`, umožní pouze změnit jméno plánu, což je jediná textová položka. Plán obsahuje seznam dnů (viz sekce 2.6.5), kde každý obsahuje další seznam tréninkových záznamů (tedy sémanticky tréninků). Chceme tedy uživateli zobrazit v první řadě seznam dnů a když si na některý den klikne zobrazí se detail dne, tedy seznam tréninků daného dne. Detail dne bychom chtěli udělat jako dialog (viz sekce 2.8.2) a to jen pro to, aby to vypadalo vizuálně lépe. Detail dne se přímo týká tréninku, tedy chceme aby uživatel neztratil kontakt s obrazovkou se seznamem dnů. Pro oba seznamy nám pouze zbývá přidat tlačítko na přidání dalšího dne do plánu, respektive tréninku do daného dne (viz Obrázek 2.28).

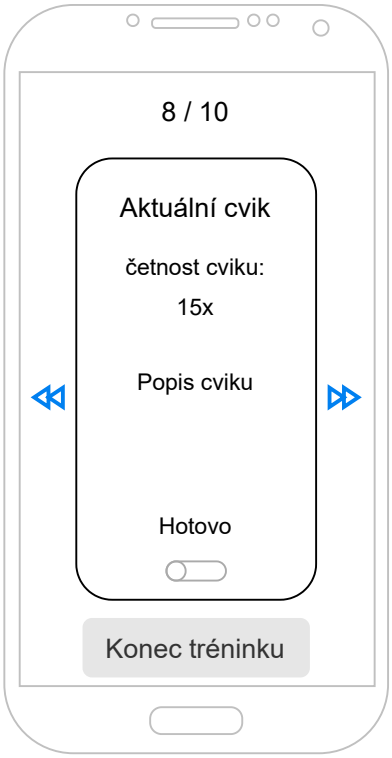
Chtěli bychom zde aby uživatel mohl „přirozeně pohybovat“ s objekty v plánu a vše se automaticky ukládalo. Výrazem „přirozeně pohybovat“ myslíme systém „drag and drop“, který je již znám z mnohých počítačových operačních systémů (viz článek Drag and drop na Wikipedii [14]), ale zároveň se uplatňuje i v moderních mobilních aplikacích. Pro odstranění dané položky bychom pak chtěli, aby uživateli stačilo přetáhnout položku do strany – tzv. „swipe“.



Obrázek 2.28: Návrh obrazovky pro detail a editaci tréninkového plánu

2.8.6 Spuštěný trénink

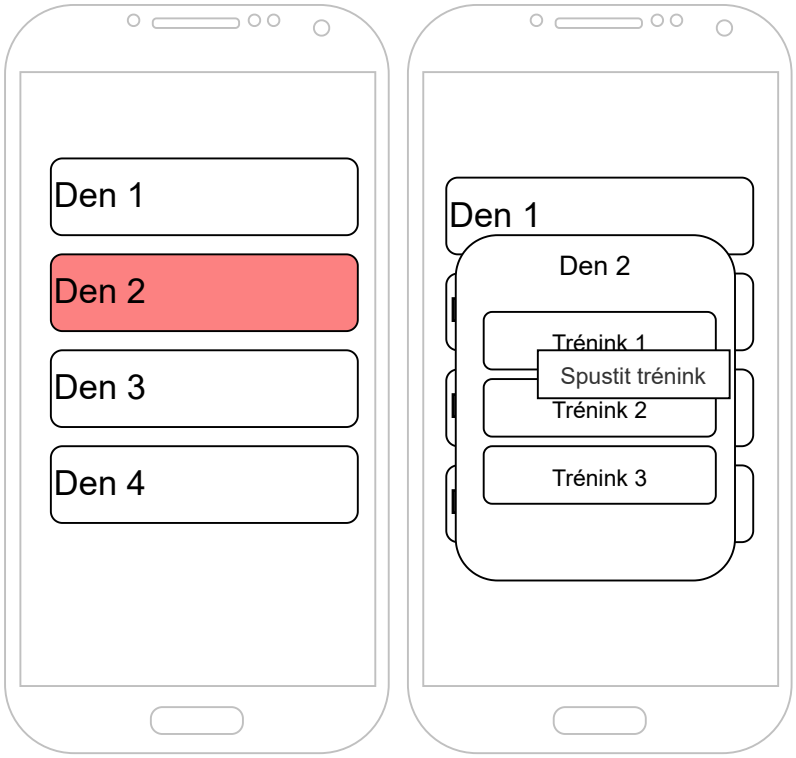
Další obrazovkou kterou jsme si definovali a nepopsali je obrazovka pro spuštěný trénink. Tato obrazovka vhledem k požadavkům (viz podkapitola 1.6 – požadavky 18.1, 18.2, 18.3) musí ukazovat detail právě cvičeného cviku v rámci spuštěného tréninku, musí označit cvik za hotový a musí uživateli umožnit procházet všechny cviky v tréninku. Obrazovka tedy potřebuje pouze textová pole pro detaily cviku, tlačítka pro posouvání se mezi cviky a tlačítko pro označení cviku za hotový. Tímto splní funkční požadavky. Dále bychom mohli chtít ukázat kolik cviků nám zbývá nebo možnost kdykoliv ukončit trénink (viz Obrázek 2.29). Nakonec bychom mohli tlačítka pro přecházení mezi cviky doplnit o možnost „swipování“, tedy přejíždění prstem vlevo, vpravo po obrazovce, pro posouvání se mezi cviky.



Obrázek 2.29: Návrh obrazovky pro spuštěný trénink

2.8.7 Spuštěný tréninkový plán

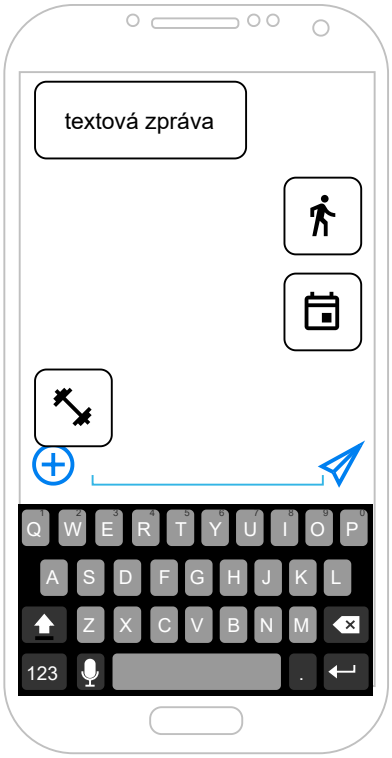
Po obrazovce spuštěného tréninkového plánu chceme, aby ukázala seznam dnů, které v plánu jsou a aby vizuálně odlišila, ve kterém dnu v plánu se nacházíme, vzhledem k času spuštění. Dále by mělo jít na daný den kliknout a zobrazit seznam tréninků pro daný den. Tréninky by měli po kliknutí nabídnout spuštění (viz sekce 2.8.6). Na příkladu (viz Obrázek 2.30) vidíme detail spuštěného tréninkového plánu, který je aktivní druhým dnem. V rámci druhého dne jsou zde naplánované tři tréninky. Obrázek vlevo pak vizualizuje situaci kdy uživatel klikl na trénink č.1 a pomocí spinneru [11] se ukázala nabídka spuštění tréninku.



Obrázek 2.30: Návrh obrazovky pro aktivní tréninkový plán

2.8.8 Chat, sdílení dat s dalšími uživateli

Pro obrazovku chatu se budeme inspirovat všemi možnými chatovacími aplikacemi. Tyto aplikace každou zprávu balí do bubliny a rozlišují odesílatele zprávy tím, na jakou stranu obrazovky bublinu se zprávou umístí. Zároveň řadí zprávy odzdola nahoru od nejnovější. Stejně tomu tak uděláme i my. V chatu samozřejmě potřebujeme textové pole pro napsání zprávy a tlačítko pro odeslání. V našem případě potřebujeme taktéž tlačítko pro odeslání tréninkové aktivity (viz Obrázek 2.31 – vlevo dole nad klávesnicí). Nakonec potřebujeme vizuálně odlišit zprávy s tréninkovými aktivitami a to i mezi nimi navzájem (viz podkapitola 1.6 – požadavek 33). Pro tuto potřebu se nejlépe hodí ikonky (viz Obrázek 2.31), které po kliknutí zobrazí detail dané aktivity (viz 2.8.2 – Dialog pro zobrazení tréninkové aktivity v chatu).



Obrázek 2.31: Návrh obrazovky pro chat

3. Analýza technických řešení

V předchozí kapitole jsme si provedli „high level“ analýzu, tedy analýzu naší aplikace jako celku a z pohledu jejích nejdůležitějších částí – datový model, databáze, grafické rozhraní. V rámci vybraných technologií a vytvořených modelů dat si v této kapitole rozebereme problémy na které jsme při implementaci narazili a museli je nějak netriviálně vyřešit.

3.1 Identifikátory

V rámci návrhu dat (viz sekce 2.6.1) jsme si vysvětlili proč u většiny dat potřebujeme identifikátory. Data, která mají identifikátor, budou mít ve Firestore databázi jiný typ identifikátorů než data v SQLite databázi, nicméně v aplikaci chceme pracovat jen s jedním modelem dat (viz podkapitola 2.6). Abychom tedy mohli používat jen jeden model dat v aplikaci, ale zároveň používat vybrané databázové technologie, potřebujeme najít způsob jak v modelu uložit identifikátor nehledě na to jestli je textový (Firestore) nebo celočíselný (SQLite). Pro tento problém použijeme čistě textové řetězce, protože číslo na text převedeme snadno, stejně tak ho bez problémů naparsujeme zpět na číslo. Parsování čísla na text a zpět je sice poněkud neefektivní, ale data se kterými se v jeden moment bude pracovat, pravděpodobně nebudou nijak velká. Vzhledem k sémantice dat (tréninky, cviky apod.), půjde pravděpodobně o desítky až malé stovky záznamů, pro které toto bude třeba udělat. V kontextu aplikace to není znatelné zpomalení, to aplikaci spíše zpomalí operace nad databázemi.

3.2 Cachování tagů

Jediná entita, které se přiřazuje tag, je cvik. Nicméně podle funkčních požadavků je potřeba moci třídit podle tagů i tréninky a tréninkové plány (viz podkapitola 1.6 – požadavky 12, 17). Množina tagů, podle které se tyto aktivity třídí vznikne sjednocením množin tagů ze všech cviků, které dané aktivity obsahují. Při načítaní z databáze nedává smysl tato sjednocení počítat, protože je často nemusíme ani použít. Množiny tagů se tedy spočítají až když je budeme potřebovat, tedy když budeme chtít tyto aktivity filtrovat. Předpokládáme, že se může snadno stát, že se uživatel ve filtrování splete a vybere jinou množinu tagů pro filtrování. V tento moment bychom množinu tagů museli počítat znova. Tento problém lze snadno vyřešit cachovaním tagů, tedy uložením tagů v dané entitě, při jejich prvním výpočtu. Při pozdějších filtrováních bychom už nemuseli toto sjednocení znovu počítat, tedy bychom ušetřili výpočetní čas. Daní za tuto cache je samozřejmě větší spotřeba operační paměti, nicméně v C# budeme mít tagy jako třídu, tedy referenční typ. Samotný tag bude ležet na haldě, protože bude již načtený pro cviky. Pro každou instanci tréninku či plánu můžeme čekat desítky až menší stovky tagů, ale spíše méně. Lze totiž předpokládat, že uživatel bude k sobě do tréninků a plánů sdružovat cviky, které mají něco společného. Kdyby tedy měla každá aktivita sto referencí na tag, bude spotřeba paměti zhruba |počet aktivit| * 100 * |reference|, tedy paměťová složitost těchto cachí je O(n), kde

n je počet záznamů dané aktivity. Reference bude pravděpodobně velká 4 nebo 8 bytů, podle velikosti adresového prostoru. Konstanta zde tedy bude přibližně 400 – 800. Stále zde můžeme mluvit o lineární paměťové složitosti. Tedy i například při řádově stovkách aktivit, budeme mít spotřebu paměti větší o řádově desítky kB, což je v kontextu velikostí dnešních pamětí, které jsou řádově jednotky GB, v pořádku.

3.3 Zacházení se smazanými odkazy

Když uživatel v aplikaci něco smaže, potřebujeme, aby se tato změna provedla i v databázi. V aplikaci ovšem nejdříve aktualizujeme daný model, aby se uživateli zobrazila daná změna. V případě smazání cviku z tréninku, smazání dne z plánu a smazání tréninku ze dne v tréninkovém plánu tedy provedeme odebrání dané entity ze seznamu. Pro aktualizaci dat v databázi si ovšem potřebujeme pamatovat, který záznam jsme smazali. Konkrétní řešení v těchto případech si nyní ukážeme.

Smazané odkazy na cviky v tréninku

Součástí entity `ExerciseInTrainingRecord` je i počet, kolikrát má být daný cvik vykonán. Pro řešení našeho problému nám stačí nastavit pro smazání tento počet na nula, ale v seznamu ho necháme. Pro uživatele se pak můžeme rozhodnout, zda-li záznamy s počtem nula zobrazíme nebo ne. Každopádně je to jasný ukazatel smazání. Když danou instanci tréninku předá aplikace databázi k updatu, databáze smaže každý záznam kde je počet cviků nula a vrátí aktualizovanou instanci, kde tyto záznamy s nulovým počtem už nebudou.

Smazané odkazy na tréninky a dny v tréninkovém plánu

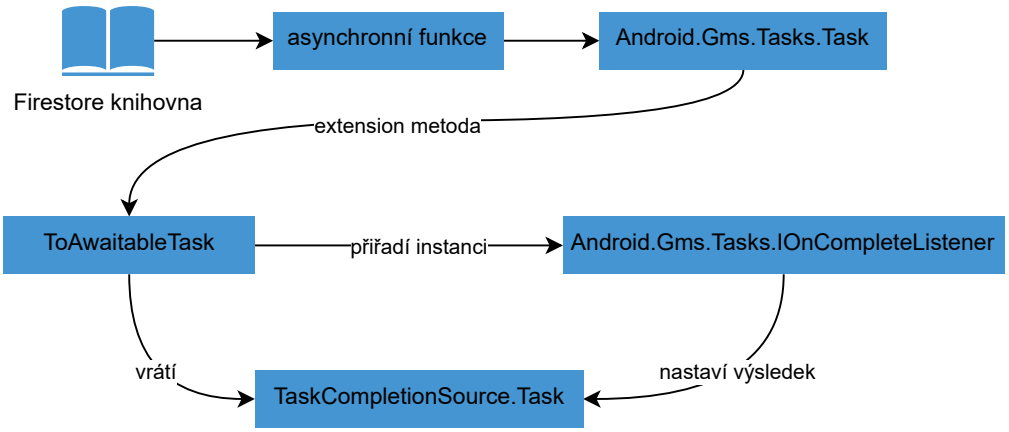
V entitách `PlanDay` a `TrainingInDayRecord` nemáme žádný ukazatel počtu dané entity. Řešení je zde, ale také jednoduché. Do odpovídajících tříd přidáme položky `deletedDays`, respektive `deletedTrainings`, do kterých budeme odkládat smazané instance. Pak je již postup stejný jako v případě smazaného cviku v tréninku. Databáze, když dostane danou instanci, aktualizuje záznamy v databázi a zároveň aktualizuje danou instanci, tedy vyčistí dané seznamy smazaných položek.

3.4 Databáze Firestore

Abychom mohli používat Firestore databázi, potřebujeme se na ni mít jak připojit. Od Googlu existují knihovny pro .NET, pomocí kterých se lze na Firestore připojit. Bohužel podle jejich dokumentace nepodporují Xamarin [15]. Naštěstí pro Xamarin Android existují knihovny přímo od Xamarinu (Microsoftu) viz nuget balíček [16]. Tato knihovna nám přináší bohužel další problém. V rámci komunikaci s databází, tedy komunikaci po síti, knihovna využívá `Tasky`, ale bohužel tasky pro Android (viz Google Play Services [17]). Tyto `Tasky` jsou velmi podobné `Taskům` ze C#, ale navzájem kompatibilní nejsou. Zároveň nenabízí podobná klíčová slova jako `await, async` a tedy ani stejné použití, místo

toho nabízí přiřazení `listeneru`, konkrétně instance třídy implementující rozhraní `IOnCompleteListener`.

Přiřazení listeneru se využívá v článku *Awaiting Google Play services tasks* [18], po jehož vzoru můžeme Android tasky převést na klasické C# tasky. Budeme chtít vytvořit `listener`, který nám nastaví výsledek `Tasku` pro C# po dokončení `Android Tasku`. Jak je znázorněno na schématu (viz Obrázek 3.1), asynchronní funkce z knihovny pro Firestore, nám na požadavek do databáze vrátí instanci `Android.Gms.Tasks.Task`. Na tomto tasku zavoláme extension metodu, která tasku přiřadí instanci třídy implementující `IOnCompleteListener`. Pro tento účel vytvoříme třídu implementující dané rozhraní a přijímající v konstruktoru instanci `TaskCompletionSource` [19]. `TaskCompletionSource` se v C# používá pro kontrolu výsledků klasického C# `Tasku`. Naše extension metoda tedy Android tasku přiřadí listener a vrátí `TaskCompletionSource.task`, tedy C# task. Ve chvíli kdy se dokončí daná asynchronní operace, spustí se kód listeneru, který nastaví výsledek tasku (právě pomocí třídy `TaskCompletionSource`), který již vrátila naše extension metoda. V aplikaci nám toto tedy umožní používat Firestore databázi, pomocí knihovny od Xamarinu, ve spojení se C# `Tasky` (viz podkapitola 3.5).



Obrázek 3.1: Schéma převodu Android tasku na C# task

3.5 Asynchronní programování

Pro libovolné aplikace které mají grafické rozhraní platí, že uživatel nechce, aby aplikace „zamrzla“, byť jen na moment. Takové zamrznutí se v aplikaci může stát kdykoliv, kdy aplikace potřebuje udělat nějakou operaci, která zabere čas dost dlouhý na to, aby si toho uživatel byl schopný všimnout. Může jít o netriviální výpočet nebo například o komunikaci na síti, která také může trvat déle. V našem případě naplňujeme oba tyto příklady. Jednak komunikujeme s online databází po síti a zadruhé za netriviální výpočet můžeme považovat například komunikaci s offline databází. Například můžeme mít v aplikaci tlačítko „ukázat tréninky“, po jehož zmáčknutí se mají z databáze načíst všechny tréninky. Kdybychom měli obsluhu tlačítka plně synchronní jako je na příkladu níže, aplikace by nám zamrzla po dobu co by trvalo načíst a zobrazit tréninky. Když se zmáčkne tlačítko, tak UI

vlákno[1] začne vykonávat danou funkci. Ve funkci se ovšem zanořujeme do další funkce pro nahrání tréninků z databáze, kterou by v tomto případě také vykonalo UI vlákno. A protože UI vlákno vykonává operaci v databázi, grafické rozhraní nám zamrzne. Totiž vlákno určené k tomu, aby grafické rozhraní obsluhovalo, vykonává jinou práci.

Špatný příklad – zamrznutí:

```
tlacitkoUkazatTreninky.Click += NahrajTreninky;
...
void NahrajTreninky(){
    //synchronní volání do databáze
    treninky = databaze.DostatTreninky();

    //zobrazení tréninků na obrazovku
    obrazovka.ZobrazSeznamTreninku(treninky);
}
```

Abychom byli schopni vyhnout se zamrznutí aplikace, nabízí se multi-vláknové programování, kdy jedno vlákno obsluhuje grafické rozhraní a druhé počítá danou operaci. Pro náš případ se ovšem více hodí koncept `Tasků` [20].

`Tasky` a asynchronní funkce nám umožňují rozdělit program na více úkolů, které se mohou vykonávat paralelně (neznamená to že se budou vykonávat paralelně) a my chceme abychom z nich měli alespoň pocit paralelního vykonávání. Princip asynchronní funkce tkví v tom, že ve chvíli kdy potřebuje počkat, než se vykoná jiná operace, uvolní daná funkce svoje prostředky, tedy vlákno. Vlákno dostane později přiřazené od `thread poolu`, když se operace na kterou čeká, dokončí. Například v naší aplikaci teoreticky budeme mít tlačítko „ukázat tréninky“ – viz příklad výše. Po zmáčknutí naše UI vlákno udělá volání do databáze. To může potenciálně trvat déle a my nechceme aby volání do databáze blokovalo UI vlákno, aby nám nezamrzla obrazovka. Použijeme tedy asynchronní metodu pro načtení dat z databáze – viz Správný příklad. UI vlákno zavolá tuto metodu, ve které potřebuje **asynchronně** počkat na výsledek dotazu z databáze. Toto čekání se v C# uskuteční pomocí klíčového slova `await`, které zde zároveň definuje asynchronní metodu. Tím se uvolní dané vlákno, v našem případě UI vlákno. To se tedy může vrátit k obsluze uživatelského rozhraní. Když se operace v databázi dokončí, přiřadí se jí z `thread poolu` jiné vlákno a dokončí se daná operace, tedy se seznam tréninků zobrazí na obrazovku.

Spávný příklad – bez zamrznutí:

```
tlacitkoUkazatTreninky.Click += NahrajTreninkyAsync;
...
//po zavolání této funkce se vrátí Task
Task NahrajTreninkyAsync(){
    //asynchronní volání do databáze
    //výsledkem asynchronního volání nejsou tréninky ale Task
    task = databaze.DostatTreninkyAsync();

    //funkce potřebuje počkat na výsledek dotazu do databáze
    //nechce ale blokovat UI vlákno
    //počká na výsledek tasku, což je seznam tréninků
```

[1] vlákno obsluhující uživatelské rozhraní – tedy odpovědné za responzivitu uživatelského rozhraní

```
12      //protože je asynchronní, vzdá se zde svého vlákna, tedy UI vlákna
13      //UI vlákno dále může obsluhovat grafické rozhraní
14      treninky = await task;
15
16      //když se task vykoná, funkce dostane
17      //od thread poolu vlákno aby mohla pokračovat
18
19      //zobrazení tréninků na obrazovku
20      obrazovka.ZobrazSeznamTreninku(treninky);
21  }
```

Konkrétně v našem případě jsme si již ukázali, jak Firestore databázi používat pomocí C# `Tasků` (viz podkapitola 3.4) a zároveň i námi zvolená knihovna pro SQLite – sqlite-net-pcl – nabízí API pro asynchronní programování pomocí C# `Tasků`. Tím, že obě databáze můžeme ovládat pomocí `Tasků` můžeme pro ně jednoduše udělat společné asynchronní rozhraní (viz podkapitola 3.6). Zároveň uvnitř tříd ovládající databáze můžeme používat `Tasky` například pro řetězení dotazů do databází. Tedy znovu práci rozložit na více úkonů, které se mohou vykonávat paralelně. Takový příklad může být při načítání jednoho tréninku – viz příklad. Potřebujeme načíst z jedné tabulky model tréninku a z druhé tabulky záznamy o cvicích. Tyto věci se mohou dít paralelně, pouze přiřazení cviků k tréninkovému modelu musí proběhnout až když máme oba výsledky.

```
1   Task NactiTreninkAsync(treninkId){
2       //zadám práci pro načtení tréninku
3       task1 = NactiTreninkZTabulkyAsync(treninkId);
4
5       //zadám práci pro načtení cviků
6       task2 = NactiZaznamyOCvicichZTabulkyAsync(treninkId);
7
8       //počkám až dostanu model tréninku
9       trenink = await task1;
10
11      //počkám až dostanu cviky a přiřadím
12      trenink.seznamCviku = await task2;
13  }
```

3.6 Použití databází

Pro naši aplikaci jsme si vytvořili datový model (viz podkapitola 2.6), pro databáze jsme si určili knihovny, které použijeme na jejich obsluhu (viz podkapitoly 2.4, 3.4) a nyní je potřeba vyřešit problém, kdy jakou z databází použít a jak zajistit abychom vždy dostali z databází náš datový model. Z funkčních požadavků 24 a 25 (viz podkapitola 1.6) nám vychází, že pro přihlášené uživatele je potřeba použít online databáze tedy Firestore, naopak z požadavku 20 nám vychází použití offline databáze, tedy SQLite. Rozhodnutí, kterou databázi použít tedy závisí na tom, zda-li je uživatel naší aplikace na daném zařízení přihlášený či nikoliv.

První přímočaré řešení, které by nás mohlo napadnout je, že v místech programu, kde je potřeba načíst libovolná data z nějaké databáze, se prostě rozhodne

jestli je uživatel přihlášený a podle toho se použije instance té dané databáze. Tohle řešení nám přinese zásadní problém v možnosti jakkoliv měnit či rozšiřovat stávající implementaci. Stačí aby se nám změnila funkce, pomocí které zjišťujeme, zda-li je uživatel přihlášený a najednou se musí změnit použití této funkce všude, kde se používá databáze. Stejně tak tomu bude, když bychom chtěli jakkoliv upravit použití dané databáze. Najednou bychom museli její používání měnit všude, kde databázi používáme.

K tomu abychom se zbavili tohoto problému a udělali kód udržitelnější a rozšiřitelnější, potřebujeme extrahovat rozhodování o tom, který typ databáze použít, na jedno místo. Zde nám poslouží návrhový vzor `Factory`. Factory patttern nám nejen extrahuje odpovědnost rozhodování, ale umožní například jednotnou parametrizaci, na základě které vyrobí instanci daného typu databáze. Například když budeme chtít splnit funkční požadavky 26, 27 (viz podkapitola 1.6), budeme potřebovat ve stejnou chvíli přistup do offline databáze i online databáze. Vytvoříme tedy třídu `DatabaseFactory` a funkci, která nám bude vracet potřebný typ. K tomu aby nám facotry mohla vrátit jednotný typ potřebujeme třídy pro databáze spojit společným předkem nebo rozhraním (rozhraním). V našem případě se hodí jednoznačně více společné rozhraní, protože nepotřebujeme, žádnou defaultní implementaci, ani nějaká společná data, vzhledem k tomu, že jde i o úplně odlišné technologie ani žádné společné metody a zároveň i významově nám jde jen o kontrakt, kterému obě databázové třídy musí dostát.

Nakonec je třeba se ještě rozhodnout jestli vytvoříme jednu velkou třídu pro ovládání celé databáze nebo raději rozdělíme toto ovládání databáze do více tříd. Pokud bychom měli jednu velkou třídu pro online databázi a jednu pro offline databázi, museli by obě tyto třídy implementovat jedno společné rozhraní (viz výše). V tomto rozhraní bychom jistě potřebovali i metody pro načítání dat o chatu a registrovaných uživatelích, to ale dává smysl jen v případě online databáze, nicméně z principu rozhraní by tyto metody musela implementovat i třída pro offline databázi. Otázkou by zde bylo co by v offline databázi měli tyto funkce dělat. V tomto případě by tedy bylo lepší rozdělit databázové třídy podle toho, za co nesou odpovědnost. Takové řešení i více odpovídá dnešní snaze mít jednotlivé komponenty softwaru méně provázané (viz Loose coupling – Wikipedia [21]). Budeme mít tedy samostatné třídy a tedy i rozhraní pro následující části databází:

- tagy
- cviky
- tréninky
- tréninkové plány
- chat
- uživatele
- aktivní tréninkové plány

3.7 Import dat mezi databázemi

Podle funkčních požadavků uživatel naší aplikace musí být schopný importovat data z jeho lokální databáze do online databáze (viz podkapitola 1.6 – požadavky 26, 27). Navíc v rámci chatu, když uživatel pošle libovolnou tréninkovou aktivitu, budeme potřebovat importovat tuto aktivitu z uživatelovy databáze do databáze daného chatu (viz sekce 2.6.8 – Chat ve Firestoru). Naopak když si uživatel z chatu bude chtít tréninkové aktivity uložit k sobě do databáze, budeme potřebovat importovat tyto aktivity z databáze chatu do uživatelovy databáze. Nejenže tyto databáze (databáze chatu a tréninkových aktivit) mají různé rozhraní (viz podkapitola 3.6), ale ve všech zmíněných případech je potřeba projít danou tréninkovou aktivitu do hloubky a opravit v ní reference vzhledem k novým identifikátorům v druhé databázi (viz sekce 2.6.1).

Jako první přímočaré řešení se zde nabízí, provést tuto opravu referencí v místě, kde to zrovna potřebujeme. Podobně jako v případě výběru vhodné databáze, to není vhodné řešení. Jak jsme nastínili, import dat se nám objeví na vícero místech. Představme si tedy, že nám v pozdějším vývoji změní jakkoliv struktura dat. Museli bychom projít celý algoritmus opravy referencí na každém místě kde se nachází. Pokusíme se tedy celý algoritmus extrahovat do samostatné třídy.

Vytvoříme třídu `ImportHelper`, která tento import dat bude provádět. Jak jsme již zmínili například rozhraní pro chat a rozhraní pro ostatní databáze tréninkových aktivit budou jistě různé, tedy nelze jednoduše dodat naší třídě např. dvě různé instance databází pro tréninky. S tímto problémem, nám mohou pomoci `delegáti` [22]. Nedodáme naší třídě entitu pro ukládání, nýbrž odkaz na funkci, pomocí které víme, že daná entita uložit lze. Kdybychom v místě potřeby importu dat měli funkci, která ne zcela odpovídá definovanému delegátovi v `ImportHelperu`, pomůžeme si `lamda výrazy` [23], s jejichž pomocí můžeme vytvořit delegáta stejného typu jako to definuje `ImportHelper`.

3.8 Správa uživatelů

Jeden z argumentů, proč jsme si pro online databázi zvolili právě Firebase, byla zabudovaná správa uživatelů. Tato správa uživatelů poskytuje vše od přihlašování, odhlašování, registraci až po udržovaní přihlášeného uživatele, zasílání ověřovacích emailů a možností přihlášení pomocí jiné služby. Potřebujeme ovšem znovu knihovnu abychom tyto služby mohli používat. Existuje, podobně jako pro databázi (viz podkapitola 3.4), knihovna od Xamarinu přímo pro Android – `Xamarin.Firebase.Auth` [24].

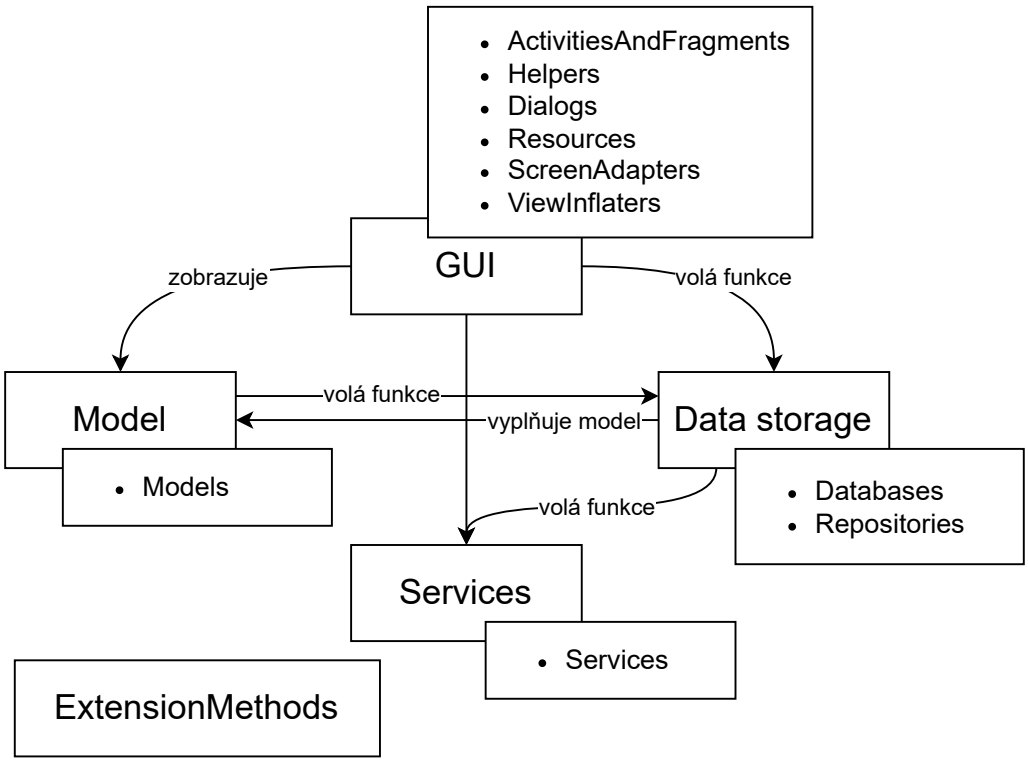
Vytvoříme si pro tuto knihovnu wrapper [8], který si uzpůsobíme tak, aby nám dodával ty funkce které budeme potřebovat a hlavně aby se nám staral o výjimky vyhozené z knihovny.

4. Implementační dokumentace

V této kapitole si popíšeme již hotovou aplikaci z pohledu implementace. Budeme se postupně zanořovat do větších a větších detailů implementace, aby čtenář nahlížející do kódu věděl, jak a kudy proudí data, jak které prvky navzájem komunikují apod.

Aplikace je implementována v C# za použití Visual Studia 2019, celá práce je tedy solution pro Visual Studio. V rámci solution je aplikace v jediném projektu. V projektu jsou zdrojové soubory rozděleny do složek podle implementovaní funkcionality. Například zdrojové kódy pro databáze jsou ve složce `Databases`. Každá z těchto složek, spadá do jednoho ze čtyř celků, ze kterých se aplikace skládá (viz Obrázek4.1):

- Model dat,
- GUI – grafické uživatelské rozhraní,
- Data storage – vše co se týká databází,
- Services.



Obrázek 4.1:
Schéma top level celků a seznamy složek spadajících do jednotlivých celků

Model dat jsou převážně datonosné třídy a doplňkové funkce pro získání dalších detailů, které v dané třídě nejsou přímo uloženy. Tyto funkce v modelech využívají databáze (viz podkapitola 4.1). Model dat je pak používán v GUI, kde je zobrazován a v databázích, kde je naplněn daty.

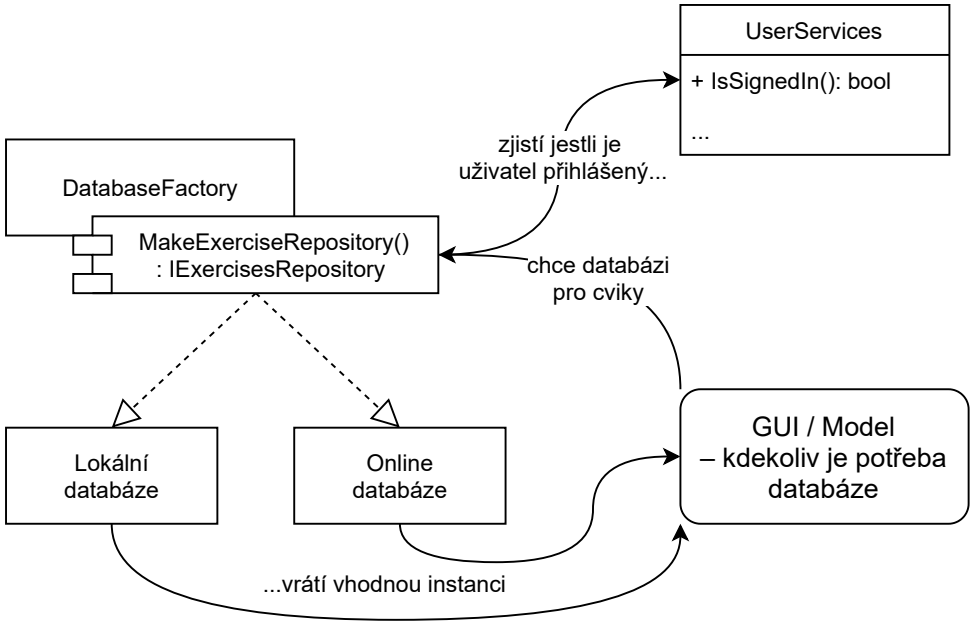
`Data storage` obsahuje jednotná rozhraní pro databáze, která se jednotně nazývají `Repositories` a jsou umístěna ve stejnojmenné složce. Ve zbytku aplikace se vždy pracuje s instancí třídy implementující tato rozhraní. Jejich jména tvoří předpona „I“ + název modelu + „Repository“. Například rozhraní pro databázi chatu se jmenuje `IChatRepository`. Dále jsou zde jednotlivé implementace databází a pomocné třídy pro fungování databází, například třída `DatabaseFactory`, která nám vytváří instanci správné databáze. Ta rozhoduje podle toho zda-li, je uživatel přihlášený (viz sekce 4.2.5), proto `data storage` využívá `services`.

Do `services` spadají služby tykající se business logiky aplikace nebo i běhu aplikace. Nachází se zde zatím jedna důležitá služba – `UserServices`, která zajišťuje komunikaci s Firebase, co se týče uživatele, tedy přihlašovaní apod.

Vše co se týká jakkoliv zobrazování dat nebo interagování uživatele s aplikací, považujeme za grafické uživatelské rozhraní – GUI. Zdrojové soubory (viz seznam na Obrázku 4.1), jsou zde rozděleny podle implementovaného způsobu zobrazování nebo podle účelu. `ActivitiesAndFragments` je složka s Android aktivitami a fragmenty, `Helpers` obsahují pomocné třídy k zobrazovaní, `Dialogs` obsahují grafické dialogy vyrobené na míru, složka `Resources` obsahuje zdrojové kódy v `xml` pro Android design – návrh obrazovek, obrázky, definici všech menu aj., složka `ScreenAdapters` obsahuje ovládání pro zobrazování položek v seznamech a nakonec složka `ViewInflaters` obsahuje třídy k dynamickému vytváření grafických prvků na míru. Uživatelské rozhraní přitom používá všechny ostatní celky. GUI používá modely dat, které zobrazuje uživateli a dovoluje je uživateli upravit. K načtení dat do modelů nebo naopak uložení GUI používá databáze a nakonec používá i jedinou službu ze `Services` – `UserServices`, aby uživateli zprostředkovalo možnost přihlášení, odhlášení a registrace.

Mimo všechny čtyři celky stojí ještě složka `ExtensionMethods` s jedinou statickou třídou `Extensions` obsahující extension metody pro třídy, které nemají původ v naší aplikaci, ale jsou standardní C# třídy nebo jsou z nějaké z knihoven použitých v aplikaci.

Už víme, že GUI a model používají databáze. Jak jsme zmínili, součástí `Data storage` je i třída `DatabaseFactory`, která vrací vhodnou instanci databáze. Je třeba dodržet, že kdykoliv je v aplikaci potřeba databáze, necháme vytvořit její instanci právě `DatabaseFactory`. Na schématu (viz Obrázek 4.2) můžeme vidět, jak probíhá rozhodovací proces `DatabaseFactory` pro to, jakou instanci v místě potřeby dostaneme. Když např. GUI potřebuje databázi, vytvoří si instanci `DatabaseFactory` a zavolá její factory metodu podle toho, který typ databáze potřebuje. Na schématu je znázorněna metoda `MakeExerciseRepository`. `DatabaseFactory` si zjistí jestli je v aplikaci uživatel přihlášený a podle toho se rozhodne jakou instanci databáze vrátí.

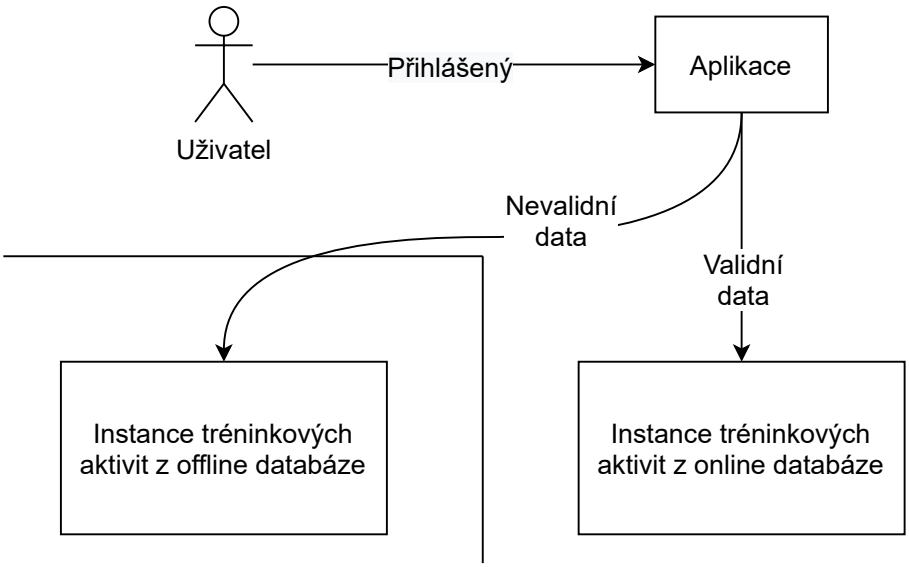


Obrázek 4.2: Schéma práce DatabaseFactory

4.1 Modely

Modely dat jsou v naší aplikaci asi nejpřímočařejším a nejjednodušším celkem. Jednotlivé modely dat jsme si v úvodní kapitole popsali z pohledu uživatele (viz sekce 1.4.1) a v další kapitole jsme si vytvořili modely tak, aby s nimi aplikace mohla pracovat a aby odpovídaly uživatelskému modelu (viz podkapitola 2.6). Z pohledu implementace jde o mapování 1:1 jednoho modelu na jednu C# třídu. Třídy jsou primárně datonosné, mají tedy především veřejné atributy. Třídy `Training, Plan, PlanDay, TrainingInDayRecord` mají doplňkové funkce/atributy, které počítají doplňkové informace o daném modelu, které nejsou součástí modelu. Některé z těchto metod počítají výsledky na základě dat která model obsahuje, ostatní metody se dotazují na další data do databáze.

V celé aplikaci se předpokládá, že aplikace funguje buď v offline režimu, tedy když je uživatel odhlášený, nebo v online režimu, když je uživatel přihlášený. Režim přímo ovlivňuje, jaká databáze se použije pro práci s daty. V offline režimu se používá lokální SQLite databáze, zatímco online se používá Firestore databáze. To která databáze se použije zase přímo ovlivní model. Data mezi jednotlivými typy databází nejsou kompatibilní. Model je tedy striktně spjatý s režimem aplikace. Metody na modelech, které načítají další data z databáze, se tedy mohou volat jen v tom režimu, ve kterém byl daný model načtený. Tedy po přepnutí režimu aplikace se data načtená z odpovídající databáze stanou nevalidní (viz Obrázek 4.3). Jediná možnost kdy se data z jiné databáze (vůči režimu databáze) používají, je import data z offline databáze do online, kdy se pouze kopírují dané záznamy. Obecně na modelech platí, že všechny funkce vykonávají asynchronní operace, tzn. dotaz do databáze. Počítání dodatečných dat na základě dat v modelu je pak implementováno pomocí C# vlastností s pouze metodou `get`.



Obrázek 4.3: Schéma validity dat

Training

Při načítání všech cviků pro jeden daný trénink, nechceme načítat některé cviky dvakrát. Chceme načíst jen množinu cviků (bez duplicit) a k tomu potřebujeme funkci pro načtení množiny identifikátorů všech obsažených cviků. Tato funkce vychází přímo z dat uložených v modelu a vždy se počítá znovu. Konkrétně je toto implementováno pomocí vlastnosti `ExercisesIds` (viz kód níže). Pro filtrování tréninků podle tagů potřebujeme načíst množinu všech tagů daného tréninku. Tagy tréninku jsou určeny sjednocením tagů ze všech cviků, které trénink obsahuje. Trénink má ale pouze seznam `ExerciseInTrainingRecord` a nikoliv cviků (`Exercise`). Potřebujeme tedy nejdříve načíst cviky z databáze abychom dostali tagy. K tomu slouží funkce `GetTagsAsync` (viz kód níže), která načte cviky z databáze a extrahuje z nich jejich tagy. Protože funkce pracuje s databází, musí být asynchronní.

```
1 public List<string> ExercisesIds { get{ ... }}
2 public async Task<List<Tag>> GetTagsAsync();
```

Plan

Stejně jako u tréninku i u plánu potřebujeme způsob, jak získat seznam tagů, abychom mohli plány podle tagů filtrovat. Máme zde tedy také asynchronní funkci pro získání tagů – `GetTagsAsync` (viz kód níže). Dále tréninkový plán obsahuje seznam dnů (`PlanDay`) a dny obsahují seznam záznamů o trénincích (`TrainingInDayRecord`) – tedy obsahují jen odkazy na tréninky. Už jen pro načtení jména tréninku, potřebujeme získat seznam tréninků, tedy dotázat se do databáze, proto i zde je funkce `GetTrainingsAsync` asynchronní. Mimo jiné i funkce `GetTrainingsAsync` potřebuje množinu identifikátorů všech tréninků z plánu a k tomu je zde vlastnost `TrainingIds`, která při každém zavolání projde dny v plánu a vytvoří danou množinu identifikátorů.

```
public async Task<List<Training>> GetTrainingsAsync();
public async Task<List<Tag>> GetTagsAsync();
public HashSet<string> TrainingIds { get {...} }
```

PlanDay

Podobně jako u tréninku počítáme seznam identifikátorů cviků, i zde potřebujeme mít možnost spočítat si množinu identifikátorů tréninků obsažených v dnu, kvůli načítání tréninků samotných. Proto i zde je vlastnost `TrainingsIds` (viz kód níže), která při každém zavolání počítá znova množinu identifikátorů z tréninků, s tím rozdílem, že zde se počítá i jejich četnost. Součástí třídy `PlanDay` je i funkce `GetTraininsAsync` (viz kód níže), která z databáze může načíst tréninky pro daný den plánu, aniž bychom museli načítat všechny tréninky daného plánu.

```
public Dictionary<string, int> TrainingsIds { get { ... }}
public async Task<List<Training>> GetTrainingsAsync();
```

TrainingInDayRecord

Nakonec i pro samotný `TrainingInDayRecord` máme funkci pro asynchronní načtení tréninku z databáze – `GetTrainingAsync` (viz kód níže). Funkce načte právě ten trénink, na který se instance `TrainingInDayRecord` odkazuje. Tato funkce má využití například při spouštění tréninku z funkce aktivního tréninkového plánu.

```
public async Task<Training> GetTrainingAsync();
```

Ostatní

Zbylé modely jsou pouze datonosné a tedy obsahují pouze vlastnosti nebo datová pole, která kopírují návrh daného modelu. Neobsahují žádné funkce či vlastnosti, které by v modelu něco měnili nebo počítali další informace. Tyto třídy mají všechny položky nastavené jako veřejné – `public`. Těmito třídami jsou:

- `ActivePlan`,
- `ChatModel`,
- `ExerciseInTrainingRecord`,
- `Exercise`,
- `MessageModel`,
- `PersonModel`,
- `Tag`.

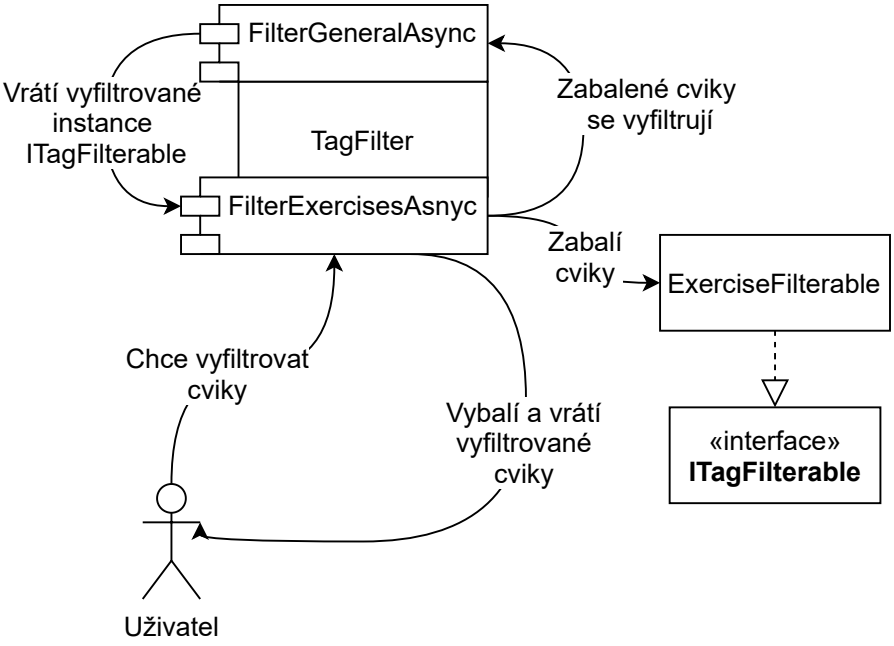
4.1.1 TagFilter

Pro aplikaci jsme vytvořili ještě jednu datovou třídu pro filtrování podle tagů – `TagFilter`. Data z této třídy se neukládají do žádné databáze, slouží pouze pro potřeby filtrování. Konkrétně jde o třídu se dvěma atributy – bool atribut `OrFilter` říkající zda-li jde o konjunktivní filtr nebo o disjunktivní filtr a atribut `tags`, což je seznam tagů pro filtrování. To jsou totiž dva vstupy od uživatele v případě, že chce seznam nějaké tréninkové aktivity filtrovat. Třída `TagFilter` pak nabízí jednu veřejnou specifickou filtrovací metodu pro každou tréninkovou aktivitu:

- `FilterExercisesAsync`,
- `FilterTrainingsAsync`,
- `FilterPlansAsync`.

Protože je ale sémantika filtrování pro všechny tréninkové aktivity stejná, tak abychom neimplementovali filtrování třikrát (pro každou aktivitu znovu), používáme zde rozhraní `ITagFilterable`, které má pouze metodu `GetTagsAsync` pro získání tagů. Na schématu (viz Obrázek 4.4) je vidět proces filtrování seznamu tréninkové aktivity (konkrétně cviků). Uživatel si vytvoří filtr a následně se na jeho instanci filtru zavolá specifická metoda pro filtrování dané aktivity. Při zavolání filtrování se tréninkové aktivity zabalí do tříd implementující rozhraní `ITagFilterable`. Univerzální funkce `FilterGeneralAsync` pak použije funkci definovanou v rozhraní `ITagFilterable` pro získání seznamu tagů a vyfiltruje seznam tréninkových aktivit, bez rozdílu o jakou aktivitu jde. Při návratu se tréninkové aktivity ve své specifické funkci zase rozbalí na původní modely.

```
1  public class TagFilter
2  {
3      public List<Tag> tags = new List<Tag>();
4      public bool OrFilter { get; set; }
5
6      public async Task<List<Exercise>>
7          FilterExercisesAsync(List<Exercise> exercises);
8
9      public async Task<List<Training>>
10         FilterTrainingsAsync(List<Training> trainings);
11
12     public async Task<List<Plan>>
13         FilterPlansAsync(List<Plan> plans);
14
15     private async Task<List<ITagFilterable>>
16         FilterGeneralAsync(List<ITagFilterable> items);
17
18     interface ITagFilterable
19     {
20         Task<List<Tag>> GetTagsAsync();
21     }
22 }
```



Obrázek 4.4: Schéma filtrování cviků

4.2 Data storage

Data storage dělíme stejně jako podle složek na `Repositories` a `Databases`. `Repositories` jsou rozhraní definující funkce pro každý oddělený kus databáze. Jsou zde rozhraní pro cviky, tréninky, tréninkové plány, tagy, chat a uživatelská data. Každé rozhraní přikazuje implementaci jen těch funkcí, které pro danou entitu vyplývají z funkčních požadavků (viz podkapitola 1.6). Například rozhraní pro databázi chatu obsahuje jen funkce pro uložení a čtení dat, nikoliv mazání a editaci, protože mazání a editace z požadavků nevyplývá. Naopak rozhraní pro databázi cviků obsahuje funkce pro mazání, editaci (update), vytvoření, načtení. Všechny funkce daného rozhraní – repozitáře – jsou definovány s návratovou hodnotou `Task` nebo případně `Task<T>`, kde T je libovolný objekt (například `Task<Exercise>`), aby mohly být implementovány asynchronně.

Všechny implementace týkající se databází se nachází ve složce `Databases`. Tu dále dělíme na složky `LocalDatabase` – implementace SQLite databáze, `Firestore` – implementace Firestore databáze, `DatabaseHelpers` – pomocné třídy pro ovládání databází, plus je zde navíc samostatná třída ve zdrojovém souboru `DatabaseFactory.cs` (viz sekce 4.2.5).

4.2.1 Repositories

Jak bylo řečeno výše, všechny metody pro databáze ve všech repozitářích jsou definovány tak aby mohly být implementovány asynchronně. Zároveň definují jen nutné minimum metod pro daný celek. Repozitář je vytvořený pro každý celek, z uživatelského hlediska, který se ukládá do libovolné databáze:

- `IExerciseRepository` – repozitář pro cviky,
- `ITrainingRepository` – repozitář pro tréninky,

- `IPlanRepository` – repozitář pro tréninkové plány,
- `ITagRepository` – repozitář pro tagy,
- `IChatRepository` – repozitář pro vše týkající se chatu,
- `IUserRepository` – repozitář pro data o uživatelích,
- `IActivePlanRepository` – repozitář pro aktivní tréninkové plány.

4.2.2 DatabaseHelpers

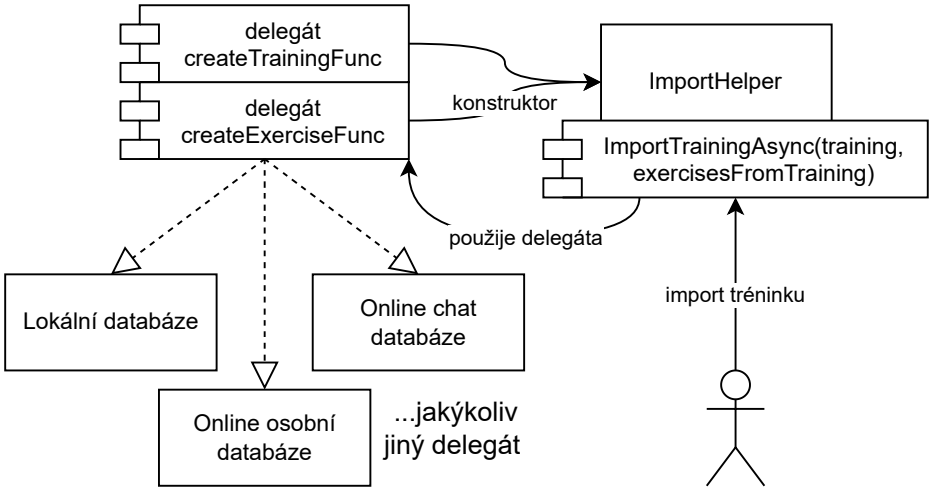
Složka `DatabaseHelpers` obsahuje třídy `ImportHelper` a `AfterDeletePatchingHelper`. `ImportHelper` je třída zajišťující import tréninkových aktivit do jiného uložiště a tedy i spravování odkazů napříč aktivitami (viz podkapitola 3.7). Třída `ImportHelper` je schválně implementována co nejobecněji, protože přenosu dat mezi databázemi nebo i v rámci databáze se využívá na více místech – import dat z offline do online databáze, zasílání tréninkových aktivit v chatu, ukládaní zaslaných aktivit z chatu. `AfterDeletePatchingHelper` se stará o mazání neplatných záznamů v databázi po tom co se odstraní nějaká z tréninkových aktivit.

ImportHelper

`ImportHelper` ukládá tréninkové aktivity pomocí delegátů, které se instanci třídy poskytnou při jejím vzniku. Třída proto poskytuje tři konstruktory, podle potřeby použití. Jeden pro ukládání pouze cviků, další pro ukládání cviků a tréninků a třetí pro ukládání všech tří tréninkových aktivit. V případě že potřebujeme importovat jen cviky, stačí nám definovat delegát pro uložení cviku. Každý z konstruktorů dále nabízí možnost přidat instanci `ITagRepository`, pokud nám záleží na párování tagů z cviků s tagy z dané databáze. Pokud instance `ITagRepository` není dodaná, cviky se uloží vždy jen tak jak jsou, včetně seznamu tagů, se kterými při takovém importu není nijak manipulováno. Toho se využívá v ukládání cviků do databáze chatu (viz 2.6.8 – Chat ve Firestoru). Pokud je instance `ITagRepository` dodaná v konstruktoru, s identifikátory tagů importovaných cviků se manipuluje při mapování na tagy z dodané databáze nebo při vytváření nových tagů v cílovém místě uložení. Nakonec pokud je použita funkce pro import libovolné tréninkové aktivity, jejíž delegát pro ukládání na cílové místo nebyl při vzniku instance dodán, funkce vyhodí `InvalidOperationException` výjimku.

Na schématu (viz Obrázek 4.5) je vidět jak probíhá import tréninku pomocí `ImportHelperu`. Když programátor potřebuje importovat do nějaké databáze data, vytvoří si instanci `ImportHelperu`, kde do konstruktoru dodá delegáta pro uložení daných tréninkových aktivit. Na schématu je příklad pro import tréninku. V takovém případě musí programátor dodat delegáta pro uložení tréninku a delegáta pro uložení cviku – `createTrainingFunc, createExerciseFunc`. Funkce `ImportTrainingAsync` pro import tréninku pak použije delegáta pro uložení tréninku i delegáta pro uložení cviku, protože trénink se na cviky odkazuje pomocí identifikátorů a neobsahuje je explicitně. Zároveň `ImportHelper` opraví právě

odkazy uvnitř tréninku, aby odpovídaly novým identifikátorům cviků v cílové lokalitě. Schéma zároveň znázorňuje pomocí přerušovaných šipek, že delegáti se mohou odkazovat na libovolné uložiště – konkrétně offline databázi, online (osobní) databázi nebo online databázi přidruženou k chatu. Zároveň, je zde teoreticky možnost dodat i delegáta pracujícího s jinou databází nebo provádějícího jiné libovolné operace, které `ImportHelper` již nekontroluje a případné výjimky nezachytává.



Obrázek 4.5: Schéma importu tréninku

AfterDeletePatchingHelper

Třída `AfterDeletePatchingHelper` obsahuje pro každou tréninkovou aktivitu jednu funkci, plus jednu funkci pro tagy. Funkce jsou určené pro projití celé uživatelovy databáze a promazání všech referencí na danou aktivitu/tag, aby v databázi nevznikala nevalidní data. Příslušné funkce mají být zavolány ihned po odstranění dané aktivity nebo tagu z databáze, aby pro tento účel byla použitá stejná instance databáze jako pro mazaní. `AfterDeletePatchingHelper` totiž taktéž používá pro vytvoření instance databáze `DatabaseFactory`, tedy použitá databáze závisí na tom zda-li je uživatel přihlášený nebo ne. Naopak nezávisí na pořadí volání zda-li je první daný záznam z databáze vymazaný nebo je nejdříve zavolaná daná funkce. Je ovšem potřeba aby tréninková aktivita/tag v argumentech funkcí měli původní identifikátor.

```
1 public class AfterDeletePatchingHelper
2 {
3     public async Task TagDeletedPatch(Tag tag);
4     public async Task ExerciseDeletedPatch(Exercise exercise);
5     public async Task TrainingDeletedPatch(Training training);
6     public async Task PlanDeletedPatch(Plan plan);
7 }
```

4.2.3 Online databáze – Firestore

Implementace online databáze se nachází ve složce `Firestore`. Nachází se zde jedna třída pro každý repozitář, která dané rozhraní repozitáře implementuje:

- `FirestoreTagDatabase`,
- `FirestoreExerciseDatabase`,
- `FirestoreTrainingDatabase`,
- `FirestorePlanDatabase`,
- `FirestoreUserDatabase`,
- `FirestoreChatDatabase`,
- `FirestoreActivePlanDatabase`.

Složka dále obsahuje statickou třídu `FirestoreNames`. Tato třída obsahuje pouze statické, `readonly` textové položky, ve kterých jsou uloženy konstanty pro pojmenování jednotlivých atributů či kolekcí ve struktuře Firestore databáze. Nakonec složka obsahuje třídu `AndroidTaskOnCompleteListener`, která slouží jako `onCompleteListener` pro android tasky.

Implementace Firestore tříd

Třídy implementující některý z repozitářů (rozhraní pro databáze), využívají knihovny `Xamarin.Firebase.Firestore` pro přístup do online databáze. Výsledkem volání jsou Android tasky, které převádíme do klasických C# tasků (viz podkapitola 3.4). Funkce pak jednotlivé výsledky dotazů do databáze `await`ují a v případě načítání dat, dále zpracovávají. Výsledkem tasku je v případě, kdy načítáme celou kolekci nebo posíláme do databáze `query` (dotaz), kolekce dokumentů – `QuerySnapshot` – nebo jen jeden konkrétní dokument v případě, kdy chceme konkrétní dokument a známe jeho identifikátor – `DocumentSnapshot`. Jediný případ kdy v našich implementacích posíláme do databáze `query`, jsou metody, kdy chceme načíst jen podmnožinu dokumentů (cviků, tréninků, plánů) na základě seznamu identifikátorů.

AndroidTaskOnCompleteListener

Třída `AndroidTaskOnCompleteListener` slouží jako univerzální implementace rozhraní `IOnCompleteListener`, po vzoru článku *Awaiting Google Play services tasks* [18]. Toto rozhraní definuje metody pro třídu, jejíž instance může být přiřazena Android tasku jako `onCompleteListener`. Naše implementace dostává v konstruktoru instanci `TaskCompletionSource`, která slouží pro kontrolu výsledku klasického C# tasku (viz dokumentace TaskCompletionSource [19]). Třída `TaskCompletionSource` má vlastnost `Task`, která představuje task, který daná instance třídy kontroluje, tedy tento task lze i ještě nedokončený předat k dalšímu zpracování. V našem případě se při dokončení Android tasku spustí provádění metody `OnComplete`, která nastaví výsledek tasku z instance `TaskCompletionSource`. Metoda kopíruje vlastnosti Android tasku do C# tasku, tedy pokud původní task je zrušený, i výsledný C# task se nastaví jako zrušený, pokud původní Android task nese výjimku, i výsledný task ponese výjimku a pokud je původní task úspěšný, okopíruje se výsledek původního tasku do výsledného. Poněvadž výsledkem Android tasku je vždy objekt dědící

po `Java.Lang.Object` a zároveň výsledkem našich dotazů do Firestore databáze mohou být kolekce dokumentů nebo jen jeden dokument, i náš výsledný C# task má jako výsledek nastavený `Java.Lang.Object`.

4.2.4 Offline databáze – SQLite

Pro implementaci lokální databáze jsme si vybrali technologii SQLite ve spojení s knihovnou pro .NET `sqlite-net-pcl` (viz sekce 2.4.1). Ve složce `LocalDatabase` se nachází jeden soubor pro implementaci každého rozhraní tréninkové aktivity, jeden soubor pro implementaci `ITagRepository`, jeden soubor pro implementaci `IActivePlanRepository` a samostatný soubor s ovládáním tabulky pro mapování tagu na cvik – `LocalTagRecordDatabase.cs`. Všechny tyto soubory implementují funkce pro ovládání tabulek SQLite. Tyto implementace potřebují mít otevřené `SQLiteAsyncConnection`, které stačí vždy jen jedno a jen jednou ho vytvořit. K tomu je zde ještě soubor `SQLiteDatabaseConnection.cs`. V tomto souboru je implementována třída `SQLiteDatabaseConnection`, která pomocí návrhového vzoru Singleton [25] poskytuje právě potřebnou instanci třídy `SQLiteAsyncConnection`. Tato instance se lazy inicializuje při první potřebě, ale znova již ji v daném běhu aplikace není třeba inicializovat.

Zároveň pro lokální databázi neimplementujeme rozhraní `IChatRepository` a `IUserRepository`, protože aplikace v offline režimu nemá podporovat dané funkce.

Modely pro SQLite

Vybraná knihovna pro ovládání SQLite nám umožňuje nepoužívat SQL dotazy a příkazy, včetně těch pro vytváření tabulek. Tabulky se vytvoří automaticky podle definovaných tříd a podle jejich C# atributů (viz dokumentace C# – atributy [10]). Abychom této funkce využili, pro modely cviku, tréninku, tréninkového plánu a aktivního tréninkového plánu, které potřebujeme ukládat v offline databázi, vytvořili jsme si ve podsložce `SQLModels` identické modely pro SQLite, ale s dvěma rozdíly. Identifikátor je zde typu `int` a jsou zde jen jednorozměrné atributy – tzn. třídy neobsahují seznamy – které do SQLite budeme ukládat do samotných tabulek (viz podkapitola 2.6). Dále jsme zde museli vytvořit modely i pro každou další tabulku v SQLite viz podkapitola 2.6 – sekce s modelováním pro SQLite. Pro každou tabulku v offline databázi je zde tedy jedna třída, jejíž atributy definují sloupce dané tabulky. Tyto třídy jsou samostatně v podsložce `SQLiteModels` a každá třída má ve jméně příponu *SQLite*. Žádná z těchto tříd se nevyskytuje jinde v programu než v implementaci lokální databáze.

4.2.5 DatabaseFactory

Pro potřeby přepínání správné databáze jsme si navrhli společné rozhraní databází a navrhli jsme si řešení jak dostat správnou implementaci daného rozhraní (viz podkapitola 3.6). Postup práce `DatabaseFactory` a její použití je popsáno v úvodu kapitoly (viz Obrázek 4.2).

`DatabaseFactory` obsahuje jednu factory metodu pro každý repozitář (viz sekce 4.2.1). Metody vracející repozitář, který může mít i offline implementaci,

mají volitelný `bool` argument, nabízející explicitní vrácení offline databáze. Výchozí hodnota argumentu je `false`. Možnosti explicitního vrácení offline databáze se využívá při importu dat z offline do online databáze.

```
class DatabaseFactory
{
    public IExercisesRepository MakeExerciseRepository(bool offline = false);
    public ITrainingRepository MakeTrainingRepository(bool offline = false);
    public IPlanRepository MakePlanRepository(bool offline = false);
    public ITagRepository MakeTagRepository(bool offline = false);
    public IActivePlanRepository
        MakeActivePlanRepository(bool offline = false);
    public IUserRepository MakeUserRepository();
    public IChatRepository MakeChatRepository();
}
```

4.3 Services

V Services (`EasyTrain/EasyTrain/Services`) máme dvě třídy. Třída `MyFirebaseMessagingService` je pouze přípravou na notifikace v zařízení (viz podkapitola 6.4) a zatím nemá žádnou funkčnost. Druhou třídou je statická třída `User` poskytující funkce potřebné k obsluze uživatelského účtu ve Firebase.

UserServices

`UserServices` obsahují pouze statickou třídu `User`, poskytující funkce k přihlášení, odhlášení, registraci, extrakci e-mailu, extrakci uživatelova identifikátoru a funkci vracející `bool` zda-li je uživatel přihlášený. Třída vytváří wrapper nad knihovnou `Xamarin.Firebase.Auth` a poskytuje jen funkce využité v naší aplikaci. Stejně tak vytváří wrapper pro výjímky z této knihovny – `UserException`.

4.4 GUI – Grafické uživatelské rozhraní

Základním pilířem grafického rozhraní v Android aplikaci jsou aktivity. Aktivita představuje jednu obrazovku dané aplikace. To jak aktivita vypadá se odvyjí od jejího grafického zdrojového souboru psaného v `xml` – takzvaného `layoutu`. Vzhled jiných grafických prvků (např. dialogů) je zpravidla taktéž definován `layouty`. Grafické prvky lze vytvářet i dynamicky ve vykonávaném kódu, ale pro větší prvky by to bylo zbytečně náročné. Všechny designové prvky definované v `xml`, ale i další grafické prvky – obrázky, položky menu, programátorem definované barvy nebo dimenze – jsou uloženy v adresáři `Resources`. Všechny `layouty` jsou pak uložené v podadresáři `layout`. Bohužel zde není možné soubory třídit jinak, tato struktura je definována přímo Androidem (viz Android Resource Basics [26]).

Aktivita může zobrazit více fragmentů najednou. Fragmenty jsou grafické prvky podobné aktivitám, připojené právě k aktivitám. Fragmenty mají vlastní `layouty` a vlastní životní cyklus spojený s callbacky. Fragment můžeme považovat za „podaktivitu“. V aplikaci používáme fragmenty jako „pod-obrazovky“,

kdy necháme část obrazovky nezměnenou (například toolbar s navigačním rozhraním), ale funkce obrazovky se změní.

V rámci aktivit a fragmentů jsou využívány základní grafické prvky Androidu, např. tlačítko, textové pole apod., ale i složitější struktury, jejichž ovládání vyžaduje netriviální množství kódu. Mezi tyto prvky patří dialogy a seznamy. Dialogy (viz sekce 2.8.2) jsou menší vyskakovací okna, kde uživatel neztratí spojení s původní stránkou (aktivitou/fragmentem). V aplikaci jsou pomocí dialogů implementovány menší funkční bloky, které mohou být použity na vícero místech. Zdrojové kódy v C# pro dialogy, jsou umístěné ve složce `Dialogs`. U seznamů je zapotřebí definovat zobrazení jednoho prvku v seznamu a také chování seznamu pro různé uživatelské vstupy – např. „swipe“ nebo kliknutí na daný prvek apod. K definování tohoto chování se používají `adaptéry`, jejichž zdrojové kódy, jsou umístěné ve složce `ScreenAdapters`.

Poslední složitější grafickou strukturou jsou komplexnější grafické prvky, složené z jednoduchých grafických prvků (`Views`), které potenciálně můžeme chtít používat opakovaně, tedy je chceme mít samostatně naprogramované. Jde pouze o design definovaný pomocí `xml` souboru a o statickou třídu nazvanou `Inflater`. V aplikaci jsou následující 4 `Inflater` třídy:

- `ExerciseCardViewInflater`,
- `PlanCardViewInflater`,
- `TagViewInflater`,
- `TrainingCardViewInflater`.

`Inflater` třídy slouží pouze k definovaní, jak má daný prvek vypadat s konkretními daty. Tyto třídy mají statické funkce, které jen vyplní daný prvek daty, případně definuje jeho chování – např. po zmáčknutí tlačítka. Tyto funkce můžeme přirovnat ke konstruktorům, ale pro konkrétní grafické prvky. Zdrojové kódy k `Inflater` třídám se nachází ve složce `ViewsInflaters`.

4.4.1 Aktivity a fragmenty

Aplikace se pomocí aktivit dělí na funkční celky. Aktivity – funkční celky – pak můžeme dále dělit na menší funkční celky pomocí fragmentů. Fragmenty představují konkrétnější funkce, které na mobilním zařízení zaberou celou obrazovku, ale chceme pro ně společné grafické rozhraní nebo např. jednoduše sdílet data mezi fragmenty. Aktivita pak mezi fragmenty přepíná podle potřeby. Příkladem je editace a zobrazení detailu cviku (viz 4.4.1 – Detail a editace tréninkové aktivity). Zdrojové C# soubory každého z těchto funkčních celků, složených z aktivit a fragmentů, jsou v samostatných podadresářích ve složce `ActivitiesAndFragments`.

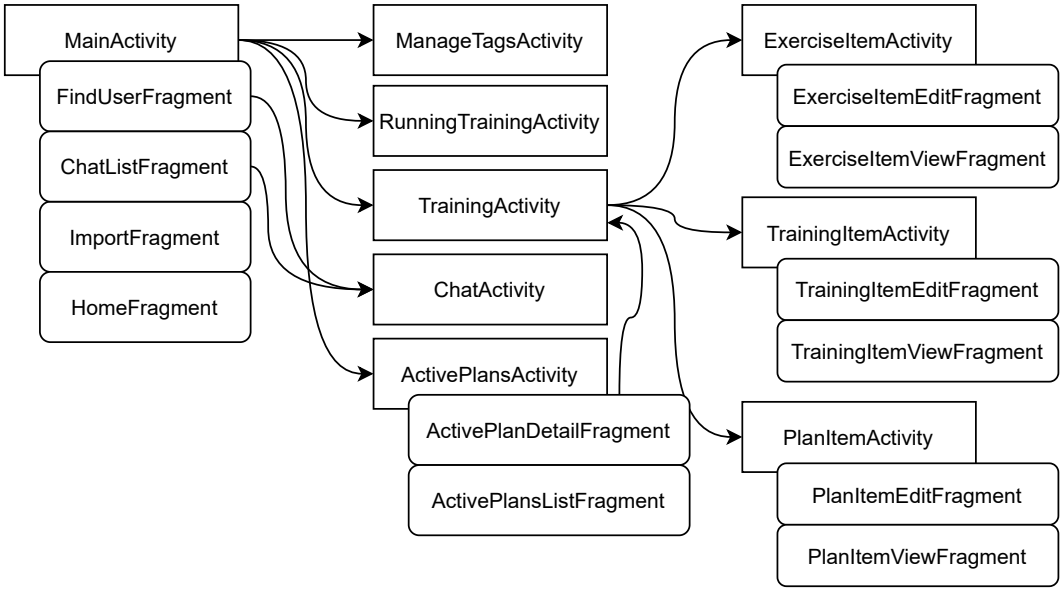
Když uživatel přejde v aplikaci na jinou funkcionalitu, zpravidla tím spustí novou aktivitu. Funkcionalita následující aktivity může být ovlivněna právě vstupem od uživatele v předchozí aktivitě, například když uživatel v seznamu klikne na cvik, aby si zobrazil jeho detaily. Tehdy si aktivity potřebují předat informace o tom, který cvik je ten, co je třeba zobrazit. Aktivity si mezi sebou mohou předávat data pomocí třídy `Intent` (viz Xamarin Android dokumentace – Intent

[27]), kde se data předávají stylem `klíč : hodnota`. Pokud je funkční celek implementovaný s pomocí fragmentů v rámci aktivity – například detail cviku a jeho editace – fragmenty si mohou mezi sebou data předávat skrze aktivitu, ke které jsou fragmenty přidružené. Každý fragment musí být přidružený k nějaké aktivitě, která poté spouští metody jeho životního cyklu (viz Xamarin Android dokumentace – Creating a fragment [28]) – např. `OnCreate`.

Konkrétní podobu rozdělení do funkčních celků, tedy aktivit a fragmentů, můžeme vidět na Obrázku 4.6, kde je znázorněno schéma aktivit s přidruženými fragmenty. Šipky na schématu pak znázorňují prokliky mezi aktivitami a fragmenty v aplikaci, tedy jak se k dané funkcionalitě lze v aplikaci prokliknout. Schéma například znázorňuje, že z hlavní aktivity (`MainActivity`) nebo přidružených fragmentů, se lze prokliknout na 5 jiných aktivit (viz šipky na Obrázku 4.6). Funkce jednotlivých aktivit a přidružených fragmentů jsou popsány v následujícím seznamu:

- `MainActivity` – hlavní aktivita:
  - `FindUserFragment` – obrazovka se seznamem registrovaných uživatelů,
  - `ChatListFragment` – obrazovka se seznamem všech uživatelových konverzací,
  - `ImportFragment` – obrazovka pro import tréninkových aktivit z offline do online databáze,
  - `HomeFragment` – domovská obrazovka aplikace.
- `ManageTagsActivity` – databáze tagů.
- `RunningTrainingActivity` – zobrazení spuštěného tréninku.
- `TrainingActivity` – databáze tréninkových aktivit.
- `ChatActivity` – obrazovka chatu.
- `ActivePlansActivity` – aktivita pro zobrazování spuštěných tréninkových: plánů
  - `ActivePlanDetailFragment` – obrazovka pro zobrazení detailu aktivního tréninkového plánu,
  - `ActivePlansListFragment` – obrazovka se seznamem aktivních tréninkových plánů.
- `ExerciseItemActivity` – zobrazení konkrétního cviku a jeho editace:
  - `ExerciseItemEdtiFragment` – obrazovka pro editaci cviku,
  - `ExerciseItemViewFragment` – obrazovka pro zobrazení detailu cviku.
- `TrainingItemActivity` – zobrazení konkrétního tréninku a jeho editace:
  - `TrainingItemEdtiFragment` – obrazovka pro editaci tréninku,
  - `TrainingItemViewFragment` – obrazovka pro zobrazení detailu tréninku.

- `PlanItemActivity` – zobrazení konkrétního plánu a jeho editace:
  - `PlanItemEdtiFragment` – obrazovka pro editaci jména tréninkového plánu,
  - `PlanItemViewFragment` – obrazovka pro zobrazení a editaci detailu tréninkového plánu – seznam dní a tréninků v jednotlivých dnech.



Obrázek 4.6: Schéma aktivit a fragmentů

Hlavní stránka

Když uživatel otevře aplikaci, zobrazí se hlavní aktivita – `MainActivity`. Její vzhled je dán souborem `activity_main.xml`. V rámci hlavní aktivity se zobrazují 4 fragmenty a vždy je zobrazen alespoň jeden z nich:

- `FindUserFragment`,
- `ChatListFragment`,
- `ImportFragment`,
- `HomeFragment`.

Tyto fragmenty mají různé funkce (popsané v seznamu v sekci 4.4.1), ale mají společné grafické rozhraní hlavní aktivity. Konkrétně mají všechny obrazovky dané těmito fragmenty společný `toolbar` a přístup do hlavního navigačního rozhraní (viz sekce 2.8.3). `MainActivity` implementuje chování `toolbaru` – v metodě `OnCreate` – a ovládání společného menu – v metodě `SetupDrawerContent`. Pomocí menu je možné se dostat na další funkce aplikace implementované v dalších aktivitách (viz Obrázek 4.6) nebo na funkce implementované pomocí jednoho ze čtyř fragmentů `MainActivity`. Prvky v menu jsou dány souborem `menu.xml` v podsložce `Resources/menu`. `MainActivity` tedy také implementuje přepínání mezi vlastními fragmenty a volání dalších aktivit.

V menu se dále nachází tlačítka pro přihlášení, odhlášení a registraci uživatele. Tyto funkce jsou uživateli poskytnuty pomocí volání funkce `OpenLoginRegisterDialog`, která zobrazí dialog obsahující daný formulář. Dialog není implementován samostatně, ale pouze jako součást `MainActivity`, protože jinde v aplikaci ho není třeba. Dialog je univerzální pro přihlášení i registraci a jeho vzhled se nastaví pomocí funkcí `RegisterSetup` a `LoginSetup`. Funkce `SetupGuiAfterSignIn` a `SetupGuiAfterLogout` implementují úpravu menu po přihlášení/odhlášení – skrytí/zobrazení tlačítek po přihlášení a odhlášení. Pro přihlášení, odhlášení a registraci uživatele v aplikaci `MainActivity` využívá třídu `UserServices` (viz 4.3). `MainActivity` pouze v dialogu zachytí případnou výjimku a zobrazí chybovou hlášku v dialogu (například v případě chybného hesla). Po registraci nového uživatele `MainActivity` zavolá funkci `AskForImportData`, jenž zobrazí dialog s dotazem na import dat z offline databáze do online databáze (viz funkční požadavky 1.6 – požadavek č. 26). Jako dialog pro dotaz na uživatele použije `AreYouSureDialog` (viz 4.4.2 – AreYouSure Dialog), protože jde o jednoduchý dotaz s odpovědí ano/ne. Funkce `UpdateUserAsync` v `MainActivity` také implementuje možnost změny jména přihlášeného uživatele pomocí dialogu. K tomu používá `UserRepository` (viz sekce 4.2.1) a `DatabaseFactory` (viz sekce 4.2).

`FindUserFragment` obsahuje pouze seznam všech uživatelů aplikace a vyhledávací textové pole. Vzhled fragmentu je definovaný v `xml` souboru `findUsers_fragment.xml`. Pro načtení všech uživatelů používá `UserRepository` (viz sekce 4.2.1), jejíž instanci získává pomocí `DatabaseFactory`. Filtrování probíhá po každé změně v textovém poli. Filtrace kontroluje zda-li je daný řetězec obsažený ve jménu uživatele nebo jeho e-mailu. Seznam uživatelů ovšem načte jen jednou, který si nadále pamatuje a filtrování provádí nad kopií seznamu, abychom se vyhnuli opakovanému načítání uživatelů z databáze. Po kliknutí na položku v seznamu se uživateli nabídne zaslat druhému uživateli zprávu. To se dále implementuje v `ChatActivity` (viz 4.4.1 – Konverzace).

Fragment `ChatListFragment` je velice podobný `FindUserFragment` s tím rozdílem, že zobrazuje seznam již započatých konverzací a není zde vyhledávací pole. Přesný vzhled je definován v souboru `messages_fragment.xml`. Stejně jako `FindUserFragment` používá `UserRepository` a tedy i `DatabaseFactory`. Po kliknutí na konverzaci se otevře detail dané konverzace v aktivitě `ChatActivity` (viz 4.4.1 – Konverzace).

`ImportFragment` implementuje možnost kdykoliv importovat data z offline databáze do online databáze (viz funkční požadavky 1.6 – požadavek č. 27). Vzhled fragmentu je definován v `import_fragment.xml`. Funkce importu je ovšem přístupná, pouze pokud je uživatel v aplikaci přihlášený. Fragment si při vzniku pomocí `DatabaseFactory` vytvoří jednotlivé instance offline databáze i momentálně používané online databáze (dané `Repository` – viz sekce 4.2.1). Fragment si také vyrobí instanci třídy `ImportHelper` (viz 4.2.2 – ImportHelper) a poskytne jí instance online databází. Pro výběr tréninkové aktivity k importu používá dialogy (viz 4.4.2 – Dialogy pro výběr tréninkové aktivity):

- `ChooseExerciseDialog`,
- `ChooseTrainingDialog`,
- `ChoosePlanDialog`.

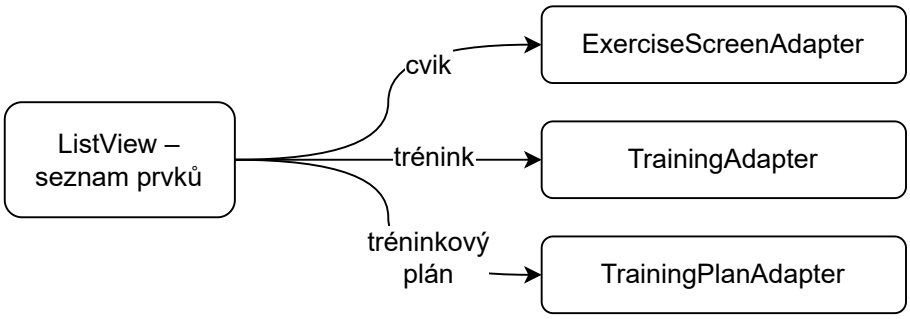
Dialogům při jejich vzniku předá příslušnou instanci offline databáze aby uživatel vybral tréninkovou aktivitu právě z offline databáze. Po výběru aktivity, kterou chce uživatel importovat, se případně z offline databáze načtou další aktivity, které jsou ve vybrané aktivitě obsažené (např. při výběru tréninku se musí načíst obsažené cviky) a pomocí `ImportHelperu` se data uloží do online databáze. V průběhu celého procesu po vybrání tréninkové aktivity k importu se uživateli zobrazí dialog načítání – `ProgressLoadingDialog` (viz 4.4.2 – Dialog načítání).

Databáze tréninkových aktivit

Pro databázi tréninkových aktivit (cviky, tréninky, plány – nikoliv grafické prvky) nám stačí jediná aktivita – `TrainingActivity`. Vzhled stránky je daný grafickým souborem `activity_training.xml`. Obrazovka se skládá ze tří základních částí:

- spodní navigační lišta (viz sekce 2.8.3) pro přepínání mezi databází cviků, tréninků a plánů,
- seznam prvků v databázi,
- plovoucí tlačítka pro přidání nového záznamu a pro filtrování záznamů.

Kliknutím na spodní lištu se pouze změní hodnota ve vlastnosti `ViewMode`, na jednu z tří možností `enumu mode` – `exercise, training, plan`. Jako postranní efekt `setteru` vlastnosti `ViewMode` se změní i obsah seznamu prvků. V aktivitě je pouze jeden seznam pro zobrazení všech tří tréninkových aktivit. Tento seznam je zobrazený pomocí Android třídy `ListView`. Jeho obsah se dynamicky mění pomocí změny vlastnosti `ListView.Adapter`, jenž definuje způsob zobrazení prvků v seznamu. Pro každou z tréninkových aktivit máme implementovaný vlastní `Adapter` (viz 4.4.3 – Seznam tréninkových aktivit v databázi). Na schématu je znázorněno jaká `Adapter` třída se používá pro zobrazení dané tréninkové aktivity (viz Obrázek 4.7) – například pro zobrazení tréninku v seznamu se používá třída `TrainingAdapter`. Pokud uživatel klikne na prvek seznamu, otevře se jedna ze tří aktivit, určených pro zobrazení detailu daného prvku – `ExerciseItemActivity, TrainingItemActivity, PlanItemActivity` (viz 4.4.1 – Detail a editace tréninkové aktivity).



Obrázek 4.7: Seznam položek a adaptéry které využívá

Nakonec jsou zde tlačítka pro filtraci tréninkových aktivit a pro přidání nového záznamu do databáze. Přidání nového záznamu do databáze spustí jednu ze tří

aktivit, stejných jako když uživatel klikne na prvek v seznamu (viz výše – 4.4.1 – Detail a editace tréninkové aktivity). Tlačítko pro filtrování záznamů otevře dialog `FilterDialog` (viz 4.4.2 – Filtrační dialog). Dialog při zavření vrátí aktivitě instanci `TagFilter` (viz sekce 4.1.1), která se odvíjí od uživatelova vstupu. Následně se zavoláním příslušné funkce na instanci `TagFilter` (viz sekce 4.1.1) vyfiltruje seznam tréninkových aktivit (podle režimu zobrazení) a zobrazí se upravený (vyfiltrovaný) seznam. Tuto funkcionalitu implementuje funkce `FilterItemsAsync`.

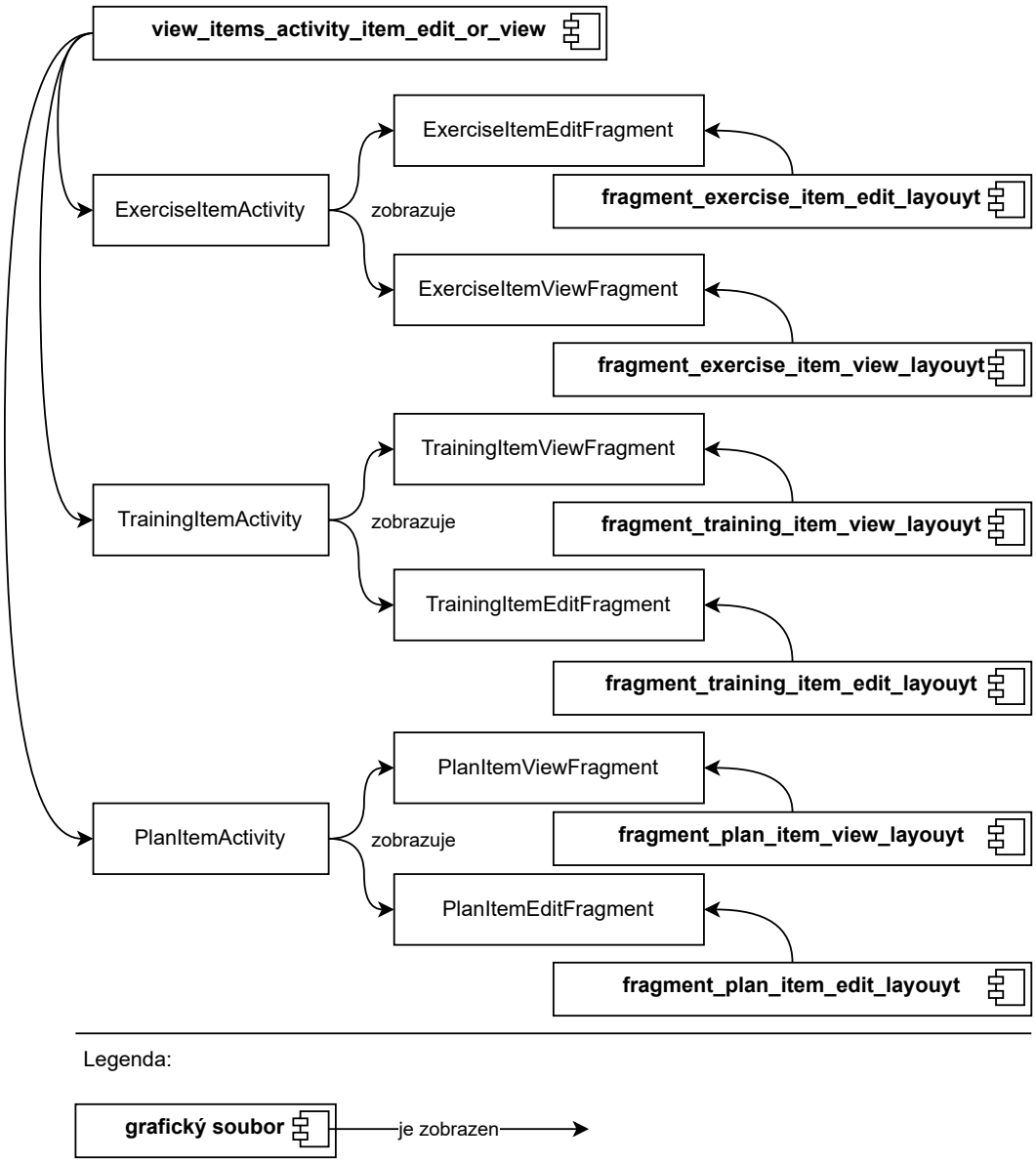
Detail a editace tréninkové aktivity

Pro zobrazení detailu či editaci nějaké tréninkové aktivity jsou vytvořeny tři aktivity, které všechny implementují stejný grafický soubor, kde je pouze `toolbar` a místo pro zobrazení fragmentů – `view_items_activity_item_edit_or_view`. Jak je znázorněno na schématu (viz Obrázek 4.8), těmito aktivitami jsou:

- `ExerciseItemActivity`,
- `TrainingItemActivity`,
- `PlanItemActivity`.

Každá z těchto aktivit dále používá dva fragmenty, jeden pro zobrazení detailu dané tréninkové aktivity, druhý pro jeho editaci nebo vytvoření nového. Výjimka je fragment pro zobrazené tréninkového plánu, který zároveň umožňuje editaci (viz 4.4.1 – Editace tréninkového plánu). Každý fragment pak má svůj vzhled definovaný vlastním grafickým `xml` souborem (viz Obrázek 4.8). Tento vzhled fragmentů odpovídá návrhům ze sekce 2.8.5.

Aktivity jsou odpovědné za zobrazení správného fragmentu a případné předání instance tréninkové aktivity. V aktivitě se tedy načítají data z databází. Stejně tak se zde volají ostatní operace databází, na základě vstupu od uživatele ve fragmentech – např. aktualizace dat, smazání dat. Například při editaci cviku se ve fragmentu `ExerciseItemEditFragment` vyplní model daty od uživatele a zavolá se funkce `UpdateExerciseAsync` na rodičovské aktivitě `ExerciseItemActivity`. Tato metoda se na základě identifikátoru rozhodne zda-li jde o nový záznam nebo již existující záznam, který je třeba aktualizovat a zavolá příslušnou operaci databáze. K těmto operacím využívají příslušné `repository` (viz sekce 4.2.1) a `DatabaseFactory` (viz sekce 4.2.5).



Obrázek 4.8: Schéma aktivit pro zobrazení a editaci položek

Všechny aktivity nadále používají `toolbar`, který aktivity využívají pro zobrazení takzvaného `options menu`, které v režimu zobrazení tréninkové aktivity nabízí editaci a smazání dané tréninkové aktivity.

Detail cviku – zobrazení tagů

V rámci fragmentu `ExerciseItemViewFragment` se nejen zobrazují textové atributy cviku samotného, ale navíc všechny jeho tagy (viz návrh – sekce 2.8.5). Také je zde tlačítko pro přidání dalšího tagu. Abychom dosáhli grafického zobrazení jako je v návrhu, tj. aby se jednotlivé tagy byly graficky ohraničeny a zarovnávali se do řádků, nehledě na velikost daného tagu, je zde použit `FlexboxLayout`, konkrétně wrapper knihovna pro originální knihovnu od Googlu (viz stránka nuget balíčku [29]). Knihovna `FlexboxLayout` zde vytváří podobné vlastnosti jako má CSS Flexible Box Layout. `FlexboxLayout` je zde pouze kontejner pro další grafické prvky. Za poskytnutí instance grafického prvku pro tag zde odpovídá

statická třída `TagViewInflater`, konkrétně funkce `InflateTagView`.

Pro přidání již existujícího tagu nebo vytvoření a přidání tagu je zde použit `AddTagDialog` (viz 4.4.2 – Dialog pro přidání a výběr tagu). Odpovědnost za uložení případného nového tagu do databáze je ponechána právě dialogu.

Editace tréninkového plánu

V části naší práce zabývající se návrhem uživatelského rozhraní při zobrazovaní tréninkového plánu (viz 2.8.5 – Detail tréninkových plánů) jsme si zadali způsob editace tréninkového plánu pomocí systému „drag and drop“. Tato funkce je tedy dostupná okamžitě při zobrazení detailu tréninkového plánu, tedy ve fragmentu `PlanItemViewFragment`. Přesto je zde i fragment `PlanItemEditFragment`, který implementuje pouze editaci jména daného plánu. Na rozdíl od zbytku aplikace, kde se pro zobrazení seznamu používá Android třída `ListView`, se při zobrazování seznamu dnů v plánu i tréninků v konkrétním dni používá třída `RecyclerView`. Podobně jako u `ListView` se i zde používají adaptéry pro definování vzhledu a chování. Zde je použit `DayInPlanRecyclerViewAdapter` (viz 4.4.3 – Adaptéry pro tréninkový plán). Kromě jména tréninkového plánu, se editace plánu odvíjí od uživatelovy manipulace se seznamem, tedy s jeho adaptérem. Proto editaci plánu (krom jména) implementuje právě tento adaptér. Fragment zde nechává odpovědnost za ukládání změn v tréninkovém plánu na adaptéru.

Obrazovka konverzace

Funkcionalita a chování obrazovky pro konkrétní konverzaci je implementováno aktivitou `ChatActivity`. Vzhled obrazovky je definován grafickým souborem `activity_chat.xml`. Aktivita při jejím startu očekává jméno příjemce dané zprávy a jeho identifikátor. Bez identifikátoru zobrazí obrazovku konverzace, ale bez žádných zpráv. Aktivita si pomocí `DatabaseFactory` (viz sekce 4.2.5) vytvoří instanci `IChatRepository` (viz sekce 4.2.1) a načte či vytvoří danou konverzaci. Zprávy v modelu konverzace (viz sekce 2.6.8) nejsou seřazené. Po načtení se tedy zprávy seřadí podle času odeslání. Pro zobrazení zpráv je zde použita třída `ListView` s adaptérem `ChatMessageAdapter`, který zobrazuje zprávy podle jejich typu a podle jejich odesílatele (viz 4.4.3 – Adaptér zpráv v konverzaci). Zprávy obsahující tréninkovou aktivitu a nikoliv text se zobrazují pouze pomocí jednoduchých obrázků. Aby si uživatel mohl detailněji prohlédnout libovolnou poslanou tréninkovou aktivitu, je zde pro její zobrazení použit `ChatShowItemDialog` (viz 4.4.2 – Dialog pro zobrazení tréninkové aktivity).

Pro zasílání zpráv jsou zde funkce podle typu posílané zprávy:

- `SendNewTextMessage` – pro posílání textových zpráv,
- `SendExercise` – pro posílání cviků,
- `SendTraining` – pro posílání tréninků,
- `SendPlan` – pro posílání tréninkových plánů.

Zatímco u textové zprávy se pouze vyplní její text a může se uložit do databáze zpráv (poslat), u zpráv posílajících tréninkovou aktivitu je nejdříve třeba danou

aktivitu a případně další obsažené aktivity uložit do databáze chatu. Identifikátor dané aktivity odkazující se do databáze konverzace, se pak do těla zprávy vyplní společně s jejím typem (viz sekce 2.6.8). Právě k tomu slouží výše zmíněné funkce (viz seznam). Pro tento účel je zde pro každou z tréninkových aktivit ještě funkce zařizující načtení ze soukromé databáze a následné uložení do databáze konverzace.

- `SaveExerciseToDbAsync`,
- `SaveTrainingToDbAsync`,
- `SavePlanToDbAsync`.

Pro načtení obsažených tréninkových aktivit (např. pro trénink to jsou cviky) jsou zde použity příslušné repozitáře (viz sekce 4.2.1) a pro správné uložení dané aktivity do databáze konverzace – tj. ošetření referencí mezi modely – je zde použita třída `ImportHelper` (viz sekce 4.2.2).

Obrazovka spuštěného tréninku

Obrazovka pro spuštěný trénink implementuje funkční požadavek č. 18 (viz podkapitola 1.6). Vzhled obrazovky je definován v `activity_running_training`. Funkce je implementována pomocí aktivity `RunningTrainingActivity`. Aktivita umožňuje být spuštěna pro předem vybraný trénink(např. proklikem z aktivního tréninkového plánu). Toto je umožněno předáním identifikátoru tréninku pomocí třídy `Intent`, pod klíčem „itemId“. Pro načtení daného tréninku aktivita používá `DatabaseFactory, IExerciseRepository a ITrainingRepository`. Pokud aktivitě není předán žádný identifikátor, nechá uživatele vybrat trénink z databáze pomocí `ChooseTrainingDialog` (viz 4.4.2 – Dialogy pro výběr tréninkové aktivity).

Protože model tréninku obsahuje pouze reference na obsažené cviky (viz sekce 2.6.4) – `ExerciseInTrainingRecord`, je potřeba cviky separátně načíst (klasicky pomocí dané `IExerciseRepository`) a následně spojit s odpovídajícími instancemi `ExerciseInTrainingRecord`. K tomu je zde používaná funkce `MergeExercisesAndExercisesInTrainingsRecords` implementovaná statickou třídou `ModelsFunctions`. Záznamy se z databáze nenačítají nijak seřazené, ale protože na pořadí cviků v tréninku záleží, je potřeba před zobrazením uživateli, seznam setřídit podle pořadí. To vše implementuje funkce `StartTrainingAsync`. Funkce je asynchronní, protože potřebuje počkat na výsledek asynchronní operace načítání dat z databáze.

Zobrazení aktuálního cviku implementuje funkce `ShowCurrentExercise`. Pro procházení cviků jsou zde funkce `PreviousExercise, NextExercise`. Pro podporu „swipování“ je zde implementována interní třída `OnSwipeTouchListener`.

Obrazovka aktivního plánu

Funkce aktivních tréninkových plánů (viz podkapitola 1.6 – požadavek č. 40) je implementována pomocí aktivity `ActivePlansActivity` a dvou fragmentů přidružených k této aktivitě. Aktivita zobrazí `ActivePlansListFragment`, když uživatel chce zobrazit seznam všech právě aktivních plánů. To je též výchozí obra-

zovka této funkcionality. Když si uživatel chce prohlédnout detail konkrétního aktivního tréninkového plánu, aktivita zobrazí `ActivePlanDetailFragment`.

`ActivePlansActivity` při jejím startu načte všechny záznamy o aktivních plánech z databáze (viz sekce 2.6.6), tedy použije `DatabaseFactory` (viz sekce 4.2.5) a příslušné repozitáře (viz sekce 4.2.1). Model aktivního plánu se na plán odkazuje pomocí identifikátoru, je tedy potřeba načíst z databáze i plány a spárovat je s danými záznamy aktivních tréninkových plánů. Toto implementuje funkce `LoadActivePlans`. Tato funkce dále implementuje smazání všech již zastaralých záznamů z databáze. Projde seznam dvojic záznamů aktivních plánů a odpovídajících plánů a pokud byl daný plán spuštěn před více dny než je jeho délka, smaže se z databáze. Takto vytříděný seznam aktivita předá `ActivePlansListFragment`, kde se seznam zobrazí.

`ActivePlansListFragment` má dvě funkce – zobrazení seznamu všech aktivních tréninkových plánů a spuštění tréninkového plánu. Načtení aktivních plánů zajišťuje rodičovská aktivita (viz předchozí odstavec). K zobrazení seznamu používá třídu `ListView` ve spojení s `ActivePlansAdapter` (viz sekce 4.4.3). Kliknutím na prvek seznamu se otevře detail plánu – `ActivePlanDetailFragment` (viz další odstavec). Funkce `StartNewPlan` implementuje spuštění nového tréninkového plánu. Používá `ChoosePlanDialog` (viz 4.4.2 – Dialogy pro výběr tréninkové aktivity) k vybrání daného plánu z databáze a následně uloží nový záznam do databáze aktivních tréninkových plánů. K tomu funkce používá `DatabaseFactory` a `IActivePlanRepository`.

`ActivePlanDetailFragment` již v konstruktoru dostane instanci `ActivePlan` a příslušnou instanci `Plan`. Jediná funkce, kterou fragment poskytuje je zobrazení detailu daného plánu. K tomu využívá Android třídu `RecyclerView` ve spojení s třídou `ActivePlanDaysRecyclerViewAdapter`. V rámci adaptéru se dále používá `ActivePlanDayInPlanDetailDialog` (viz 4.4.2 – Dialogy detailu dne z tréninkového plánu), který zobrazí detail konkrétního dne v plánu a umožní spustit libovolný trénink z toho dne (viz 4.4.1 – Obrazovka spuštěného tréninku).

Databáze tagů

Vedle databáze tréninkových aktivit máme v aplikaci odděleně databázi tagů (viz podkapitola 1.6 – požadavek č. 19). Tato funkcionalita je celá implementována v aktivitě `ManageTagsActivity`. Funkcionalita je oddělená od databáze tréninkových aktivit, protože jde o pouhý prostředek jak aktivity třídit a zároveň předpokládáme, že by tato funkce byla používána tak často jako právě databáze tréninkových aktivit. Uživateli se tagy zobrazí pomocí třídy `ListView` a adaptéru `TagAdapter`. Aktivita také nabízí textové pole pro hledání tagu podle toho, zda-li jméno tagu obsahuje daný podřetězec.

Aktivita uživateli nabízí mazání tagu a vytváření nových tagů (vytváření nových tagů je možné i na obrazovce detailu cviku, viz 4.4.1 – Detail cviku – zobrazení tagů). Vytváření i mazání tagů je implementováno pomocí univerzálního dialogu `TagEditDialog` (viz 4.4.2 – Dialog pro správu tagu), který je připravený i na případnou editaci jména tagu (zatím není součástí). Za samotnou manipulaci s databází zde ovšem odpovídá `ManageTagsActivity` a používá k tomu `DatabaseFactory` a `ITagRepository`. Pokud uživatel chce daný tag odstranit, zobrazí se mu navíc `AreYouSureDialog` (viz 4.4.2 – AreYouSure Dialog) pro potvrzení jeho záměru, zároveň s upozorněním možného efektu na cviky obsahující

daný tag (viz podkapitola 1.6 – požadavek č. 19.1)

4.4.2 Dialogy

K čemu dialogy slouží jsme si řekli již v sekci 2.8.2 – zobrazení detailu, dotazování se uživatele, drobné funkce. Uzavření malé funkce do dialogu nám umožňuje tuto funkcionalitu použít nehledě na Android aktivitě, která v aplikaci zrovna běží. Stejně jako aktivity či fragmenty (viz sekce 4.4.1) i vzhled dialogů je definován grafickým `xml` souborem. V aplikaci jsou dialogy implementovány zobrazovány normálním vytvořením jejich instance pomocí konstruktoru. V konstruktoru daného dialogu se pak volá funkce dialogu `Show`, která daný dialog zobrazí na obrazovce. V případě že dialog (vstup od uživatele) dále ovlivní další chování aplikace, je to v naší práci implementováno pomocí callbacků. Dialog dostane v konstruktoru callback, který zavolá podle jeho povahy – např. `onConfirmAction` (viz 4.4.2 – AreYouSure Dialog) je akce, která se zavolá, když vstup od uživatele je potvrzující.

AreYouSure Dialog

`AreYouSureDialog` v aplikaci používáme pro účely dotázání se uživatele, zdali si je jistý a chce pokračovat v dané operaci. Vzhled dialogu je definován v souboru `are_you_sure_popup_dialog`. Dialog nabízí a zároveň vynucuje dva parametry. Řetězec `messageText` je text, který se zobrazí jako zpráva uživateli v dialogu (např. „Opravdu chcete smazat tento cvik?“). Druhý parametr je callback `onConfirmAction`, který se zavolá při kladné odpovědi uživatele.

Dialogy pro výběr tréninkové aktivity

V aplikaci je hodně míst, kde je třeba vybrat nějakou tréninkovou aktivitu z databáze. Může jít o posílaní aktivit jiným uživatelům, import z offline databáze, přidávání cviků do tréninku apod. Pro tento účel jsou v aplikaci implementovány univerzální dialogy pro výběr jednotlivých aktivit:

- `ChooseExerciseDialog`,
- `ChooseTrainingDialog`,
- `ChoosePlanDialog`.

Tyto dialogy uživateli zobrazí seznam všech tréninkových aktivit daného typu v jeho databázi. Uživateli pak umožní jednu položku v seznamu označit a následně konečně vybrat. Všechny tyto dialogy vyžadují v konstruktoru callback, volaný když uživatel vybere danou tréninkovou aktivitu (např. `Action<Exercise> onExerciseChosedCallback`). Instanci vybrané tréninkové aktivity vloží do callbacku jako argument. Dále v konstruktoru nabízí, jako volitelný argument, instanci příslušného repozitáře pro danou tréninkovou aktivitu (viz sekce 4.2.1), kterou dialog použije jako zdrojovou databázi pro načtení všech aktivit daného typu. Pokud tato instance není dodaná, pro získání této instance se zde používá `DatabaseFactory` (viz sekce 4.2.5). Všechny dialogy nakonec uživateli umožňují i

filtrovat seznam všech aktivit podle jména – pomocí textového pole přímo v dialogu, ale také pomocí tagů, k čemuž využívá `FilterDialog` (viz 4.4.2 – Filtrační dialog).

Dialogy se jinak v implementaci neliší až na `ChooseTrainingDialog`, který v konstruktoru nabízí další volitelný argument – definovat callback volaný při zrušení dialogu ze strany uživatele (kliknutí na „cancel“ nebo na obrazovku mimo dialog). Této funkce je u dialogu využíváno ve funkci spuštěného tréninku (viz 4.4.1 – Obrazovka spuštěného tréninku), když se uživatel rozhodne zavřít dialog, aniž by vybral trénink k momentálnímu trénování.

Dialog načítání

`ProgressLoadingDialog` je používán jako grafické znázornění práce probíhající na pozadí. V aplikaci je používán, když se například potřebuje projít databáze a promazat záznamy o konkrétním tagu, který uživatel předtím smazal. Vzhled dialogu je určen souborem `loading_progressbar.xml`. Dialog má povinný parametr – textový řetězec, určující co se uživateli zobrazí za zprávu, vedle točícího se kolečka znázorňujícího načítání. Dialog dále obsahuje volitelný parametr `cancelable`, jehož výchozí hodnota je `true`. Parametr určuje, zda-li dialog může být uživatelem zrušen pomocí hardwarového tlačítka „zpět“ na zařízení nebo pomocí ťuknutí na obrazovku mimo pole dialogu.

Filtrační dialog

Kdekoliv v aplikaci, kde lze procházet tréninkové aktivity nebo kde uživatel nějakou aktivitu vybírá (viz 4.4.2 – Dialogy pro výběr tréninkové aktivity), je umožněno uživateli aktivity třídit pomocí tagů. K tomuto účelu slouží `FilterDialog`. Vzhled dialogu je určen souborem `filter_dialog_popup`.

Dialog v konstruktoru vyžaduje callback `onFilterSetCallback` s instancí třídy `TagFilter` jako parametr (viz sekce 4.1.1). Dialog na základě vstupu od uživatele vyplní instanci `TagFilter`, kterou při potvrzení filtrování (kliknutím na tlačítko „Filter“) předá jako parametr právě callbacku `onFilterSetCallback`. Dále dialog nabízí instanci `TagFilter` jako volitelný parametr v konstruktoru, výchozí hodnota je `null`. Pokud žádná instance není předána, vytvoří si dialog prázdný model a postupuje jak je napsáno výše. Pokud je předaná instance nerovna `null`, použije se jako výchozí hodnota. Toho se používá, když chce uživatel navázat na své předchozí filtrování v rámci dané funkce (např. když po prvním filtrování chce přidat do filtru ještě další tag).

Dialog nabízí tlačítko pro vynulovaní filtru. K zobrazení všech tagů v uživatelově filtru dialog používá `FlexboxLayout`. Tagy ve `FlexboxLayoutu` zobrazuje stejně jako se zobrazují tagy u detailu cviku. Pro přidávání tagu do filtru dialog používá `AddTagDialog` (viz 4.4.2 – Dialog pro přidání a výběr tagu).

Dialogy detailu dne z tréninkového plánu

Tréninkový plán je pouze seznam dnů, kde každý den obsahuje seznam tréninků. Pro zobrazení detailu jednoho dne tedy potřebujeme zobrazit seznam tréninků. V analýze uživatelského rozhraní aplikace (viz 2.8.5 – Detail tréninkových

plánů, 2.8.7 – Spuštěný tréninkový plán) jsme si navrhli zobrazit tento detail pomocí dialogu. Detail dne v tréninkovém plánu je třeba zobrazit ve dvou funkcích a každá funkce má svoji implementaci dialogu:

- `DayInPlanDetailDialog` – při běžném procházení a editaci tréninkového plánu v rámci databáze (viz 4.4.1 – Editace tréninkového plánu),
- `ActivePlanDayInPlanDetailDialog` – při procházení aktivního tréninkového plánu (viz 4.4.1 – Obrazovka aktivního plánu).

Tyto dialogy nejsou myšleny nijak univerzálně, ale čistě účelově pro danou funkcionalitu a nahrazují samostatnou obrazovku. Vzhled obou dialogů je definovaný stejným grafickým souborem – `day_in_plan_detail_popup_dialog.xml`. Stejně tak oba dialogy vyžadují v konstruktoru instanci daného `PlanDay` (viz sekce 2.6.5) a identifikátor plánu, ke kterému den patří. `DayInPlanDetailDialog` navíc vyžaduje v konstruktoru callback `onDismissAction`, který se zavolá, když se dialog zavře. Callback se zde používá k zobrazení změn v daném dnu na obrazovce.

Oba dialogy používají pro zobrazení seznamu tréninků třídu `RecyclerView` ve spojení s adaptéry `TrainingInPlanRecyclerViewAdapter` pro editaci plánu a `ActivePlanTrainingsRecyclerViewAdapter` pro detail dnu aktivního tréninkového plánu. Nakonec dialog pro editaci obsahuje navíc tlačítko pro přidání dalšího tréninku do daného dnu, k čemuž používá `ChooseTrainingDialog` (viz 4.4.2 – Dialogy pro výběr tréninkové aktivity). Na druhé straně dialog detailu aktivního plánu navíc zobrazuje popup menu, po kliknutí na trénink, s možností přímo spustit daný trénink.

Dialog pro přidání a výběr tagu

Při editaci cviku nebo filtraci tréninkových aktivit, může uživatel potřebovat vybrat tag z jeho databáze. K tomuto účelu slouží `AddTagDialog`. Jeho vzhled je definovaný v souboru `add_create_tag_popup.xml`. Při editaci cviku tento dialog dokonce uživateli nabízí možnost vytvořit nový tag. V implementaci je tato volba zařízená volitelným parametrem `creatingAllowed` v konstruktoru, jehož výchozí hodnota je `true` a tedy je standardně v dialogu povoleno vytvářet nové tagy. V konstruktoru je navíc jako povinný parametr callback `onTagChosenCallback`. Tento callback se zavolá s instancí vybraného tagu jako parametrem, když uživatel učiní výběr tagu.

Grafické rozhraní je zde implementováno pomocí `spinneru` (viz Xamarin Android dokumentace – Spinner [11]), který po rozbalení nabídne všechny tagy z databáze. Pokud je povoleno vytváření nových tagů, `spinner` po rozbalení jako první nabídne možnost „Vytvořit nový tag“. Když je tato možnost vybrána, zobrazí se v dialogu textové pole pro název nového tagu. Naopak pokud tato možnost vybrána není, textové pole je schované. Když uživatel vybere tag, který chce a potvrdí výběr tlačítkem, zavolá se callback `onTagChosenCallback` s daným tagem jako parametrem. Pokud Uživatel vytvářel nový tag, nejdříve se tag uloží do uživatelovy databáze pomocí `DatabaseFactory` (viz sekce 4.2.5) a `ITagRepository` (viz sekce 4.2.1) a až po uložení se zavolá callback.

Dialog pro zobrazení tréninkové aktivity

`ChatShowItemDialog` je dialog pro zobrazení detailu libovolné tréninkové aktivity. Za detail se zde nepovažuje plný rozsah dané aktivity, ale především jméno a případně nějaké drobnosti. Dialog je univerzální, protože sám o sobě nepracuje s modely tréninkových aktivit, ale pouze dostane hotové `view`, které zobrazí. K tomu je zde veřejná funkce `ShowItemView(View itemView)`.

Tento dialog je určený pro funkci konverzace, kde se zprávy s poslanými tréninkovými aktivitami zobrazují pouze pomocí jednoduchých obrázků. Tento dialog uživateli zobrazí detail dané tréninkové aktivity a umožní uživateli danou tréninkovou aktivitu uložit do své databáze. K tomuto účelu slouží callback `onSaveAction`, předávaný v konstruktoru.

Dialog pro zobrazení nápovědy

`QuickHelpDialog` je nejjednodušší dialog v celé aplikaci. Jeho design je definován v souboru `quick_help_popup_dialog.xml`. Obsahuje pouze pole pro zobrazení textu. Text, který zobrazí je povinný parametr konstruktoru. Dialog slouží pro zobrazení libovolné textové zprávy uživateli. V aplikaci je tento dialog použit pouze pro zobrazení rychlé nápovědy. Tuto funkci v aplikaci nabízí pouze toolbar (viz 4.4.4).

Dialog pro správu tagu

Pro manipulaci s databází tagů je v aplikaci implementován `TagEditDialog`. Jeho vzhled je definován v souboru `tag_edit_dialog`. Dialog je zamýšlen jako univerzální dialog pro editaci tagu. Protože aplikace neumožňuje editaci názvu tagu, je momentálně v aplikaci použit pouze pro vytváření nového tagu nebo mazání již existujícího tagu z databáze. K vytvoření i mazání tagu je použit v rámci aktivity s databází tagů a nikde jinde (viz 4.4.1 Databáze tagů). Nové tagy lze totiž vytvářet i pomocí `AddTagDialogu` (viz 4.4.2 – Dialog pro přidání a výběr tagu).

Dialog obsahuje pouze dvě textová pole, jedno informační, druhé pro editaci a tlačítka pro zrušení dané akce (zavření dialogu) a potvrzení dané akce. V konstruktoru dialogu je tag jako volitelný parametr. Když dialog tag v konstruktoru dostane, upraví svoje grafické rozhraní na mazání daného tagu. Pokud tag nedostane nebo je předaná reference pouze `null`, dialog grafické rozhraní nastaví na vytvoření nového tagu. Již povinným parametrem dialogu je callback `onTagEditedAction`, zavolaný s daným tagem jako parametrem po vytvoření tagu nebo po kliknutí na tlačítko smazat. Dialog sám o sobě s databází nepracuje. Callback se zde tedy používá pro operace s databází.

4.4.3 Seznamy a AdapterViews

V naší aplikaci, ale i obecně v jiných mobilních aplikacích, jsou uživateli často zobrazovány seznamy s různými položkami. Abychom mohli přizpůsobit vzhled těchto položek v seznamu a abychom mohli zobrazovat libovolné modely (třídy), používají se na Androidu tzv. `adaptéry`. Adaptér je třída, která definuje právě

vzhled, ale i chování jednotlivých prvků v seznamu. Instance `adaptéru` se pak předá danému seznamu.

Seznamy jsou pak na Androidu implementovány pomocí tříd `ListView` a `RecyclerView`. `Adaptéry` pro `ListView` dědí po třídě `BaseAdapter<T>`, kde `T` je zobrazovaná entita. Na druhé straně `adaptéry` pro `RecyclerView` dědí po třídě `RecyclerView.Adapter`. Třída `RecyclerView` je novější a poskytuje více možností použití, například jednoduché dynamické přidávání a odebírání prvků ze seznamu nebo různé rozložení seznamu (například mřížka se 3 položkami na jednom řádku). Ve většině případů jsme si v aplikaci vystačili s jednodušší třídou `ListView`, pouze při zobrazování detailu tréninkového plánu jsme použili `RecyclerView` (viz 4.4.1 – Editace tréninkového plánu, Obrazovka aktivního plánu). V této sekci si popíšeme některé adaptéry, které nejsou jen přímočaré grafické zobrazení nějakého modelu. Všechny `adaptéry` jsou ve sloužce `ScreenAdapters`:

- `ActivePlanDaysRecyclerViewAdapter`,
- `ActivePlanTrainingsRecyclerViewAdapter`,
- `ActivePlansAdapter`,
- `ChatMessageAdapter`,
- `DayInPlanRecyclerViewAdapter`,
- `ExerciseInTrainingEditRecyclerViewAdapter`,
- `ExerciseInTrainingViewAdapter`,
- `ExerciseNameAdapter`,
- `ExerciseScreenAdapter`,
- `PersonsAdapter`,
- `PlanChooseListAdapter`,
- `TagAdapter`,
- `TrainingAdapter`,
- `TrainingChooseListAdapter`,
- `TrainingInPlanRecyclerViewAdapter`,
- `TrainingPlanAdapter`.

Adaptéry pro tréninkový plán

V aplikaci jsou celkem 4 různé adaptéry pro zobrazení detailů tréninkového plánu. Dva adaptéry jsou pro zobrazení aktivního tréninkového plánu a dva pro zobrazení detailu a editaci plánu v databázi. Zároveň dva z těchto adaptérů jsou určeny k zobrazení seznamu dní v plánu a zbylé dva jsou k zobrazení tréninků v rámci jednoho dne. Všechny tyto adaptéry dědí po `RecyclerView.Adapter` a jsou tedy určené pro třídu `RecyclerView`. Zároveň všechny tyto adaptéry implementují vnořenou třídu `MyViewHolder` dědící po `RecyclerView.ViewHolder`, jejíž instance představuje prvek, graficky zobrazující jednu položku v seznamu. Těmito adaptéry jsou:

- `ActivePlanDaysRecyclerViewAdapter` – pro seznam dnů aktivního tréninkového plánu,
- `ActivePlanTrainingsRecyclerViewAdapter` – pro seznam tréninku v jednom dni v rámci aktivního tréninkového plánu,
- `DayInPlanRecyclerViewAdapter` – pro seznam dnů tréninkového plánu,
- `TrainingInPlanRecyclerViewAdapter` – pro seznam tréninků tréninkového plánu.

`ActivePlanDaysRecyclerViewAdapter` je adaptér pro zobrazení dnů aktivního tréninkového plánu. V konstruktoru dostane instanci `ActivePlan` (viz 2.6.6) a instanci `Plan` (viz 2.6.5), kterého se spuštěný plán týká. V konstruktoru si seřadí dny a tréninky v rámci dne a dále už jen zobrazí seznam dnů. Podle data spuštění daného plánu, přebarví na červeno tu položku v seznamu dnů, která odpovídá reálnému dnu, kdy si uživatel aktivní tréninkový plán zobrazuje. Nakonec už jen nastaví pro každou položku seznamu `OnClickListener`, který otevře dialog `ActivePlanDayInPlanDetailDialog` (viz 4.4.2 – Dialogy detailu dne z tréninkového plánu) a v něm zobrazí tréninky daného dne.

`ActivePlanTrainingsRecyclerViewAdapter` je adaptér zobrazující položky v seznamu tréninků aktivního tréninkového plánu. V konstruktoru dostane instanci daného dne, ze kterého má zobrazit dané tréninky – `PlanDay` (viz 2.6.5). `PlanDay` obsahuje pouze seznam záznamů odkazujících se na trénink pomocí identifikátoru – `TrainingInDayRecord`. Adaptér tedy pro každou položku tréninku volá funkci `GetTrainingAsync` na dané instanci `TrainingInDayRecord`, čímž načte trénink z databáze. Pro každou položku seznamu adaptér nastaví `OnClickListener`, který zobrazí `popup menu` (viz Xamarin Android dokumentace – Popup menu [12]), které uživateli nabídne spuštění daného tréninku (viz 4.4.1 – Obrazovka spuštěného tréninku).

`DayInPlanRecyclerViewAdapter` zobrazuje dny v tréninkovém plánu a umožňuje editaci daného plánu. V konstruktoru adaptér dostane instanci daného plánu a instanci `IOnStartDragListener`, která slouží pro pohyb prvků v seznamu, viz níže. Editaci adaptér umožňuje pomocí „drag and drop“ systému a pomocí „swipování“. Pro tento účel adaptér implementuje pomocné rozhraní `IItemDragAndSwipeHelperAdapter` (viz 4.5.4), jehož funkce jsou volány při různých akcích s prvky seznamu:

- `OnItemDismiss(int position)` – pro odstranění prvku ze seznamu,

- `OnItemMove(int fromPosition, int toPosition)` – změna pozice prvku v seznamu,
- `OnItemClearView()` – volaná, když uživatel přestane interagovat s prvkem v seznamu.

Funkce `OnItemDismiss, OnItemMove` upravují interní seznam dnů v rámci tréninkového plánu a provolávají funkce z rodičovské třídy pro oznámení změn v seznamu (viz Xamarin Android dokumentace – RecyclerView [30]). Funkce `OnItemClearView` se zavolá když uživatel přestane interagovat s prvky v seznamu a uloží provedené změny do databáze. Pro tento účel potřebuje tento adaptér instanci `IPlanRepository` (viz 4.2.1). Adaptér pro každou položku seznamu nastaví `OnClickListener`, který otevře dialog pro zobrazení daného dne (viz 4.4.2 – Dialogy detailu dne z tréninkového plánu). Také pro každou položku nastaví `OnLongClickListener`, který spustí funkci `OnStartDrag` na instanci `IOnStartDragListener`, kterou adaptér dostal v konstruktoru.

`TrainingInPlanRecyclerViewAdapter` slouží pro zobrazování a editaci tréninků v rámci dne tréninkového plánu. V konstruktoru adaptér dostane instanci daného dne – `PlanDay`, identifikátor plánu – pro ukládání změn do databáze, instance `IOnStartDragListener` – pro implementaci „drag and drop“ systému. Obdobně jako `DayInPlanRecyclerViewAdapter`, implementuje adaptér pomocné rozhraní `IItemDragAndSwipeHelperAdapter` (viz 4.5.4). Tedy funkci pro pohyb prvku v seznamu, funkci pro odstranění prvku ze seznamu a funkci pro ukončení uživatelovy interakce s prvkem – v této funkci obdobně uloží změny provedené na dané instanci `PlanDay`. K tomu používá instanci `IPlanRepository` (viz 4.2.1). Nakonec obdobně jako adaptér pro dny, nastaví pro každou položku seznamu `OnLongClickListener`, který spustí funkci `OnStartDrag` na instanci `IOnStartDragListener`.

Seznam tréninkových aktivit v databázi

Pro zobrazení seznamu aktivit v databázi tréninkových aktivit je použita třída `ListView`, tedy všechny adaptéry zde dědí po `BaseAdapter<T>`, kde T je `Exercise`, `Training` nebo `Plan`. Třídy adaptérů jsou:

- `ExerciseScreenAdapter` – pro cviky,
- `TrainingAdapter` – pro tréninky,
- `TrainingPlanAdapter` – pro tréninkové plány.

Jednotlivé adaptéry se v implementaci neliší, pouze zobrazují jméno daného prvku a v případě cviku ještě obtížnost cviku (viz 6.4). Stejně tak každý adaptér nastaví každému prvku `OnClickListener`, který otevře detail dané tréninkové aktivity (viz 4.4.1 – Detail a editace tréninkové aktivity).

Adaptér zpráv v konverzaci

Zprávy v konverzaci jsou v aplikaci zobrazeny pomocí `ListView`. Adaptérem pro zprávy je `ChatMessageAdapter` dědící po `BaseAdapter<MessageModel>`.

Adaptér v konstruktoru dostane `ChatModel`, což je pouze identifikátor dané konverzace a seznam zpráv `List<MessageModel>`. Adaptér si v konstruktoru zprávy seřadí podle času odeslání. Při zobrazování zpráv pak vizuálně odliší zprávy podle odesílatele a také podle typu zprávy. Pokud zpráva obsahuje tréninkovou aktivitu, místo textu zobrazí tlačítko, které po stisknutí zobrazí dialog s náhledem dané tréninkové aktivity a možností uložení (viz 4.4.2). Otevření dialogu a zobrazení tréninkové aktivity implementují funkce:

- `ShowExerciseDialog`,
- `ShowTrainingDialog`,
- `ShowPlanDialog`.

`ChatShowItemDialog` vyžaduje v konstruktoru callback `onSaveAction`. Adaptér pro každou tréninkovou aktivitu poskytuje pomocí funkcí:

- `GetExerciseSavingAction`,
- `GetTrainingSavingAction`,
- `GetPlanSavingAction`.

Tyto funkce vracejí akce ukládající danou tréninkovou aktivitu k uživateli do soukromé databáze, tedy kopírování z databáze konverzace. K tomu tyto metody využívají příslušné repozitáře (viz 4.2.1) a `ImportHelper` (viz 4.2.2). Nakonec má adaptér pomocné funkce pro načítání tréninkových aktivit z databáze konverzace.

Adaptér cviků editovaného tréninku

`ExerciseInTrainingEditRecyclerViewAdapter` je dalším adaptérem, který je určen pro třídu `RecyclerView` a dědí tedy po `RecyclerView.Adapter`. Adaptér je použit na obrazovce editace tréninku (viz 4.4.1 – Detail a editace tréninkové aktivity), kde je pro úpravu pořadí cviků implementován systém „drag and drop“. Proto je zde použita třída `RecyclerView` ve spojení `RecyclerView.Adapter` a nikoliv `ListView`.

Adaptér v konstruktoru obdrží `IOnStartDragListener`, instanci tréninku – `Training` a seznam cviků které zobrazí – `List<Exercise>`. Adaptér také implementuje rozhraní `IItemDragHelperAdapter` (viz 4.5.4), kde jsou definovány funkce metody pro „drag and drop“ systém a nikoliv pro „swipování“. Využita je ale pouze funkce `OnItemMove`, pomocí které se graficky vymění prvky v seznamu, když je uživatel přetáhne. Funkce `OnItemClearView` zde nemá žádnou funkci. Adaptér dále už jen pro každou položku seznamu nastavuje `OnClickListenery` pro všechny jejich tlačítka (viz 2.8.5). Jsou zde tlačítka pro přetahování položek v seznamu, zdvojení cviku, smazání cviku a tlačítka pro zvýšení, respektive snížení opakování daného cviků. Pouze u tlačítka pro „dragging“, tedy přetahovaní položek v seznamu, je nastaven `SetOnTouchListener`, jemuž je předána nová instance `StartDragOnTouchEvent` (viz 4.5.3), který v konstruktoru dostane `IOnStartDragListener`, předaný v konstruktoru adaptéru.

4.4.4 Toolbar

Toolbar je grafický prvek sloužící pro zlepšení orientace uživatele a pro ukotvení ovládacích prvků na obrazovce. V aplikaci je implementován pouze jako grafický prvek, respektive pouze v `xml`. Nemá tedy žádný zdrojový kód v C#. Jeho zdrojový soubor se nachází ve složce `Resources/layout`. Toolbar je použit ve většině Android aktivit v aplikaci (viz 4.4.1). Ve zdrojovém souboru je v toolbaru definovaná jediná věc – tlačítko nápovědy. Každá aktivita, která toolbar používá si toto tlačítko může použít podle svého nebo ho skrýt. Krom námi definovaného tlačítka nápovědy, toolbar sám o sobě nabízí další funkce, které v aplikaci používáme (viz dokumentace Android – Toolbar [31]). Mezi ně patří především navigační tlačítko (jako tlačítko zpět) a akční menu, které může uživateli nabídnout další akce.

4.5 Helpers

V této podkapitole si popíšeme zbylé netriviální konstrukty, které nezapadají do žádné z ostatních kategorií. Jedná se o pomocné třídy či rozhraní. Všechny se nacházejí ve složce `Helpers`:

- `OnClickListenerWrapper`,
- `IOnStartDragListener`,
- `OnStartDragListenerWrapper`,
- `IItemDragHelperAdapter`,
- `IItemDragAndSwipeHelperAdapter`,
- `StartDragOnLongClickEvent`,
- `StartDragOnTouchEvent`.
- `DragItemTouchHelperCallback`,
- `OnEndAnimationListener`,
- `SwipeAndDragItemTouchHelperCallback`.

4.5.1 OnClickListenerWrapper

Třída `OnClickListenerWrapper` slouží k zabalení akce po kliknutí na grafický prvek, například tlačítko. Grafické prvky totiž vyžadují instanci třídy implementující rozhraní `View.IOnClickListener`. Jedná se tedy o použití návrhového vzoru `adaptér` (viz adapter pattern – Wikipedia [8]). Třída implementuje `View.IOnClickListener` a dědí po třídě `Java.Lang.Object`, která za nás implementuje všechny funkce rozhraní až na `OnClick(View v)`. Třída v konstruktoru dostane delegáta, kterého provolá při kliknutí na dané view.

4.5.2 OnStartDragListenerWrapper

`OnStartDragListenerWrapper` je třída sloužící pro provolání akce při táhnutí nějakého grafického prvku. V aplikaci toho je využito tam, kde je implementován systém „drag and drop“. Třída implementuje rozhraní `IOnStartDragListener`, kde je jediná funkce – `OnStartDrag(RecyclerView.ViewHolder viewHolder)` Třída se implementací velmi podobá třídě `OnClickListenerWrapper`. Nicméně Android nenabízí žádné rozhraní nebo třídu na začátek tažení prvku. Tato třída tedy musí být použita ve spojení s třídou, která rozpozná, že uživatel chce nějaký prvek přetáhnout. Těmito třídami jsou `StartDragOnTouchEvent` a `StartDragOnLongClickEvent` (viz sekce 4.5.3).

4.5.3 StartDrag events

Třídy `StartDragOnTouchEvent` a `StartDragOnLongClickEvent` reprezentují událost, kdy chce uživatel přetáhnout nějaký prvek seznamu. Tento systém v aplikaci implementujeme pomocí dvou gest uživatele. Jedna varianta je na dlouhé podržení na daném prvku – `StartDragOnLongClickEvent`, druhá varianta je, že každý prvek má tlačítko, na které když uživatel klikne, může s prvkem tahat po obrazovce – `StartDragOnTouchEvent`. Obě tyto třídy dědí po `Java.Lang.Object` a dále podle typu implementují rozhraní `IOnTouchListener`, respektive `IOnLongClickListener`. Obě třídy v konstruktoru obdrží instanci třídy implementující `IOnStartDragListener` (viz sekce 4.5.2), jejíž jedinou funkci provolají při vyvolání jejich funkcí `OnLongClick`, respektive `OnTouch`.

4.5.4 IItemDragHelperAdapter

Rozhraní `IItemDragHelperAdapter` definuje funkce pro implementaci systému „drag and drop“ v adaptérových třídách.

```
1 public interface IItemDragHelperAdapter
2 {
3     void OnItemMove(int fromPosition, int toPosition);
4     void OnItemClearView();
5 }
```

Funkce `OnItemMove` reprezentuje moment, kdy uživatel přetáhnul položku z pozice v seznamu `fromPosition` na pozici `toPosition`. Funkce `OnItemClearView` reprezentuje okamžik, kdy uživatel přestane interagovat s prvkem seznamu.

4.5.5 IItemDragAndSwipeHelperAdapter

Rozhraní `IItemDragAndSwipeHelperAdapter` definuje jednu funkci pro implementaci systému „drag and drop“ a „swipování“ v adaptérových třídách a rozšiřuje rozhraní `IItemDragHelperAdapter` (viz sekce 4.5.4).

```
1 public interface IItemDragAndSwipeHelperAdapter : IItemDragHelperAdapter
2 {
3     void OnItemDismiss(int position);
4 }
```

Funkce `OnItemDismiss` reprezentuje moment, kdy uživatel „swipne“ prvek seznamu na pozici `position`.

4.5.6 RecyclerView callbacky

Systémy „drag and drop“ a „swipování“ v aplikaci implementujeme pomocí třídy `ReyclerView`. Tato třída nám poskytuje pomůcku v podobě callbacků, s pomocí kterých můžeme například definovat animace nebo provolávat další funkce. V naší aplikaci tyto funkce implementují třídy `DragItemTouchHelperCallback` a `SwipeAndDragItemTouchHelperCallback`. Obě tyto třídy musí dědit po třídě `RecyclerView.Widget.ItemTouchHelper.Callback`. Tím umožníme, aby v našem `RecyclerView` byly použity jako callbacky.

Tyto třídy již obsahují implementaci systémů „drag and drop“ a „swipování“. Pro jejich fungování se jejich instance spojí s instancí třídy `RecyclerView`, která vše potřebné zařídí, konkrétně:

```
//vytvoříme callback
var callback = new DragItemTouchHelperCallback(adapter);

//vytvoříme instanci ItemTouchHelperu s naším callbackem
var itemTouchHelper = new ItemTouchHelper(callback);

//náš itemTouchHelper připojíme k našemu seznamu cviků
itemTouchHelper.AttachToRecyclerView(exercisesRecyclerView);
```

V našich callback třídách pak např. můžeme povolit/zakázat právě „swipování“:

```
public override bool IsItemViewSwipeEnabled => false;
```

V třídách se pak předefinuje chování těch funkcí/vlastností, které potřebujeme. V našich třídách jsou to pak následující funkce/vlastnosti:

```
//obě třídy
public override bool IsLongPressDragEnabled;
public override bool IsItemViewSwipeEnabled;
public override int GetMovementFlags
    (RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder);
public override bool OnMove
    (RecyclerView recyclerView,
     RecyclerView.ViewHolder viewHolder,
     RecyclerView.ViewHolder target);
public override void OnSwiped(RecyclerView.ViewHolder viewHolder, int p1);
public override void ClearView
    (RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder);

//pouze SwipeAndDragItemTouchHelperCallback
public override void OnChildDraw(Canvas c,
    RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder,
    float dX, float dY, int actionState, bool isCurrentlyActive)
```

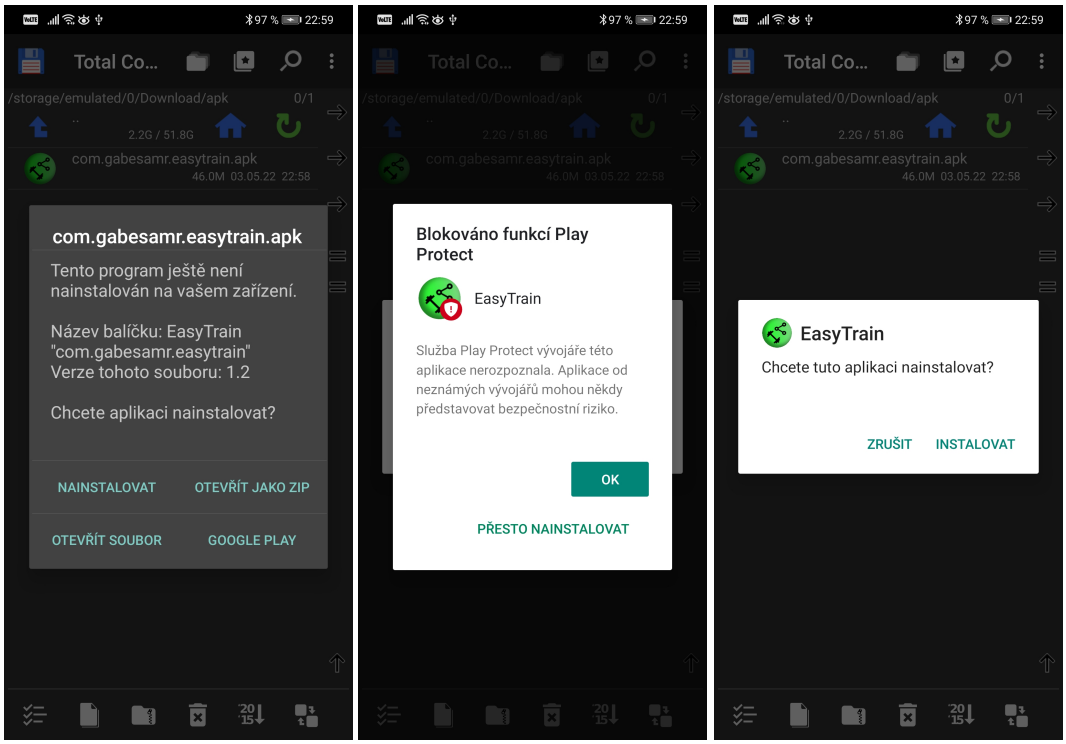
Obě třídy v konstruktoru dostanou odpovídající instanci adaptéru implementující rozhraní `IItemDragAndSwipeHelperAdapter/IItemDragHelperAdapter` (viz sekce 4.5.4, 4.5.5). v předefinovaných funkcích v našich callback třídách pak provoláváme funkce z těchto rozhraní.

5. Uživatelská dokumentace

Aplikace je vytvořena pro mobilní systém Android, nicméně není distribuována přes Google Play ani žádný jiný obchod s aplikacemi. Proto si v následující podkapitole ukážeme jak aplikaci nainstalovat na mobilní zařízení.

5.1 Instalace

V příloze teto práce, se ve složce `install files` a podsložce označující verzi, nachází soubor `com.gabesamr.easytrain.apk`. Tento soubor je potřeba nahrát do mobilu. Po nahrání je potřeba soubor na mobilu najít v libovolné aplikaci na prohlížení souborů (na Obrázku 5.1 je použíta aplikace Total Commander).

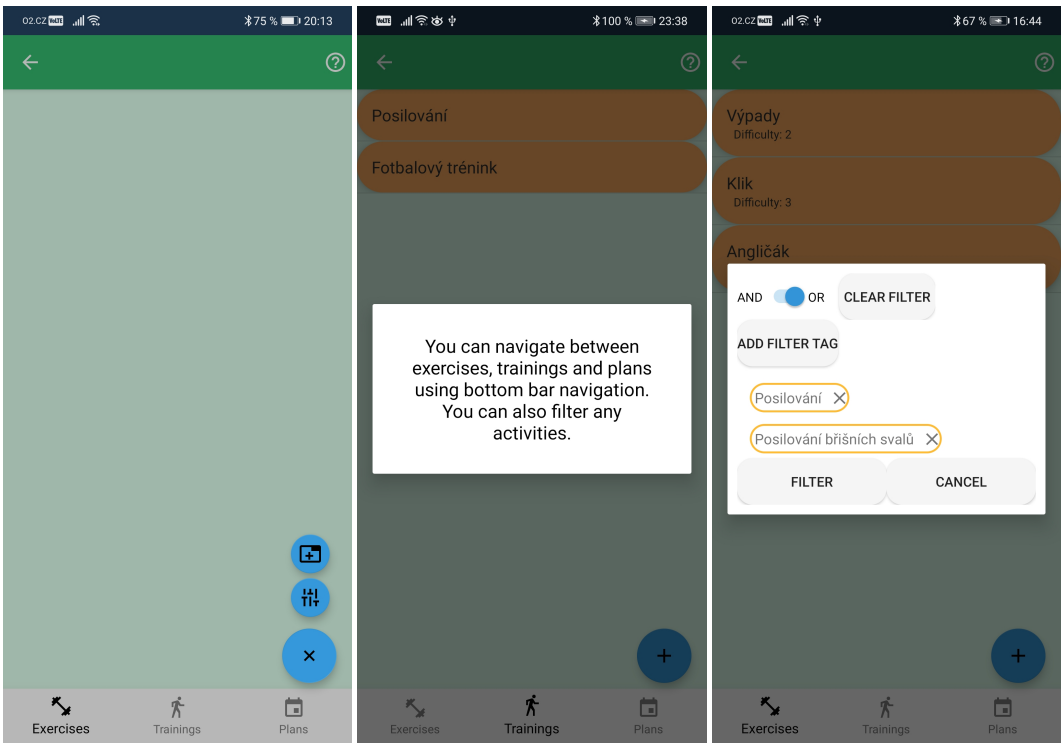


Obrázek 5.1: Průběh instalace

Pro instalaci se pokusíme soubor otevřít a systém již pozná, že jde o soubor pro instalaci aplikace a zeptá se nás, co chceme s daným souborem provést (viz Obrázek 5.1 – vlevo). Toto se může lišit, podle použité aplikace na prohlížení souborů. Pokud zvolíme možnost „instalovat“, ještě se nás zeptá samotný systém Android, zda-li chceme aplikaci nainstalovat (viz Obrázek 5.1 – vpravo). V průběhu instalace či po ní můžou vyvstat další problémy (viz Obrázek 5.1 – uprostřed) s důvěryhodností naší aplikace. Všechny tyto upozornění apod. je třeba ignorovat. Po úspěšné instalaci se aplikace bude nacházet nainstalovaná mezi ostatními aplikacemi pod názvem „EasyTrain“.

5.2 Základní práce s databází

Základním kamenem naší aplikace je databáze tréninkových aktivit. Abychom se do ní dostali, použijeme na hlavní obrazovce tlačítko „YOUR DATABASE“ nebo v menu vybereme „Go train“. V databázi se mezi seznamy cviků, tréninků a tréninkových plánů navigujeme pomocí spodní lišty (viz Obrázek 5.2).



Obrázek 5.2: Prvky databáze

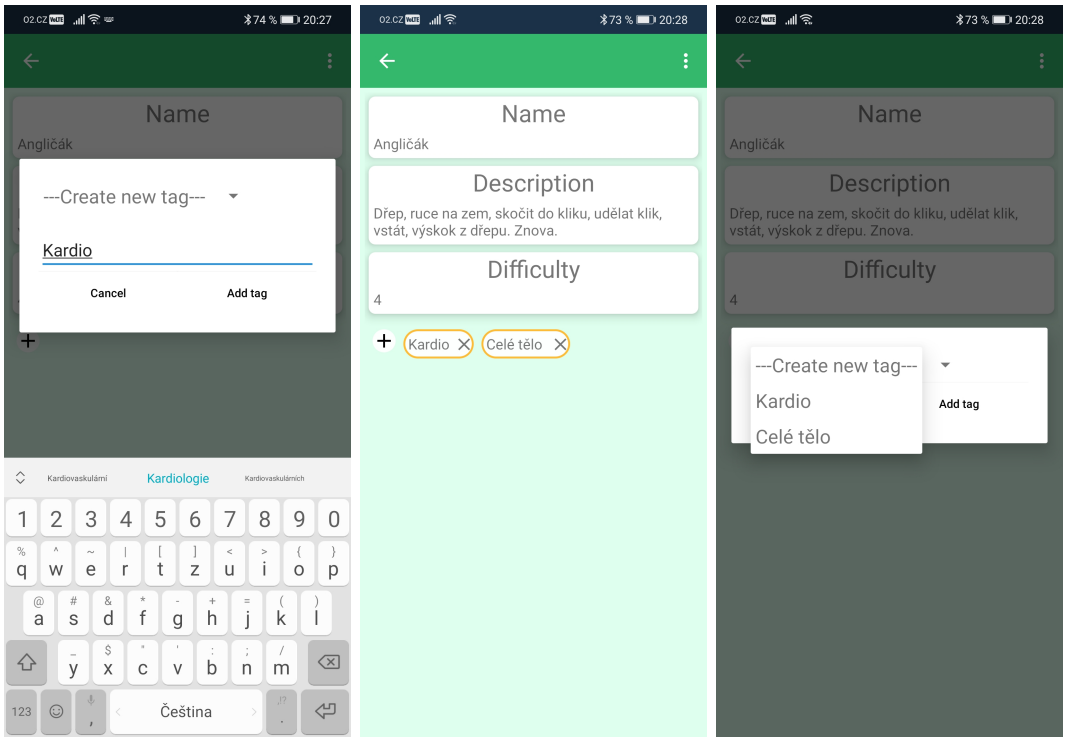
Pokud chceme vytvořit nový záznam v databázi, zvolíme příslušný typ na spodní liště, klikneme na modré tlačítko „+“ a v rozbalené nabídce vybereme horní tlačítko s ikonkou souboru (viz Obrázek 5.2 – vlevo). To nás přesměruje na stránku pro vytvoření příslušné tréninkové aktivity.

Nakonec lze všechny záznamy v seznamech i filtrovat. Při filtraci postupujeme stejně jako když chceme přidat nový záznam, ale v rozbalené nabídce klikneme na prostřední tlačítko s ikonkou „ladění“. To otevře nové okno, kde můžeme filtr upravovat(viz Obrázek 5.2 – vpravo). Můžeme filtr vymazat – tlačítko „CLEAR FILTER“, přidat další tag do filtru – tlačítko „ADD FILTER TAG“, aplikovat filtr, či zrušit filtr pomocí tlačítek „FILTER“ a „CANCEL“. Ještě je zde přepínač, který rozhoduje jaký typ filtru se aplikuje. V případě cviků, „AND“ znamená, že každý vyfiltrovaný cvik musí obsahovat **všechny** tagy z filtru a „OR“ znamená, že každý vyfiltrovaný cvik musí obsahovat **alespoň jeden** z vybraných tagů z filtru. V případě tréninků a tréninkových plánů se filtr aplikuje tak, že v celé dané aktivitě musí být alespoň jeden cvik splňující daný filtr.

5.2.1 Cvik

Cvik obsahuje povinné položky: jméno, obtížnost a volitelný popisek. Obtížnost je celé číslo od 1 do 10. Pokud bychom chtěli cvik po jeho vytvoření editovat

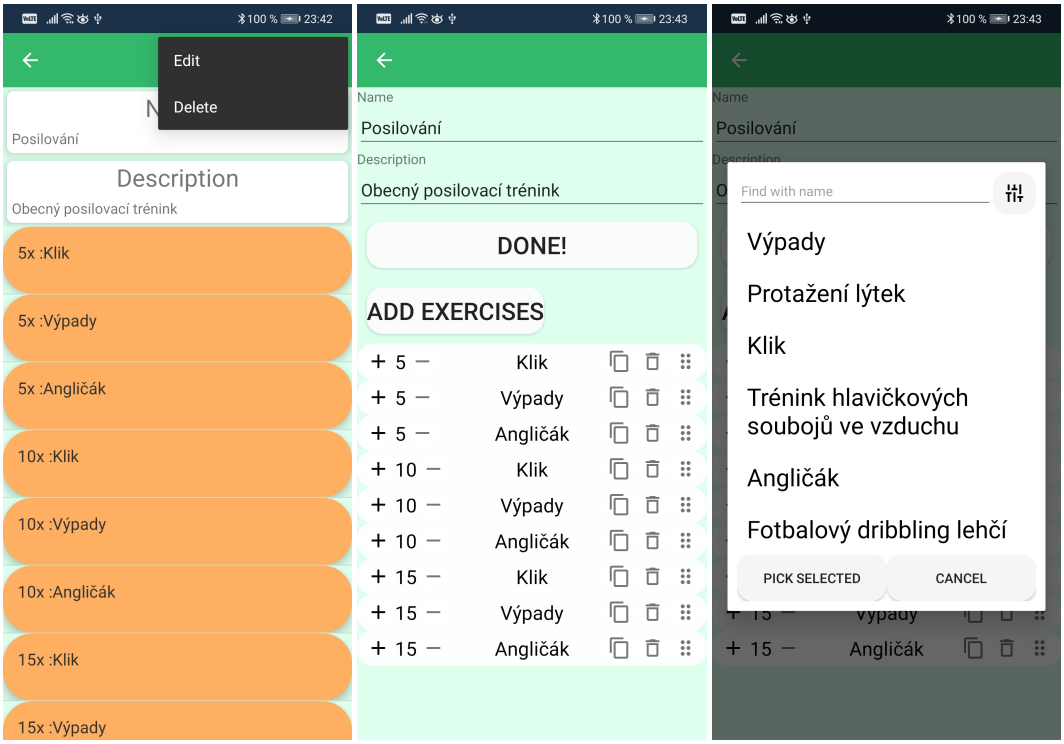
nebo smazat, na obrazovce cviku klikneme na ikonku tří teček v pravém horním rohu obrazovky, kde se nám zobrazí nabídka „edit“ a „delete“. K cviku lze navíc přiřazovat tagy. To lze na obrazovce po rozkliknutí detailu nějakého cviku (viz Obrázek 5.3 – uprostřed). Po kliknutí na tlačítko s „+“ pod detaily cviku se zobrazí okno pro přidání tagu k daném cviku. Uživatel může tag vybrat z již existujícího seznamu (viz Obrázek 5.3 – vpravo) nebo vytvořit úplně nový tag (viz Obrázek 5.3 – vlevo). Pokud bychom chtěli nějaký tag odebrat, stačí kliknout na ikonku křížku u příslušného tagu.



Obrázek 5.3: Cvik v detailu

5.2.2 Trénink

Trénink obsahuje povinné jméno a volitelný popis. Dále se do něj dají skládat cviky v libovolném množství, pořadí a libovolně se opakující. Stejně jako cvik, lze trénink editovat nebo smazat po kliknutí na příslušné tlačítko v menu v pravém horním rohu obrazovky s detailem tréninku. Kromě editace jména a popisu tréninku, lze především měnit jeho obsah. Tlačítkem „ADD EXERCISES“ se otevře nové okno (viz Obrázek 5.4 – vpravo), kde vybereme cvik, který přidáme do tréninku. Poku chceme nějaký cvik z tréninku zdvojit, můžeme kliknout na první tlačítko vpravo od jména cviku (viz Obrázek 5.4 – uprostřed). Stejně tak můžeme cvik odstranit tlačítkem s ikonkou popelnice. Odstranění způsobí, že se množství daného cviku (vlevo od jména cviku), nastaví na nulu. Cvik se úplně odstraní až po uložení pomocí tlačítka „DONE“. Nakonec pořadí cviků lze měnit. Pokud se dotkneme tlačítka s ikonkou šesti teček, můžeme položku v seznamu libovolně přetahovat.

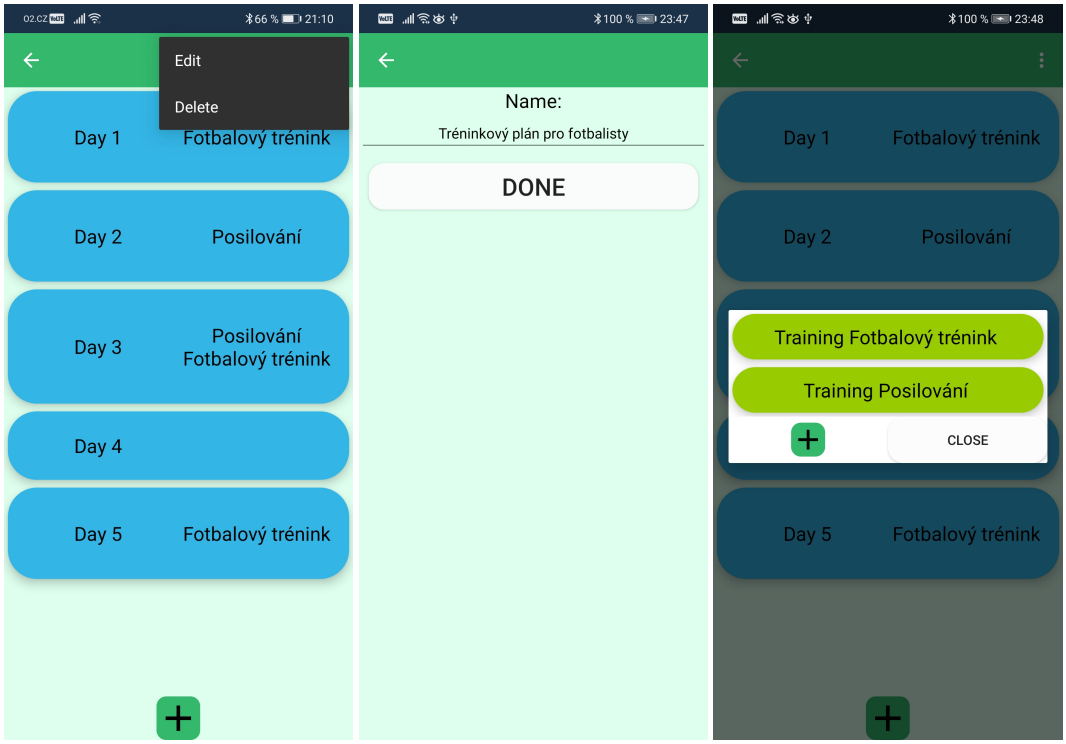


Obrázek 5.4: Trénink v detailu

5.2.3 Tréninkový plán

Poslední tréninkovou aktivitou je tréninkový plán. Stejně jako cvik a trénink i plán má svoji editační obrazovku (viz Obrázek 5.5 – uprostřed), ke které se dostaneme po kliknutí na tlačítko „Edit“. Nicméně zde lze editovat pouze jméno daného plánu. Editovat samotný obsah plánu lze už na obrazovce jeho detailu. Změny, které zde provedeme, se ukládají průběžně (viz Obrázek 5.5 – vlevo).

Tréninkový plán se skládá z dnů, které jsou zobrazeny modrými bloky. V každém dnu pak může být libovolný počet tréninků, tedy i žádný. Den do plánu přidáme kliknutím na zelené tlačítko s ikonkou plusu na spodku obrazovky. Kliknutím na den, pak otevřeme jeho detail (viz Obrázek 5.5 – vpravo), kde můžeme editovat tréninky daného dne. Také zde trénink přidáme zeleným tlačítkem na spodku okna. Nakonec dny i tréninky lze mazat nebo měnit jejich pořadí. Pokud chceme změnit pořadí, podržíme na dané položce prst a po chvíli ji můžeme přetáhnout na libovolnou pozici. Pokud chceme prvek odstranit, „swipneme“ prvek na libovolnou stranu.

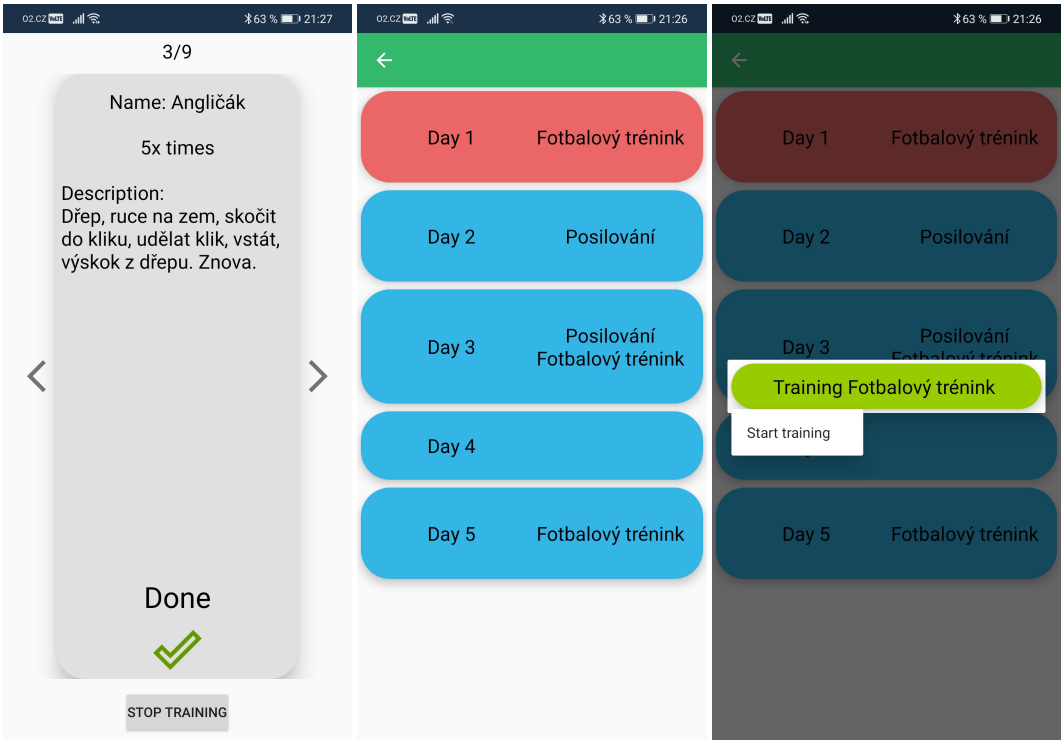


Obrázek 5.5: Tréninkový plán v detailu

5.3 Spouštění tréninku a tréninkového plánu

Aplikace poskytuje funkci spuštěného tréninku a spuštěného tréninkového plánu. Na obrazovce spuštěného tréninku (viz Obrázek 5.6 – vlevo) se ukazují jednotlivé cviky z tréninku s jejich popisky a četnostmi v tom pořadí, v jakém jsou v daném tréninku poskládány. Mezi jednotlivými položkami můžeme libovolně přecházet a označovat je za splněné. Trénink můžeme spustit kliknutím na tlačítko „Start training“ v hlavním menu anebo na „WORK OUT NOW“ na hlavní obrazovce.

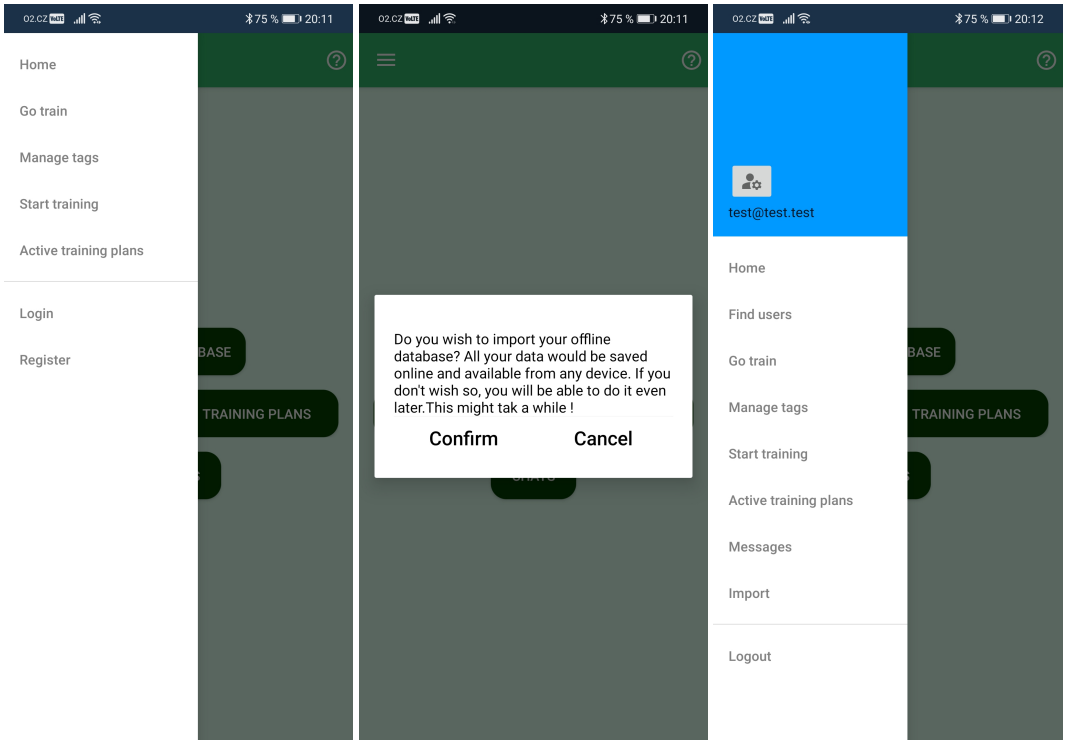
Tréninkových plánů můžeme mít i více najednou. Proto je v aplikaci obrazovka se seznamem aktivních/spuštěných tréninkových plánů. Na tu se lze dostat tlačítkem „Active training plans“ v hlavním menu anebo kliknutím na tlačítko „TRAINING PLANS“ na hlavní obrazovce. Spuštěné tréninkové plány, které celé proběhnou se sami smažou. Po rozkliknutí aktivního plánu se uživateli ukazuje, ve kterém dni tréninkového plánu se právě nachází. Tento den je v tréninkovém plánu zvýrazněn červenou barvou oproti ostatním dnům, které jsou vybarveny modře (viz Obrázek 5.6 – uprostřed). Po rozkliknutí jednoho dne tréninkového plánu, se zobrazí okno s tréninky daného dne, dle jejich pořadí (viz Obrázek 5.6 – vpravo). Navíc po kliknutí na trénink se uživateli nabídne možnost daný trénink spustit.



Obrázek 5.6: Spuštěný trénink nebo tréninkový plán

5.4 Registrace – přihlášení

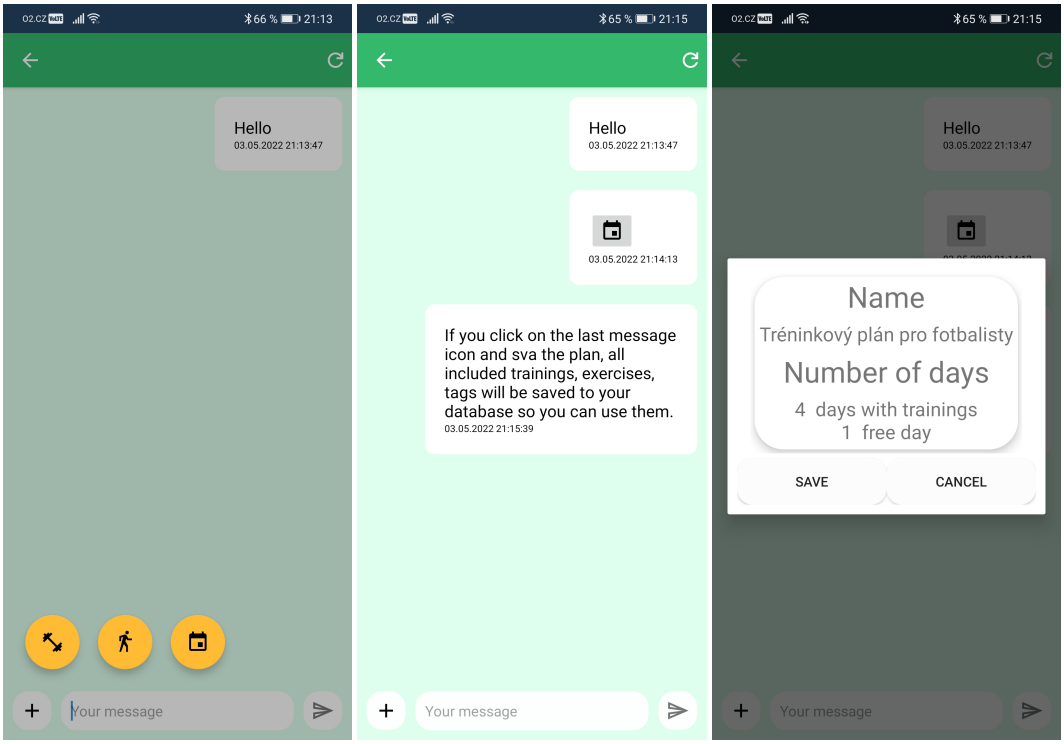
Aplikace umožňuje její používání bez vytvoření účtu v aplikaci, nicméně uživatel tím přichází o některé funkce. Například nemůže sdílet svoje tréninkové aktivity s ostatními uživateli (viz Obrázek 5.7 – vlevo odhlášený uživatel, vpravo přihlášený uživatel) Pokud uživatel není přihlášený, používá se pro všechna data uložiště v telefonu. Pokud je uživatel přihlášený, všechny jeho data se ukládají do online uložiště a tedy jsou dostupná z každého zařízení. Je důležité zmínit, že pokud nejdříve používáme aplikaci bez přihlášení a později si vytvoříme účet, data která jsme si do té doby vytvořili, nebudou z daného účtu našeho účtu dostupná v databázi. Můžeme si je ovšem importovat z paměti do telefonu do online uložiště v záložce „Import“ (viz Obrázek 5.7 – vpravo, druhé tlačítko odspodu). Zároveň, okamžitě po registraci a jen tehdy, se uživateli nabídne automatický převod všech dat z paměti telefonu do online databáze (viz Obrázek 5.7 – uprostřed).



Obrázek 5.7: Anonymní a přihlášený uživatel

5.5 Chat a sdílení tréninků

Pro přihlášené uživatele aplikace nabízí konverzaci s libovolným dalším uživatelem. Pokud chceme najít konkrétního uživatele, můžeme tak učinit v záložce „Find users“. Započaté konverzace pak nalezneme v záložce „Messages“. V rámci konverzace je možné posílat textové zprávy, ale především libovolnou tréninkovou aktivitu (viz Obrázek 5.8 – vlevo – ikonky na spodku obrazovky). Zároveň pokud pošleme například trénink, pošlou se zároveň s tím i všechny cviky, které jsou v tréninku obsažené. Stejně tak tomu je s tréninkovými plány. Tréninková aktivita se v konverzaci zobrazí pomocí ikonky (viz Obrázek 5.8 – uprostřed – prostřední zpráva). Pokud chceme nějakou poslanou tréninkovou aktivitu přijmout, klikneme na její ikonku v konverzaci. To nám zobrazí okno se náhledem dané aktivity a s tlačítkem na její uložení (viz Obrázek 5.8 – vpravo). Zde je důležité zmínit, že pokud stejnou aktivitu uložíme víckrát, bude i víckrát v naší databázi, včetně všech dalších aktivit, které jsou v ni obsaženy. Aplikace momentálně nepodporuje notifikace, tedy se nedozvíme, že nám přišla nová zpráva a zároveň se konverzace ani v reálném čase neobnovuje. Pro její obnovení a načtení nových zpráv je třeba použít tlačítko v pravém horním rohu.



Obrázek 5.8: Konverzace s jiným uživatelem

5.6 Ukázková data

Pro účely této bakalářské práce jsme připravili v aplikaci testovací účet (viz přihlašovací údaje níže). Tento účet má jíž nějaká připravená data v databázi, která si čtenář může prohlédnout, případně si zkusit něco upravit, přidat či smazat nebo spustit nějaký trénink.

```
e-mail: bakalarka@test.cz
heslo: bakalarkaheslo
```

V případě, že by při čtenář data nějak změnil a chtěl by původní připravená data obnovit, může tak učinit. V sekci s konverzacemi je třeba nalézt konverzaci s uživatelem *René Gabesam*. První zpráva této konverzace je poslaný tréninkový plán, obsahující všechna připravená data. Po uložení tohoto tréninkového plánu budou v databázi stejná data jaká jsme připravili. Před tímto krokem doporučujeme smazat zbývající tréninkové aktivity z databáze, aby po uložení tréninkového plánu nevznikli duplicity.

6. Závěr

Na závěr této práce zhodnotíme kvality naší aplikace. Shrneme zde pár uživatelských zkušeností (viz podkapitola 6.1). Zhodnotíme, jestli a jak dobře se nám podařilo splnit naše cíle (viz podkapitola 6.2). Dále si shrneme a zhodnotíme rozhodnutí týkající se výběru online databáze (viz podkapitola6.3). Nakonec zmíníme několik směrů, kterými by se aplikace mohla v budoucnu ubírat (viz podkapitola 6.4).

6.1 Uživatelské zkušenosti

V průběhu vývoje jsme naši aplikaci zkusili poskytnout několika lidem, aby ji zběžně otestovali a zhodnotili. Někteří uživatelé nám nahlásili chyby, které se nám vždy podařilo odstranit. Narazili jsme také na problémy související s tím, že uživatel používal aplikaci jinak než bylo zamýšleno nebo na neočekávaných datech. Většinu problémů se nám podařilo vyřešit, ale například nás nenapadlo, že by někteří uživatelé chtěli aplikaci používat v režimu na šířku. S tím ale zatím nepočítáme, a proto jsme prozatím otáčení aplikace zakázali.

Někteří uživatelé aplikaci komentovali s tím, že očekávali již nějakou množinu hotových tréninkových aktivit. To jistě pramení z toho, že velká část podobných aplikací něco takového nabízí a dokonce to prodává jako svoje „know how“ (viz podkapitola 1.5). Pro naší aplikace toto jistě přichází v úvahu (viz podkapitola 6.4), ale není to prvotní záměr naší aplikace.

6.2 Zhodnocení cílů

V této podkapitole zhodnotíme cíle práce, které jsme si na začátku definovali v podkapitole 1.7.

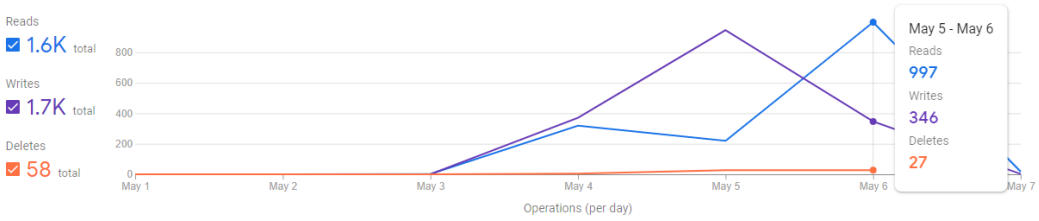
Cíle 1 a 2 jsme splnili. Výsledkem práce je aplikace pro operační systém Android, dokonce s minimální verzí systému `7.1 - Nougat`. Zároveň aplikace používá pouze technologie, které jsou v našem rozsahu zdarma.

Cíl č. 3 nám zadává aby aplikace kopírovala trendy dnešních aplikací. Struktura aplikace je členěna jednoduše, s jedním hlavním menu, po vzoru jiných úspěšných aplikací. Používáme v aplikaci grafické prvky a prvky ovládání, které taktéž dnes používají úspěšné aplikace. Na druhé straně celkový grafický dojem z aplikace není na takové úrovni, jako jiné aplikace podobného typu. Toto vyplývá pravděpodobně z toho, že při implementaci aplikace jsme se soustředili především na její funkčnost. Domníváme se tedy, že cíl 3 jsme z části naplnili, nicméně je zde prostor na zlepšení grafické stylizaci ovládacích prvků.

Poslední cíl č. 4 nám zadává splnění funkčních požadavků. Aplikace všechny funkční požadavky splňuje, tedy tento cíl jsme splnili. Práce tedy splňuje všechny požadavky, přestože jsou zde rezervy v grafickém návrhu aplikace.

6.3 Zhodnocení online databáze

Po dokončení práce a otestování na několika uživatelích se zpětně zamýšlíme nad volbou online databázové technologie a nad implementací online databázové vrstvy. Při uživatelském testování se totiž ukázalo, že už při jednotkách aktivních uživatelů, se počet čtení z databáze pohybuje v řádech tisíců a stejně tomu je s počtem zápisů (viz Obrázek 6.1). Nezpoplatněné použití Firestore databáze je omezeno na 50 000 čtení a 20 000 zápisů za den (viz dokumentace Firebase – pricing [4]). V případě, že bychom chtěli mít větší kapacitu co se počtu čtení a zápisu týče, bylo by potřeba za databázi již platit.



Obrázek 6.1: Využití online databáze

Pokud bychom chtěli aplikaci vypustit na trh, bylo by potřeba zvážit výdělečnost aplikace v porovnání s tím, co bychom museli zaplatit za její provoz. Druhá varianta by byla najít jinou online databázi, nicméně pro větší objemy dat jsme nenašli žádnou databázi, která by byla zdarma ve větších objemech dat. U Firebase Firestoru máme navíc výhodu v podobě správy uživatelů (viz sekce 2.3.2). Kdybychom tak ale chtěli učinit, databázová vrstva nám to velice dobře umožňuje, poněvadž chod aplikace je úplně oddělen a nezávislý na její implementaci.

Třetí možností je pokusit se optimalizovat množství operací prováděných s databází. Pokud by se nám to povedlo, umožnilo by nám to mít, v neplacené verzi databáze, více aktivních uživatelů. Platbám za databázi bychom se ale nevyhnuli. Pokud bychom u Firestoru zůstali, byla by tato optimalizace stejně na místě, protože cena za provoz databáze je přímo úměrná počtu čtení/zápisů a dalším operacím (viz dokumentace Firebase – pricing [4]).

6.4 Místo pro rozšíření

Naše aplikace, jako výsledek této práce, rozhodně není vyčerpávající možnou funkcionalitu. Nabízí se velké množství potenciálních rozšíření. Jedním z nich jsou již zmíněné předpřipravené tréninkové aktivity, po vzoru jiných aplikací. Zde je otázka, pro jaký sport bychom něco takového měli udělat.

Další možná rozšíření se týkají atributů tréninkových aktivit. To, jaké atributy jsme si pro tréninkové definovali (u cviku a tréninku jsou to pouze jméno a popisek) rozhodně není vyčerpávající. V aplikaci jsme rovnou rozšířili cvik o číselné hodnocení obtížnosti. U cviků i tréninků se například nabízí časová náročnost nebo seznam potřebných pomůcek. Další možnost jsou přílohy ve formě obrázků nebo dokonce videa. To by ale již kladlo větší nároky na databázi.

Další úpravy aplikace by se určitě týkaly již zmíněné optimalizace databáze. Pokud by aplikace byla v reálném provozu a získala s časem více uživatelů, museli

bychom platit za provoz databáze. Tím by se nám otevřela možnost v podobě cloudových funkci Firestore (viz dokumentace Firebase – pricing [4]), která by nám umožňovala operovat s databází na serveru a to by nám umožnilo velmi snadno implementovat notifikace na příchozí zprávu v konverzaci. To je funkce, kterou každá dnešní aplikace obsahující konverzace má. Bez cloudových funkcí bychom toho byly taky schopni, nicméně znamenalo by to pravidelné dotazy do databáze z telefonního zařízení a v závislosti na pravidelnosti těchto dotazů by se zvedal i počet čtení z databáze, což je pravý opak toho co bychom chtěli. Navíc by aplikace musela zůstávat běžet na pozadí, což také není typicky žádoucí.

Poslední potenciální úpravy by se jistě týkali grafického designu a obecného „user experience“ (uživatelského prožitku). Především co se ovládacích prvků týče, je v aplikaci jistě spousta možností na zlepšení. Například hromadné přidávání cviků do tréninků nebo hromadné mazání položek ze seznamu. Další možností jsou různé prokliky napříč tréninkovými aktivitami nebo jejich spouštění apod. Co se designu aplikace týká, vzhledem k vývoji designu mobilních aplikací v čase, lze říci, že nikdy nebude dokonalý. Aplikace, která chce mít v dnešní době dostatek uživatelů, se jistě musí držet trendů právě v oblasti designu.

Seznam použité literatury

[1] Statcounter. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2021. Accessed: 2021-12-19.

[2] Statcounter. Mobile Android Version Market Share Worldwide. https://gs.statcounter.com/android-version-market-share/mobile/worldwide/, 2021. Accessed: 2021-12-19.

[3] Wikipedia. SQL. https://cs.wikipedia.org/wiki/SQL, 2021. Accessed: 2021-12-25.

[4] Google. Pricing plans. https://firebase.google.com/pricing, 2021. Accessed: 2021-12-19.

[5] Google. Firebase Documentation. https://firebase.google.com/docs/, 2021. Accessed: 2021-12-19.

[6] Wikipedia. SQLite. https://cs.wikipedia.org/wiki/SQLite, 2021. Accessed: 2021-12-31.

[7] Wikipedia. Normalizace databáze. https://cs.wikipedia.org/wiki/Normalizace_datab%C3%A1ze, 2021. Accessed: 2021-12-25.

[8] Wikipedia. Adapter pattern. https://en.wikipedia.org/wiki/Adapter_pattern, 2021. Accessed: 2021-12-31.

[9] Microsoft. LINQ (Language Integrated Query). https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/concepts/linq/, 2022. Accessed: 2022-01-05.

[10] Microsoft. Attributes. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/, 2022. Accessed: 2022-01-05.

[11] Microsoft. Xamarin.Android Spinner. https://docs.microsoft.com/cs-cz/xamarin/android/user-interface/controls/spinner, 2021. Accessed: 2021-12-26.

[12] Microsoft. Xamarin.Android PopUp Menu. https://docs.microsoft.com/en-us/xamarin/android/user-interface/controls/popup-menu, 2021. Accessed: 2021-12-26.

[13] Google. FlexboxLayout. https://github.com/google/flexbox-layout, 2021. Accessed: 2021-12-27.

[14] Wikipedia. Drag and drop. https://cs.wikipedia.org/wiki/Drag_and_drop, 2021. Accessed: 2021-12-27.

[15] Google. Supported platforms. https://cloud.google.com/dotnet/docs/reference/help/platforms, 2022. Accessed: 2022-01-01.

[16] Microsoft. Xamarin.Firebase.Firestore. https://www.nuget.org/packages/Xamarin.Firebase.Firestore/, 2022. Accessed: 2022-01-01.

[17] Google. com.google.android.gms.tasks. https://developers.google.com/android/reference/com/google/android/gms/tasks/package-summary, 2022. Accessed: 2022-01-01.

[18] Damir's Corner. Awaiting Google Play services tasks. https://www.damirscorner.com/blog/posts/20210416-AwaitingGooglePlayServicesTasks.html, 2022. Accessed: 2022-01-02.

[19] Microsoft. TaskCompletionSource. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1?WT.mc_id=DOP-MVP-4039670&view=net-6.0, 2022. Accessed: 2022-01-13.

[20] Microsoft. Asynchronous programming. https://docs.microsoft.com/en-us/dotnet/csharp/async, 2021. Accessed: 2021-12-26.

[21] Wikipedia. Loose coupling. https://en.wikipedia.org/wiki/Loose_coupling, 2022. Accessed: 2022-01-02.

[22] Microsoft. Delegates. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/, 2022. Accessed: 2022-01-02.

[23] Microsoft. Lambda expressions. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions, 2022. Accessed: 2022-01-02.

[24] Microsoft. Xamarin.Firebase.Auth. https://www.nuget.org/packages/Xamarin.Firebase.Auth/, 2022. Accessed: 2022-01-02.

[25] Wikipedia. Singleton. https://cs.wikipedia.org/wiki/Singleton, 2022. Accessed: 2022-01-16.

[26] Microsoft. Android Resource Basics. https://docs.microsoft.com/en-us/xamarin/android/app-fundamentals/resources-in-android/android-resource-basics?tabs=windows, 2022. Accessed: 2022-03-09.

[27] Microsoft. Intent Class. https://docs.microsoft.com/en-us/dotnet/api/android.content.intent?view=xamarin-android-sdk-12, 2022. Accessed: 2022-03-24.

[28] Microsoft. Creating A Fragment. https://docs.microsoft.com/en-us/xamarin/android/platform/fragments/creating-a-fragment, 2022. Accessed: 2022-03-09.

[29] Nuget. XAB FlexboxLayout. https://www.nuget.org/packages/XAB.FlexboxLayout/, 2022. Accessed: 2022-03-30.

[30] Microsoft. RecyclerView Parts and Functionality. https://docs.microsoft.com/en-us/xamarin/android/user-interface/layouts/recycler-view/parts-and-functionality, 2022. Accessed: 2022-04-26.

[31] Android. Toolbar. https://developer.android.com/reference/androidx/appcompat/widget/Toolbar, 2022. Accessed: 2022-04-27.

Přílohy

Přehled elektronických příloh

```
.
├── source - adresář s kompletním projektem aplikace
│   ├── EasyTrain - adresář se všemi zdrojovými soubory aplikace
│   └── EasyTrain.sln - solution pro Visual Studio
├── screenshots - adresář se screenshoty z aplikace
├── install files - soubor .apk pro instalaci na Android zařízení
│   └── v1.x - podsložky udávající verzi
├── tex - adresář s LaTeX zdrojovými soubory této práce
│   └── img - obrázky použité v práci ve formátu PDF/A
├── prace.pdf - text této práce ve formátu PDF/A
└── README.txt - úvodní soubor obsahující kontakt na autora
```