Mobilní aplikace s využitím návrhových vzorů a realtime databází pro Android

Bc. Dominik Janíček

Diplomová práce
2018

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2017/2018

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	Bc. Dominik Janíček
Osobní číslo:	A16134
Studijní program:	N3902 Inženýrská informatika
Studijní obor:	Informační technologie
Forma studia:	prezenční
Téma práce:	Mobilní aplikace s využitím návrhových vzorů a realtime databází pro Android
Téma anglicky:	A Mobile Application Using Design Patterns and Realtime Data-bases for Android

Zásady pro vypracování:

1. Prostudujte a popište moderní architektury pro vývoj mobilních aplikací pro platformu Android (MVC, MVP, MVVM).
2. Prozkoumejte možnosti použití realtime databází v prostředí aplikací OS Android.
3. Porovnejte realtime databáze s obvyklým přístupem sdílení informací mezi klientem a serverem.
4. Demonstrujte možnost nasazení realtime databáze na příkladové aplikaci pro Android.
5. Shrňte funkční a nefunkční požadavky na demonstrační aplikaci.
6. Dle požadavků navrhněte a implementujte aplikaci a popište její architekturu a napojení na realtime databázi.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: tištěná/elektronická

Seznam odborné literatury:

1. LACKO, Ľuboslav. Mistrovství – Android. Brno: Computer Press, 2017. Mistrovství. ISBN 978-80-251-4875-4.
2. JACKSON, Wallace. Android apps for absolute beginners: covering Android 7. New York, NY: Springer Science Business Media, 2017. ISBN 978-1-4842-2267-6.
3. LACKO, Ľuboslav. Vývoj aplikací pro Android. Brno: Computer Press, 2015. ISBN 978-80-251-4347-6.
4. PECINOVSKÝ, Rudolf. Návrhové vzory: [33 vzorových postupů pro objektové programování]. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.
5. HILLS, Ted. NoSQL and SQL data modeling. Basking Ridge, NJ: Technics Publications, 2016. ISBN 978-163-4621-090.
6. DEKA, Ganesh Chandra. NoSQL: database for storage and retrieval of data in cloud. Boca Raton, FL: Chapman and Hall/CRC, 2017. ISBN 978-149-8784-368.

Vedoucí diplomové práce: Ing. Radek Vala, Ph.D.
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: 1. prosince 2017

Termín odevzdání diplomové práce: 16. května 2018

Ve Zlíně dne 11. prosince 2017

doc. Mgr. Milan Adámek, Ph.D.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

prof. Mgr. Roman Jašek, Ph.D.
garant oboru

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 15. 5. 2018

.....
podpis diplomanta

ABSTRAKT

Diplomová práce se zabývá architektonickými návrhovými vzory a databázemi pracujícími v reálném čase v rámci platformy Android. Hlavním cílem je srovnání příslušných architektonických návrhových vzorů a databází pracujících v reálném čase a shrnutí poznatků. Praktickým výstupem práce je mobilní aplikace, která demonstruje nabyté poznatky. Tato aplikace je vytvořena dle funkčních a nefunkčních požadavků specifikovaných v praktické části. Dále také pomocí zvoleného architektonického návrhového vzoru a vyhovující databáze. V teoretické části je stručně popsán systém Android, architektonické návrhové vzory a rozdíl mezi klasickou relační databází a databází pracující v reálném čase spolu s jejími představiteli. Praktická část se věnuje popisu rozdílů mezi jednotlivými architektonickými návrhovými vzory a taktéž rozdíly mezi představiteli databází pracujícími v reálném čase.

Klíčová slova: Android, Architektonické návrhové vzory, Databáze reálného času, Mobilní aplikace, Firebase, Firestore, Realm

ABSTRACT

This master thesis deals with design patterns and real time databases within the Android platform. The main goal is to compare design patterns, compare real-time databases and summarize the findings. The practical output of the thesis is a mobile application that demonstrates acquired findings. The application is created according to functional and non-functional requirements, and selected design pattern and we use the most suitable database. In the theoretical part is briefly described the Android system, architecture design patterns and the difference between classic relational databases and real time databases. Their representatives are also discussed and compared. The practical part is about describing the difference between the individual architecture design patterns and the differences between the representatives of the real time database.

Keywords: Android, Architecture design pattern, Real-time database, Mobile application, Firebase, Firestore, Realm

Týmto by som chcel poďakovať svojmu vedúcemu práce Ing. Radkovi Valovi, Ph.D. za jeho rady a pripomienky počas vypracovávania tejto práce a taktiež za jeho ústretovosť.

Ďalej by som chcel poďakovať Ing. Radovanovi Holíkovi za odborné konzultácie v oblasti programovania mobilnej aplikácie pre Android a pripomienky počas písania tejto práce.

OBSAH

ÚVOD

V súčasnosti sú veľmi populárne a rozšírené mobilné zariadenia ako smartphony a tablety využívajúce prevažne systém Android. Preto sa veľká časť vývoju aplikácií venuje vývoju pre tento typ zariadenia a počet mobilných aplikácií narastá. Zvyšuje sa aj ich zložitosť, a aj nároky zo strany užívateľov. Pre prehľadnosť a udržateľnosť kóde je potom prakticky nevyhnuté použitie návrhového (architektonického) vzoru.

Mobilné aplikáciu sú taktiež čím ďalej, tým viac závislé od zdieľania rôznych informácií cez internet a ich ukladanie na vzdialený server. Na to sú prevažne používané tzv. databázy reálneho času (real-time). Veľa prevažne neskúsených vývojárov nevie aké technológie a postupy je teda vhodné použiť, a to bol aj dôvod vzniku tejto práce.

V teoretickej časti je na začiatku popísaný systém Android, jeho architektúra percentuálny podiel jednotlivých verzií na trhu a taktiež aktivita (Activity).

Ďalej sú v tejto práci popísané moderné návrhové vzory pre vývoj softwaru ako MVC, MVP a MVVM a ich jednotlivé časti. Taktiež je tu popísaný DataBinding a čo sú to Live-Data.

Posledná kapitola teoretickej časti sa zaoberá porovnávaním relačnej databázy s databázou reálneho času a princípom fungovania databázy reálneho času. Koniec kapitoly je venovaný predstaviteľom databáz reálneho času ako sú Firebase, Firestore a Realm a ich vlastnostiam.

Praktická časť sa venuje porovnávaniu návrhových vzorov popísaných v teoretickej časti, hlavnými rozdielmi, výhodami a nevýhodami jednotlivých návrhových vzorov. Na konci kapitoly sú zhrnuté dôležité zistenia.

Ďalšia kapitola sa venuje porovnávaniu databáz reálneho času (Firebase, Firestore, Realm), kde sú porovnané vlastnosti týchto databáz a na konci kapitoly je zhrnutie týchto vlastností zobrazených v tabuľke .

V druhej časti praktickej časti práce je popísaná demonštračná aplikácia, jej funkčné a nefunkčné požiadavky a taktiež diagram prípadov použitia.

V poslednej kapitole je opísaná aj samotná aplikácia, a to ako z pohľadu užívateľa, kde sú popísané jednotlivé obrazovky, tak aj z pohľadu vývojára. A to popis tvorby jednotlivých častí aplikácie ako prihlásenie, správa databáze, bindovanie dát a iné.

I. TEORETICKÁ ČASŤ

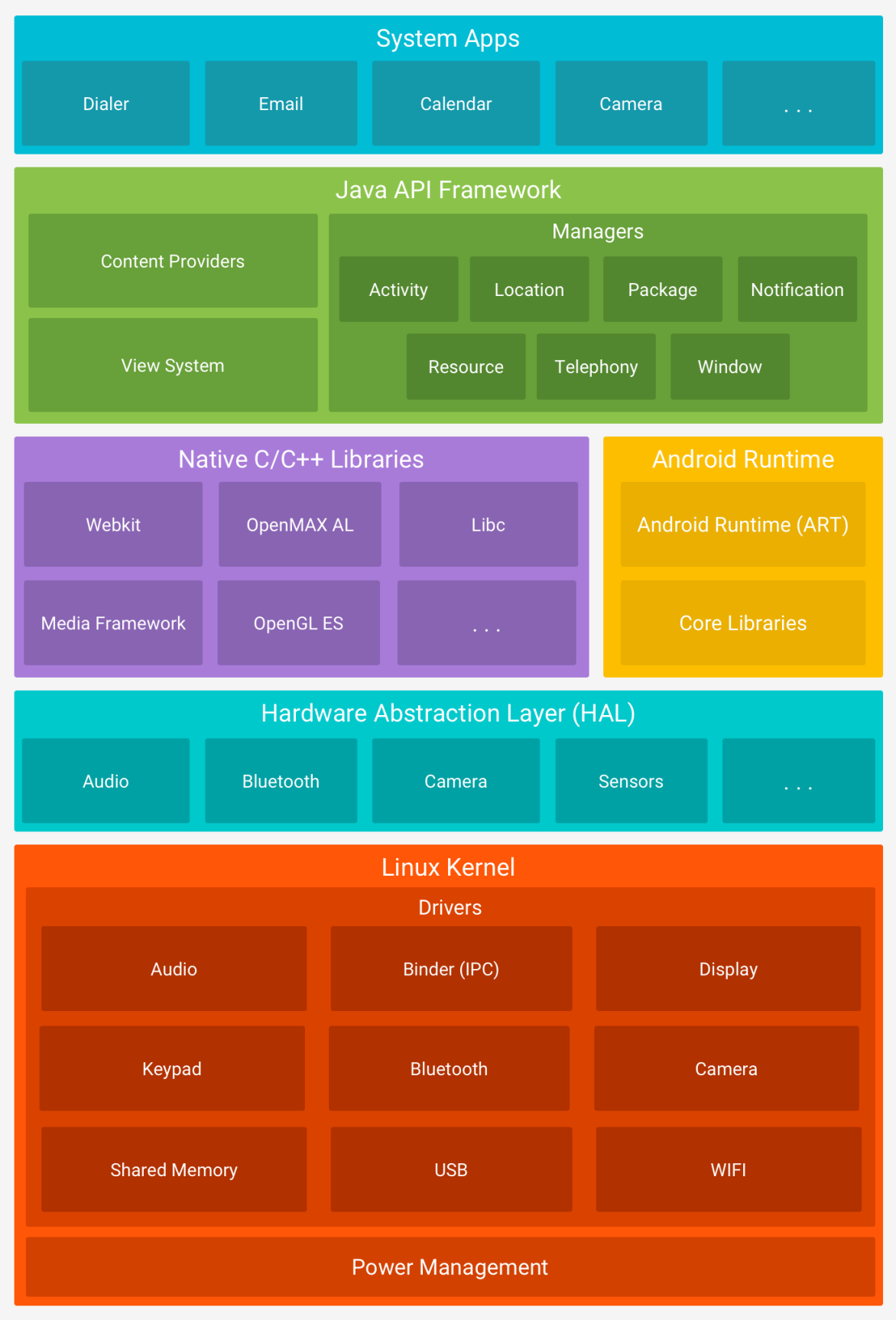
1 ANDROID

Je svetovo najrozšírenejšia mobilná platforma vyvinutá spoločnosťou Google. Je založená na modifikovanej verzii Linuxu a je open source. Primárne je využívaná pre zariadenia z dotykovou obrazovkou, ako sú smartphóny alebo tablety, ale taktiež aj pre televízie, autá alebo hodinky a iné. [1, 2, 37]

Prvý komerčný Android vyšiel v roku 2008 a v dobe písania tejto práce bola najnovšia verzia systému vo verzií 8.1.0 „Oreo“.

1.1 Architektúra Android

Architektúra Android OS je rozdelená do niekoľkých vrstiev. Nasledujúci diagram zobrazuje hlavné komponenty tejto platformy, ktoré sú ďalej rozpísané.



Obrázok 1: Architektúra systému Android [3]

1.1.1 Linux Kernel

Linuxové jadro je mimoriadne dôležitou súčasťou softvéru na takmer každom zariadení so systémom Android. Je najnižšou vrstvou architektúry a vytvára abstraktnú vrstvu medzi hardvérom a vyššími vrstvami. Jadro systému Android bolo založené na Linuxe vo verzii 2.6 dnes už 4.x. Ako je vidno na obrázku vyššie, Android využíva niekoľko vlastností, ktoré Linux ponúka, ako napríklad správu sietí, pamäte, rôzne ovládače a iné. Jedným z dôvodov, prečo bolo použité práve jadro Linuxu je jeho ľahká kompilácia na rôznych zariadeniach, a teda jeho prenositeľnosť. [4]

1.1.2 Hardware Abstraction Layer

Hardvérová abstrakčná vrstva poskytuje štandardné rozhranie, ktoré sprostredkúva schopnosti hardvéru zariadenia pre Java API framework vo vyššom leveli. Pozostáva z viacerých knižníc, z ktorých každá implementuje rozhranie pre konkrétny typ hardvérovej súčasti (kamera, bluetooth a iné). Keď API rozhranie uskutoční požiadavku na prístup k hardvéru zariadenia, systém Android načíta modul pre túto hardvérovú súčasť. [5]

1.1.3 Android Runtime

Android Runtime je riadený čas behu používaný aplikáciami a niektorými systémovými službami v systéme Android. V prípade zariadenia so systémom Android verzie 5.0 (API 21) a vyšším, každá aplikácia beží vo svojom vlastnom procese a má vlastnú inštanciu Android Runtime. Android Runtime je napísaný tak, aby spustil niekoľko virtuálnych strojov na zariadeniach s malou pamäťou pomocou súborov DEX (bytecode formát navrhnutý špeciálne pre Android, ktorý je optimalizovaný pre minimálnu pamäťovú stopu). [3, 6]

Medzi hlavné vlastnosti Android Runtime patria:

- Ahead-of-time a just-in-time kompilácia
- Optimalizovaný tzv. garbage collector
- Lepšia podpora pri ladení vrátane špeciálneho vzorkovacieho profileru, podrobných diagnostických výnimiek a hláseniach o pádoch, a taktiež možnosť nastaviť sledovacie body (watchpoints).

Android tiež obsahuje sadu runtime knižníc v jadre, ktoré poskytujú väčšinu funkcionality programovacieho jazyka Java, vrátane niektorých jazykových funkcií Java 8, ktoré používa Java API framwork.

1.1.4 Natívne knižnice C / C++

Veľa základných komponentov a služieb systému Android, ako napríklad Android Runtime a Hardware Abstraction Layer, sú postavené z natívneho kódu, ktorý vyžaduje natívne knižnice napísané v C a C++. Platforma Android poskytuje Java Framework API pre sprístupnenie niektorých z týchto natívnych knižníc aplikáciám. Napríklad možnosť získať prístup k OpenGL ES prostredníctvom frameworku Android Java OpenGL API a pridať podporu pre kreslenie a manipuláciu s 2D a 3D grafikou v aplikácii. [3, 38]

Ak vyvíjaná aplikácia vyžaduje kód C alebo C++, je možné použiť Android SDK k prístupu k niektorým s týchto natívnych knižníc priamo z kódu.

1.1.5 Java API Framework

Celý set funkcií systému Android je k dispozícii prostredníctvom rozhraní API napísaných v jazyku Java. Tieto rozhrania API tvoria stavebné bloky, ktoré sú potrebné na vytvorenie aplikácií pre systém Android, a to zjednodušením opätovného použitia jadrových, modulárnych systémových komponentov a služieb, medzi ktoré patria:

- View systém na budovanie UI obsahujúci tabuľky, zoznamy, textové bloky, tlačidlá a aj vstavaný webový prehliadač
- Správca zdrojov, ktorý poskytuje prístup k nekódovým zdrojom ako sú lokalizované reťazce (preklady), grafika a layouty.
- Správca upozornení, ktorý umožňuje všetkým aplikáciám zobrazovať ich vlastné upozornenia v stavovom riadku
- Správca aktivít. Ten riadi životný cyklus aplikácie a poskytuje spoločný zásobník spätnej navigácie
- Poskytovatelia obsahu, ktorí umožňujú aplikáciám prístup k údajom z iných aplikácií, ako napríklad Kontakty alebo na zdieľanie vlastných údajov

Vývojári majú plný prístup k rovnakým API rozhrania, ktoré používajú systémové aplikácie systému Android. [3, 37, 38]

1.1.6 Systémové aplikácie

Android je dodávaný s množstvom vstavaných aplikácií pre e-mail, SMS správy, kalendár, prehliadanie internetu, kontakty a ďalšie. Aplikácie, ktoré sú súčasťou platformy, nemajú žiadny špeciálny status vzhľadom na aplikácie, ktoré sa používateľ rozhodne inštalovať.

Takže aplikácia tretej strany sa môže stať predvoleným webovým prehliadačom, správcom SMS alebo dokonca predvolenou klávesnicou (existujú však aj výnimky, napríklad aplikácia Nastavenia). [3]

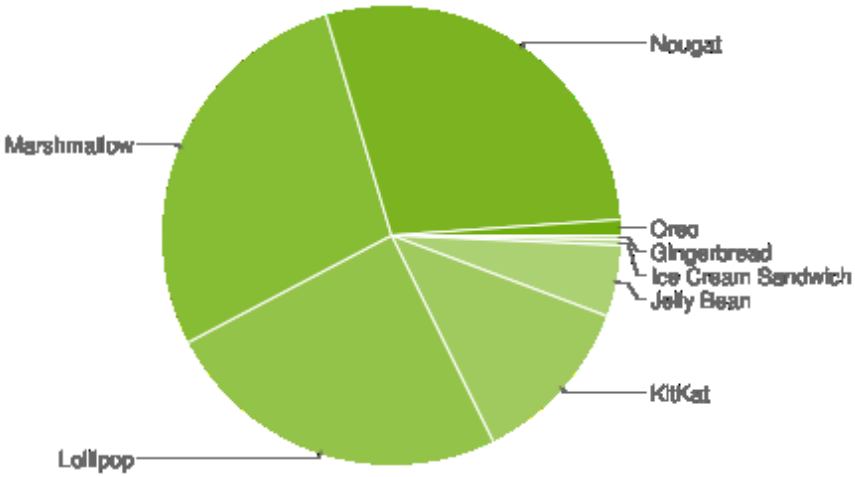
Systémové aplikácie fungujú ako aplikácie pre používateľov a poskytujú aj možnosť prístupu vývojárovi k tejto aplikácii z vlastnej aplikácie. Ak by napríklad aplikácia chcela doručiť SMS správu, nemusí vývojár tieto funkcie vytvárať sám, ale môže namiesto toho vyvolať aplikáciu SMS, ktorá je už nainštalovaná, aby doručil správu príjemcovi, ktorého užívateľ zadal.

1.2 Verzie Androidu

V nasledujúcej tabuľke je zobrazené koľko percent užívateľov využíva ktorú verziu OS Android. Táto tabuľka je taktiež vyobrazené v podobe koláčového grafu.

Tabuľka 1: Percentuálne zastúpenie verzií Android na trhu [7]

Verzia	Názov	API	Podiel na trhu
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.7%
4.2.x		17	2.6%
4.3		18	0.7%
4.4	KitKat	19	12.0%
5.0	Lollipop	21	5.4%
5.1		22	19.2%
6.0	Marshmallow	23	28.1%
7.0	Nougat	24	22.3%
7.1		25	6.2%
8.0	Oreo	26	0.8%
8.1		27	0.3%



Obrázok 2: Graf percentuálneho zastúpenia verzií Android na trhu [7]

1.3 Aktivita

Takmer všetky aktivity interagujú s užívateľom, takže trieda aktivita (Activity) sa stará o vytvorenie okna, v ktorom je možné umiestniť užívateľské rozhranie pomocou *setContentView(View)*. Aktivity sú často prezentované ako okná na celú obrazovku, ale môžu byť použité aj inými spôsobmi, ako napríklad plávajúce (floating) okná alebo vložené do inej aktivity (pomocou ActivityGroup). [8, 37, 38]

Trieda aktivita je dôležitou súčasťou celkovej životnosti aplikácie a spôsob spustenia a zostavovania aktivít je základnou súčasťou aplikačného modelu platformy.

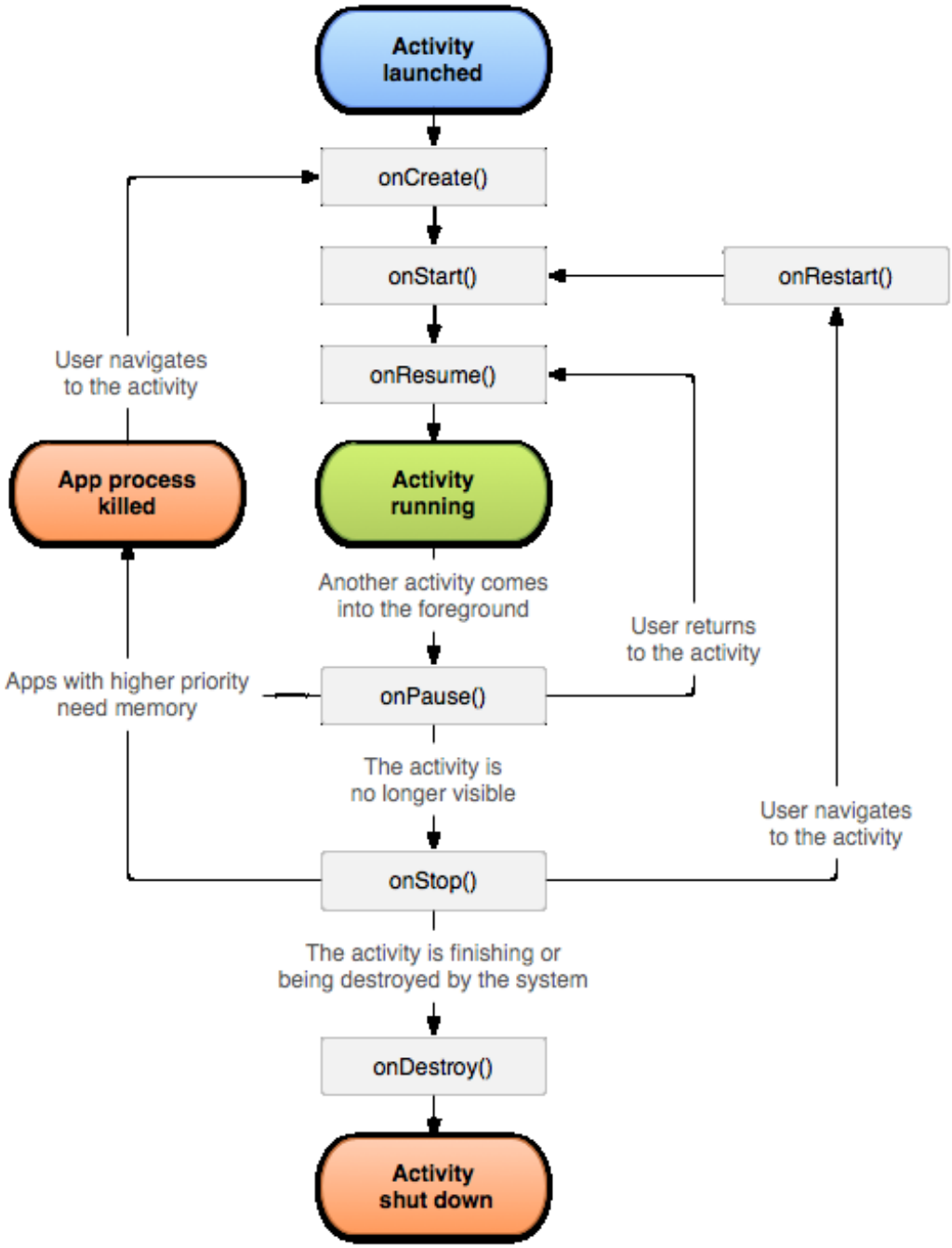
1.3.1 Fragment

Fragment predstavuje správanie alebo časť používateľského rozhrania v aktivite. Je možné kombinovať viacero fragmentov v jednej aktivite a vytvoriť tak viacúčelové používateľské rozhranie. Fragmenty je možné použiť vo viacerých aktivitách. Fragment si môžeme predstaviť ako modulárnu časť aktivity, ktorá má svoj vlastný životný cyklus, prijíma svoje vlastné vstupné udalosti a ktorú môžeme pridať alebo odstrániť počas činnosti (podobne ako "sub aktivitu", ktorú možno opätovne použiť v rôznych aktivitách). [8, 37, 38]

Fragment musí byť vždy zahrnutý do činnosti a životný cyklus fragmentu je priamo ovplyvnený životným cyklom hostiteľskej aktivity.

1.3.2 Životný cyklus aktivity

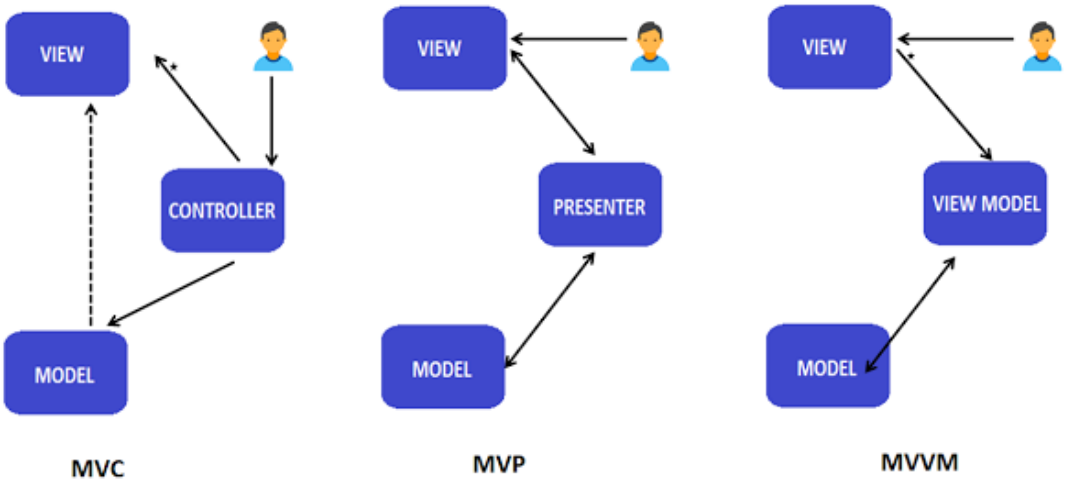
Na nasledujúcom obrázku je zobrazený životný cyklus Android Activity. Sú tu zobrazené jednotlivé stavy, do ktorých sa môže aktivita dostať.



Obrázok 3: Životný cyklus aktivity [8]

2 NÁVRHOVÝ VZOR - ARCHITEKTÚRA

Návrhový vzor (design pattern) je rokmi preverený postup, podľa ktorého postupujeme pri tvorbe programu. Použitie návrhového vzoru nám umožňuje písať kód, ktorý má logiku, ľahšie sa v ňom orientuje a je aj pre iného človeka viac čitateľný. Taktiež je menšia šanca urobiť pri písaní kódu chybu. [9, 39]



Obrázok 4: Návrhové vzory [10]

2.1 MVC

Základnou myšlienkou architektúry MVC je oddelenie logiky od výstupu. Toto delenie sa skladá s troch častí Model (model), View (pohľad) a Controller (kontrolér). Model reprezentuje dáta a business logiku, pohľad zobrazuje užívateľské rozhranie (UI) a kontrolér má na starosti aplikačnú logiku, teda tok udalostí v aplikácií. [11, 12, 39]

Obrázok 5: Model MVC

2.1.1 Model

Model sú dáta, stavy a logika programu. Môžu to byť výpočty, databázové dopyty, validácia a podobne. Je to „mozog“ aplikácie a nie je viazaný na pohľad ani kontrolér, a preto je použiteľný v mnohých kontextoch. Jeho funkcia spočíva v prijímaní parametrov a vydáva-

ní dát von. Model nevie odkiaľ dáta v parametroch prišli, ani ako budú výstupné dáta sformátované. Môže byť taktiež poverený sledovaním zmien v dátach (napr. v databáze), o ktorých následne informuje kontrolér. [11, 13, 39]

2.1.2 View

Pohľad je reprezentácia modelu a má za úlohu renderovať užívateľské rozhranie komponentmi XML, HTML a pod. Pohľad zobrazuje dáta, ktoré mu poslal kontrolér a komunikuje s ním keď užívateľ interaguje s aplikáciou. V architektúre MVC je pohľad obecne vcelku "hlúpy", pretože nemá žiadne znalosti modelu ani stavov alebo čo robiť, keď užívateľ klikne na tlačidlo, alebo zadá nejakú hodnotu. Myšlienka je taká, že čím menej pohľad vie, tým je nezávislejší od modelu, a preto je aj flexibilnejší. [10, 13, 39]

2.1.3 Controller

Kontrolér je riadiaci prvok zodpovedný za spracovanie prichádzajúcich požiadavkou. Je to v podstate "lepidlo", ktoré drží aplikáciu pohromade. Riadi, čo sa deje v aplikácií. Ak pohľad povie kontroléru, že užívateľ klikol na tlačidlo alebo zadal hodnotu, kontrolér rozhodne, čo sa má stať a ako bude reagovať model. Na základe zmenených údajov v modeli kontrolér rozhodne, či je potrebné zmeniť pohľad. V skratke kontrolér spracúva dáta prostredníctvom modelu a odovzdáva výsledky pohľadu. Je to sprostredkovateľ medzi pohľadom a modelom. V Android aplikáciách je takmer vždy reprezentovaný aktivitou (Activity) alebo fragmentom (Fragment). [10, 13, 39]

2.2 MVP

Tento návrhový vzor je podobný ako MVC. Je od neho odvodený, kde je ale kontrolér nahradený prezentérom, a teda MVP znamená Model (model), Viewu (pohľad) a Preseneter (prezentér). MVP rozbíja kontrolér tak, že prirodzene spojí pohľad a aktivitu bez väzieb na ostatné „zodpovednosti“ kontroléru. Hlavným rozdielom medzi kontrolérom a prezentérom je, že kontrolér definuje chovanie a typicky spravuje niekoľko pohľadov. Prezentér je vo väčšine prípadov priamo viazaný s jedným. Ďalej v architektúre MVC môže pohľad komunikovať priamo s modelom zatiaľ čo v MVP je to prezentér, ktorý je zodpovedný za komunikáciu medzi pohľadom a modelom. [10, 13, 14]

Obrázok 6: Model MVP

2.2.1 Model

Rovnaký ako v MVC.

2.2.2 View

Pohľad je zobrazovacia vrstva, ktorá zobrazuje informácie užívateľovi a všetky udalosti (stlačenie tlačidla, vloženie textu ...) presúva k prezentéru. Jedinou zmenou oproti MVC je, že je aktivita alebo fragment teraz súčasťou pohľadu. Je dobrou praxou implementovať aktivite pohľadu rozhranie (Interface) tak, aby mal prezentér rozhranie na kódovanie. Týmto eliminuje spojenie so špecifickým pohľadom a umožní nám to jednoduché testovanie s falošnou implementáciou pohľadu. [13, 14]

2.2.3 Presenter

Úlohou prezentéru je spracovávať udalosti pohľadu a dopytovať sa na dáta z modelu. Po spracovaní údajov modelom ich výsledky prezentér odovzdá späť pohľadu. Prezentér komunikuje s pohľadom pomocou rozhrania, ktoré je definované v triede prezentéru, na ktoré prichádzajú dáta. Aktivita, fragment alebo iná komponenta pohľadu implementuje toto rozhranie a zobrazuje dáta ako potrebuje. [10, 13, 14]

V skratke v návrhovom vzore MVP prezentér manipuluje s modelom a aktualizuje pohľad. Pohľad a prezentér sú od seba úplne oddelené a komunikujú medzi sebou iba pomocou rozhrania.

2.3 MVVM

MVVM podporuje obojsmerný data binding (dátová väzba) medzi pohľadom a View-Model-om. Toto umožňuje automatické šírenie zmien vo vnútri ViewModel-u do pohľadu. Vo všeobecnosti ViewModel používa tzv. observer pattern (vzor pozorovateľa) na informovanie modelu o zmenách vo ViewModel-i.

Obrázok 7: Model MVVM

2.3.1 Model

Rovnaký ako v MVC.

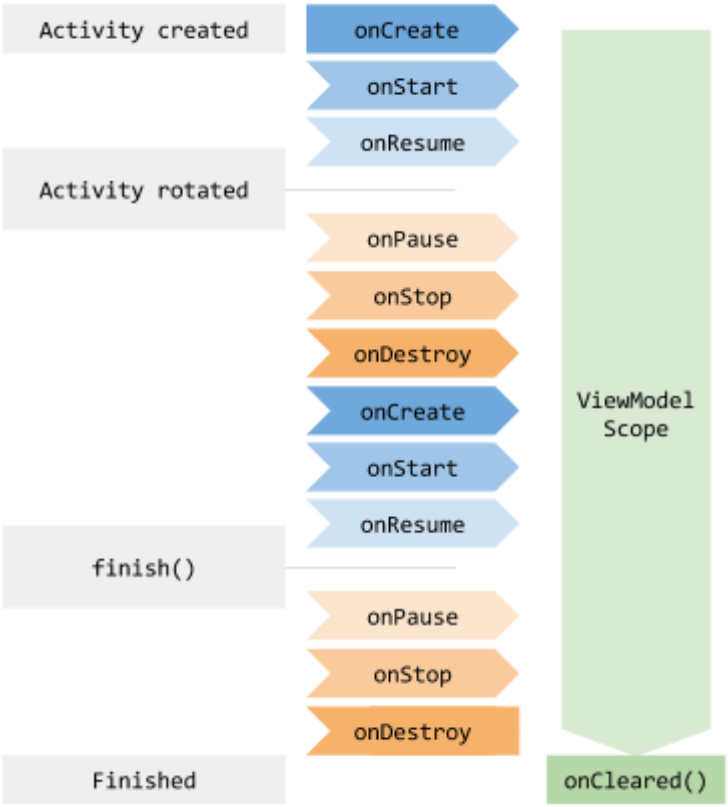
2.3.2 View

Pohľad sa flexibilne viaže na pozorovateľné premenné a akcie vyvolané ViewModel-om. [13]

2.3.3 ViewModel (VM)

VM je zodpovedný za spustenie metód, príkazov a ďalších vlastností, ktoré pomáhajú udržovať stav pohľadu, manipulovať s modelom na základe vyvolaných akcií v pohľade a spúšťať udalosti v pohľade samotnom. Pohľad má odkaz na VM, ale ten nemá žiadne informácie o pohľade. Medzi nimi existuje väzba many-to-one. To znamená, že viac pohľadov môže byť namapovaných na jeden VM. VM je kompletne nezávislý na pohľade. [10, 13]

Obojsmerný data binding medzi pohľadom a VM zaisťuje, že modely a vlastnosti vo VM sú synchronizované s pohľadom. Návrhový vzor MVVM je vhodný pre aplikácie, ktoré potrebujú podporu obojsmernej väzby údajov.



Obrázok 8: Porovnanie životného cyklu aktivity a ViewModel-u [15]

Na tomto obrázku je zobrazené porovnanie životného cyklu VM a Android Activity pri rotácií obrazovky. Z obrázku je teda zrejmé, že aktivita sa pri rotácií zničí a znovu spustí na rozdiel od VM, ktorý túto rotáciu prežije. Takto sú uchované všetky dáta napr. hodnoty textových polí a iné.

2.4 Data Binding

Poskytuje flexibilný mechanizmus pre synchronizáciu dát a užívateľského rozhrania (UI). Data binding je proces, pri ktorom sa vytvára spojenie medzi UI aplikácie a jej logikou. Ak je toto zviazanie správne nastavené a dáta poskytujú správne notifikácie, tak potom, ak sú dáta zmenené, prvky, ktoré sú zviazané s týmito dátami, sa automaticky aktualizujú. Takýmto spôsobom môžeme zautomatizovať zmeny v dátach a ich odzrkadlenie v UI. Výhodou je, že takto môžeme pracovať s väčšinou zdrojov dát. [16, 17]

Data binding môže taktiež znamenať, že ak sa zmení hodnota "vonkajšej" reprezentácie dát v prvku (v UI), automaticky sa zmení aj základná hodnota. Napríklad, ak užívateľ upraví hodnotu napr. v TextBox-e, základná hodnota dát (vo VM) sa automaticky aktualizuje tak, aby odrážala túto zmenu.

2.4.1 LiveData

Je to pozorovateľná Android trieda na udržiavanie dát. Umožňuje komponentám v aplikácií sledovať objekty typu LiveData pre zmenu údajov bez toho, aby medzi nimi vytvorili explicitné a pevné závislosti. Toto kompletne oddeľuje producenta od konzumenta dát typu LiveData. [18, 19]

Veľkou výhodou je aj to, že LiveData rešpektujú životný cyklus komponent aplikácie (aktivity, fragmenty, servisy) a zvládajú správu životného cyklu objektov, čo zaistí, že objekty LiveData nespôsobia tzv. memory leak.

LiveData rešpektujú životný cyklus Android aplikácie, to znamená, že nebude vyvolávať svoje „pozorovateľské“ volania (aktualizovať stavy) kým jeho hostiteľ (aktivita alebo fragment) nie je v aktívnom stave (prešiel cez stav *onStart()* ale nie cez napr. *onStop()*). LiveData taktiež automaticky zmažú „pozorovateľa“ keď ich hostiteľ prejde do stavu *onDestroy().*

3 REAL-TIME VS RELAČNÁ DATABÁZA

V tejto kapitole sú popísané rozdiely medzi relačnou databázou a databázou reálneho času, ich výhody a nevýhody. Taktiež tu je opísaný princíp fungovania databázy reálneho času a niektorý predstavitelia tohto typu databázy ako Firebase, Firestore a Realm.

3.1 Relačná databáza

Relačná databáza je databáza založená na relačnom modeli, teda na tabuľkách, čo sú dvojrozmerné štruktúry tvorené záhlavím a telom. V takýchto tabuľkách je potom riadok chápaný ako jeden záznam a stĺpce sú tzv. atribúty. Tie majú svoj dátový typ a doménu (množina prípustných hodnôt). [22, 40]

Pojem "relačná databáza" súvisí s teóriou množín. Každá konkrétna tabuľka totiž realizuje podmnožinu karteziánskeho súčinu množín prípustných hodnôt všetkých stĺpcov - relácii.

3.2 Real-time databáza

Databáza reálneho času je databáza, ktorá využíva spracovanie dát v reálnom čase na spracovanie požiadavkou neustále sa meniacich dát. To je hlavný rozdiel od klasických databáz, ktoré pracujú prevažne s časovo konštantnými dátami. Spracovanie v reálnom čase v tomto prípade znamená, že požiadavka je spracovaná dostatočne rýchlo, aby sa výsledok mohol okamžite vrátiť. Takýto typ databáz sa používa napr.: v rezervačných systémoch, multimédiách, účtovníctve, pri riadení procesov alebo v chatovacích aplikáciách a aplikáciách, v ktorých je nutné zdieľať rôzne údaje ako stav, poloha a iné. [20, 21, 40, 41]

Databázy reálneho času sú najčastejšie charakterizované:

- Prístupom k dátam v časovom obmedzení
- Prístupom k dátam s dočasnou platnosťou

Dané operácie zahrňujú zber dát, ich spracovanie v závislosti na kontexte predchádzajúcich dát a poskytnutie výsledku. Ďalej taktiež spracovanie dočasných dát, ktoré stratia svoju platnosť po uplynutí určitého času. Na rozdiel od klasických databáz majú dáta tohto typu databáze vlastnosť, ktorá vyplýva s toho, že použité dáta v prípade časovo kritických aplikácií musia čo najpresnejšie odrážať reálny stav. Keďže dáta sú zbierané v diskrétnych intervaloch, je daná reprezentácia len aproximáciou reality. S pribúdajúcim časom je táto

aproximácia menej presná až dosiahne bod, kedy neodráža skutočný stav reality. S toho dôvodu sa rozlišujú nasledujúce pojmy:

- Časová značka – kedy bola hodnota získaná
- Absolútny interval platnosti – interval v ktorom je aktuálna hodnota považovaná za platnú
- Relatívny interval platnosti

Časové nároky na odozvu systému spolu s dočasnou platnosťou dát určujú časové požiadavky na transakcie. Preto korektnosť databázových transakcií v reálnom čase nezáleží len na rýchlosti logických výpočtov, ale tiež na čase, kedy sú výsledky transakcie vybavené. [21]

3.3 Porovnanie Real-time vs Relačná DB

Relačné databázy ako MySQL, PostgreSQL alebo SQLite reprezentujú uložené dáta v tabuľkách a riadkoch. Sú založené na tzv. relačnej algebre, zatiaľ čo nerelačné databázy, medzi ktoré patria aj databázy reálneho času, reprezentujú dáta v kolekciách napr. JSON dokumentov. [23, 40, 41]

Relačné databázy využívajú štruktúrovaný dopytovací jazyk (SQL), čo je dobrá voľba pre aplikácie, ktoré zahŕňajú správu viacerých transakcií. Štruktúra relačnej databázy umožňuje prepojiť informácie z rôznych tabuliek pomocou tzv.: cudzích kľúčov (alebo indexov), ktoré sa používajú na jednoznačnú identifikáciu akýchkoľvek dát v rámci tejto tabuľky. Ostatné tabuľky sa môžu vzťahovať na tento cudzí kľúč, aby sa vytvorilo prepojenie medzi ich údajmi a údajmi, na ktoré odkazuje cudzí kľúč. To je užitočné pre aplikácie, ktoré sú náročné na analýzu dát.

Ak chceme, aby aplikácia spracovala veľa komplikovaných dopytov, databázových transakcií a rutinnej analýzy údajov, pravdepodobne sa budeme chcieť držať relačnej databázy. A v prípade, že sa aplikácia bude zameriavať na vykonávanie veľkého množstva databázových transakcií, je dôležité, aby boli tieto transakcie spoľahlivo spracované. Práve v tomto prípade naozaj záleží na ACID (súbor vlastností zaručujúce spoľahlivosť DB transakcií), a taktiež sa zohľadňuje referenčná integrita.

Referenčná integrita je koncept, v ktorom viaceré databázové tabuľky zdieľajú vzťah založený na údajoch uložených v tabuľkách a tento vzťah musí zostať konzistentný. Toto sa zvyčajne vynucuje kaskádovými akciami pridávania, odstraňovania a aktualizácie.

Relačné databázy sú skvelé, ale prichádzajú s rôznymi kompromismi. Jedným z nich je objektovo-relačný nesúlad, pretože relačné databázy neboli pôvodne vytvorené, aby brali ohľad na OOP jazyky. Najlepším spôsobom, ako sa tomuto problému vyhnúť, je vytvorenie schémy databázy s referenčnou integritou v jeho jadre. Takže pri použití relačnej databázy s OOP je potrebné premýšľať o tom, ako nastaviť primárne a cudzie kľúče, používať obmedzenia (vrátane kaskádových mazaní a aktualizácií) a ako napísať svoje migrácie.

To je veľmi náročné a v prípade veľkého množstva dát to môže byť príliš zdĺhavé a pravdepodobnosť chyby sa zvyšuje. V tejto situácií je preto potrebné zvážiť použitie nerelačnej databázy. Nerelačná databáza v podstate ukladá údaje bez explicitných a štruktúrovaných mechanizmov na vzájomné prepojenie údajov rôznych tabuliek. Ak sa dátový model databázy ukáže byť veľmi zložitý, alebo ak zistíme, že je potrebné denormalizovať databázovú schému, môžu byť nerelačné databázy najlepšou voľbou. Medzi ďalšie dôvody na výber nerelačnej databázy patria:

- Potreba ukladať serializované pole do objektov JSON
- Ukladanie záznamov, ktoré majú rôzne polia alebo atribúty v rovnakej kolekcií
- Zistenie, že je treba denormalizovať databázovú schému alebo kódovanie spojené s problémom s výkonom a horizontálnou škálovateľnosťou
- Problémy s jednoduchou predbežnou definíciou schémy z dôvodu povahy dátového modelu

NoSQL databázy umožňujú vývojárom definovať tok aplikácie celý na strane kódu. Jednou z najväčších výhod pri práci s nerelačnou databázou je to, že táto databáza nie je ohrozená útokmi typu SQL injection, pretože nerelačné databázy nepoužívajú SQL a sú z väčšej časti bez schémy. Ďalšou veľkou výhodou je, že ju teoreticky v niektorých prípadoch môžeme deliť navždy. Delenie distribuuje údaje naprieč oddielmi na prekonanie hardvérových obmedzení. Keďže NoSQL databázy neobsahujú spojenia ako sú v relačných DB a je nutné vykonať niekoľko dopytov a spojiť dáta manuálne v rámci kódu.

Tabuľka 2: Porovnanie SQL a NoSQL databáz [24]

Vlastnosť	**SQL (relačná)**	**NoSQL (real-time)**
Štýl ukladania	tabuľky	dokumenty, kľúč-hodnota, grafová štruktúra

Organizácia dát	vopred definovaná schéma	dynamická schéma
Škálovateľnosť (zvyšovanie výkonu)	vertikálne (väčšia RAM, silnejší procesor...)	horizontálne (vyšší počet serverov, inštancií)
Dopytovací jazyk	štandardizovaný SQL	vlastný dopytovací jazyk
Previazanosť dát	cudzie kľúče	vnorené dokumenty
Bezpečnosť	transakcie, konzistencia, izolovanosť	žiadna

3.4 Real-time DB – fungovanie

Princíp fungovania databázy reálneho času je v podstate jednoduchý. Najskôr sa vytvorí spojenie pomocou Websocketu medzi serverom a klientom. Následne sa na strane klienta vytvorí listener, ktorý počúva zmeny vykonané v databáze.

3.4.1 WebSocket

WebSocket je počítačový komunikačný protokol predstavujúci štandard pre plne duplexný (obojsmerný) komunikačný kanál cez jediné TCP pripojenie v reálnom čase medzi klientom a serverom. Nie je teda nutné používať techniku tzv. "Heartbeat", kedy sa klient v pravidelných intervaloch pýtal servera "Máš pre mňa niečo?" (A server väčšinu času odpovedal "Nie ..."). S WebSocket-mi je oveľa ľahšie vytvárať real-timové aplikácie, ako sú online chaty či kooperatívne služby. [25, 26]

3.4.2 Listener

Údaje uložené v databáze sa získavajú, keď pripojíme asynchrónny listener k referencii databázy. Listener sa spustí raz pre inicializáciu údajov na začiatku a znova kedykoľvek, keď sú dáta zmenené. Listener môže mať niekoľko rôznych typov udalostí. [27]

3.5 Firebase, Realm, Firestore

V tejto časti sú opísaný niektorý predstavitelia databáz reálneho času. Konkrétne Firebase, Firestore a Realm.

3.5.1 Realm

Realm je open-source objektový systém na správu databáz pre mobilné zariadenia (Android, iOS). Je taktiež dostupný aj pre iné platformy ako Xamarin, React Native a iné, dokonca aj na desktopové zariadenia (Windows). [28]

3.5.1.1 Komponenty Realm

V tejto podkapitole sú opísané základné komponenty Realm DB.

Databáza

Databáza Realm je objektovo orientovaná, multiplatformová databáza, ktorá pretrváva v lokálnom zariadení. Je dostupná pre hlavné mobilné jazyky ako Swift, Objective-C, Java, C# a JavaScript. Je výkonná a schopná zaobchádzať s veľkými dátami a požiadavkami v zlomkoch sekundy. Databáza je založená na zdieľaní objektov a synchronizácií dát v reálnom čase pomocou Realm Object Serveru. [28]

Object Server

Realm Object Server zrkadlí databázu na zariadení. Funguje ako middleware v architektúre mobilnej aplikácií a spravuje synchronizáciu dát, spracovanie udalostí a integráciu so staršími zariadeniami. Tento server efektívne a súčasne synchronizuje dáta medzi zariadeniami a automaticky rieši konflikty v reálnom čase. Taktiež poskytuje miesto na správu všetkých komunikácií, vrátane starého rozhrania API transakcií. [28]

Realm Object Server je možné nasadiť na cloud, Liunxové alebo macOS zariadenia.

Node.js SDK

Hlavným komponentom Realm Object Serveru je Realm Node.js SDK. Umožňuje vývojárom písať logiku, ktorá riadi tok dát medzi aplikáciou a serverom. A čo je najdôležitejšie, umožňuje integráciu do backendu cez rôzne API konektory, čím vytvára spojenie so staršími systémami. [28]

3.5.1.2 Hlavné funkcie

Tu sú opísané hlavné funkcie databázy Realm.

Synchronizácia dát

Základná sila Realm DB spočíva v reaktívnej architektúre, ktorá umožňuje synchronizáciu údajov v reálnom čase medzi dátami uloženými na Realm Object Serveri a údajmi uloženými v ľubovoľnej aplikácií bežiacej na mnohých mobilných platformách. Po zistení aktualizácie Realm iba synchronizuje zmeny. [28]

Škálovateľnosť

V závislosti od prostredia aplikácií a hardvéru môže jeden server objektov Realm spracovať viac ako desaťtisíc používateľov súbežne. Pre väčší rozsah Enterprise Edition umožňuje paralelné nasadzovanie viacerých inštancií Realm Object Server, takže aplikácia môže podporovať viac ako jeden milión používateľov a každá z nich poskytne rovnaké real-time skúsenosti. Server obsahuje zabudovaný vyvažovač záťaže, ktorý automaticky spracúva distribučné spojenia. [28]

Multiplatformový dátový model

Prístup Realm zabezpečuje kompatibilitu viacerých platforiem v celej aplikácií. Realm objektový server a všetky používateľské zariadenia ukladajú rovnaké dátové objekty v natívnom jazyku aplikácie. Všetky objekty počúvajú rovnaké aktualizácie a rovnako synchronizujú rovnaké dáta. Tento zjednotený model znamená, že vývojový tím navrhuje a vytvára dátový model aplikácie iba raz a potom ju jednoducho prenesie do jednotlivých variant aplikácie (iOS, Android ...). [28]

Správa udalostí

Realm server poskytuje framwork na správu udalostí, ktorý môže pri zmene dát spustiť logiku zo strany serveru. Po zmene dát na strane klienta sa dáta synchronizujú zo serverom, čo potom spustí kód na spracovanie tejto zmeny. Správa udalostí umožňuje serveru spúšťať dátový tok medzi aplikáciou a staršími API a staršími systémami. Je možné taktiež napísať vlastné funkcie pre každý spúšťač udalostí (event). [28]

Data Integration API

Pomocou Data Integration API môžu vývojári ľahko prepojiť Realm Object Server s existujúcimi dátovými zdrojmi alebo službami bez potreby dodatočného kódu na strane klienta.

To umožní zobraziť hlavné údaje v rámci skúseností s aplikáciou a spracovať dáta generované používateľmi späť do starších systémov. Rozhranie API uľahčuje vytváranie vlastných konektorov alebo použitie jedného z vopred pripravených konektorov, ako je napríklad PostgreSQL. [28]

Zálohy

Realm poskytuje odolnosť voči zlyhaniu pre kritické dáta. Umožňuje vytvárať automatické zálohy a uložiť ich na záložný server. V prípade výpadku prúdu, poškodenia hardvéru, prírodnej katastrofy alebo podobnej udalosti, môže byť aplikácia naďalej bez problémov prevádzkovaná zo sekundárneho servera. [28]

Zabezpečenie

Funkcie zabezpečenia na úrovni Enterprise, ako je šifrovanie AES-256, zabezpečujú, že citlivé údaje sú vždy bezpečné. Platforma Realm využíva štandardné šifrovanie dát na zariadeniach a serveri a šifrovanie počas synchronizácií dát. [28]

3.5.2 Firebase

Firebase je služba poskytovaná spoločnosťou Google, ktorá poskytuje funkcionalitu ako real-time databáza, autentifikácia, analýzy, zasielanie správ a hlásení o zlyhaní a iné. Jednotlivé produkty Firebase pracujú individuálne, ale zdieľajú medzi sebou údaje, takže dokážu pracovať aj spolu. [29]

Real-time databáza

Slúži na ukladanie a synchronizáciu dát medzi užívateľmi a zariadeniami v reálnom čase využívajúc cloudovú NoSQL databázu. Umožňuje dátovú synchronizáciu naprieč zariadeniami v priebehu milisekúnd a dáta zostávajú k dispozícií aj keď aplikácia prejde do režimu offline. Poskytuje tak skvelé „user experience“ (užívateľský zážitok) bez ohľadu na pripojenie k sieti. [29]

Autentifikácia

Spravuje používateľov jednoduchým a bezpečným spôsobom. Firebase Auth ponúka niekoľko metód na overenie totožnosti, vrátane e-mailu a hesla alebo pomocou poskytovateľov tretích strán ako je Google, Facebook, Twitter alebo GitHub a použiť tak existujúci účet. Umožňuje taktiež vytvorenie vlastného rozhranie alebo využitie plne upraviteľného open source UI. [29]

Cloud Storage

Ukladá a zdieľa obsah generovaný užívateľmi, ako sú obrázky, zvuk a video. Firebase SDK pre Cloud Storage pridáva bezpečnosť spoločnosti Google na nahrávanie a preberanie súborov pre aplikácie Firebase bez ohľadu na kvalitu siete. [29]

Cloud funkcie

Rozširuje funkcionalitu aplikácie o vlastný backend kód bez nutnosti spravovať a upravovať vlastné servery. Tieto funkcie môžu byť spustené udalosťami, ktoré vysielajú produkty Firebase, Google Cloud služby alebo webhook tretích strán. [30, 31]

3.5.3 Cloud Firestore

Cloud Firestore je ako už názov napovedá cloudová NoSQL databáza, na ktorú je možné získať prístup z iOS, Android a webových aplikácií priamo pomocou natívnych SDK. Cloud Firestore je k dispozícii aj v natívnych Node.js, Java, Python a Go SDK, okrem REST a RPC API. [32]

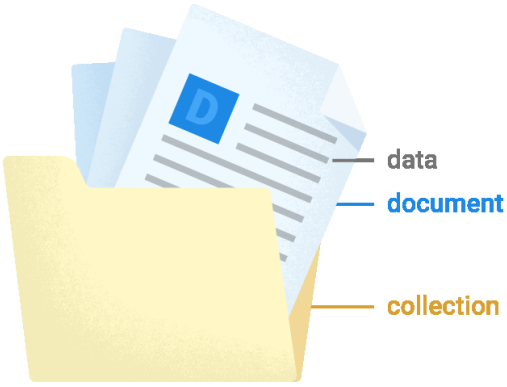
Je to flexibilná a škálovateľná databáza pre vývoj mobilných, webových a serverových aplikácií od Firebase a Google Cloud Platform. Rovnako ako databáza Realtime Firebase udržuje dáta v synchronizácii na klientskej aplikácií prostredníctvom realtime listenerov a ponúka offline podporu pre mobilné zariadenia a web, takže môžete vytvárať aplikácie, ktoré fungujú bez ohľadu na odozvu siete alebo pripojenie na internet. Cloud Firestore tiež ponúka bezproblémovú integráciu s inými produktmi Firebase a Google Cloud Platform, vrátane Cloud Functions.

3.5.3.1 Vlastnosti

Cloud Firestore ponúka automatickú multi-regiónovú duplikáciu dát, kladie dôraz na konzistenciu dát, podporuje atomické dávkové operácie a ponúka aj iné vlastnosti ako sú:

Flexibilita

Dátový model Cloud Firestore podporuje flexibilné, hierarchické dátové štruktúry. Ukladá údaje do dokumentov organizovaných do zbierok. Dokumenty môžu obsahovať komplexne vnorené objekty okrem subkolekcie. [32]



Obrázok 9: Znázornenie štruktúry Cloud Firestore [32]

Expresívne dopytovanie

V Cloud Firestore je možné použiť dopyty na načítanie jednotlivých dokumentov alebo na načítanie všetkých dokumentov v zbierke, ktorá zodpovedá parametrom dopytu. Dopyty môžu obsahovať viacnásobné, zreťazené filtre a môžu kombinovať filtrovanie a triedenie. Sú tiež štandardne indexované, takže výkonnosť dopytu je úmerná veľkosti sady výsledkov, nie na množine údajov v databáze. [32]

Aktualizácia v reálnom čase

Podobne ako real-time databáza, Cloud Firestore využíva synchronizáciu dát na ich aktualizáciu na akomkoľvek pripojenom zariadení. Je však tiež navrhnutý tak, aby efektívne obsluhoval aj jednoduché jednorazové dopyty. [32]

Podpora Offline režimu

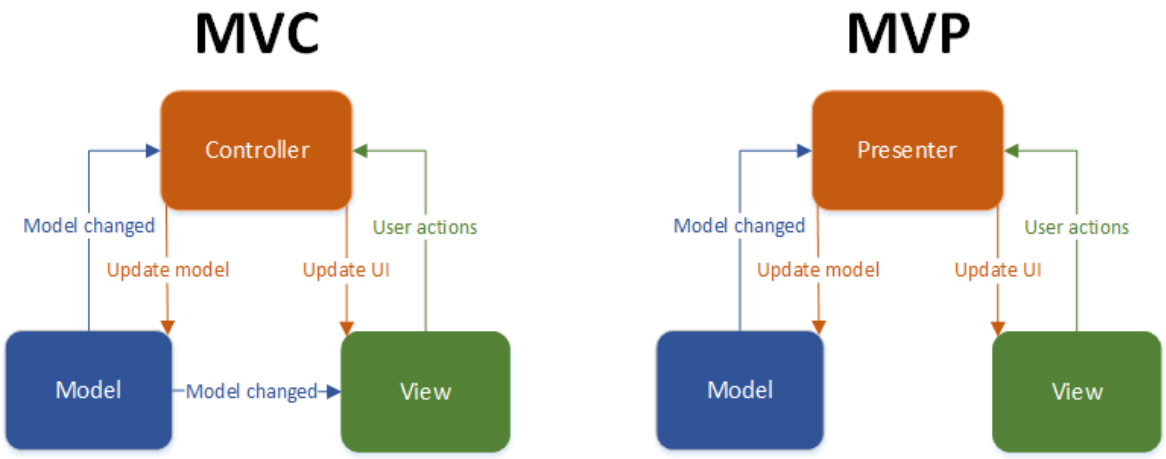
Služba Cloud Firestore ukladá údaje, ktoré aplikácia aktívne používa, takže aplikácia môže zapisovať, čítať, počúvať a vyhľadávať údaje, aj keď je zariadenie offline. Keď sa zariadenie znovu pripojí k sieti, Cloud Firestore synchronizuje akékoľvek zmeny a uloží ich do Cloud Firestore. [32]

II. PRAKTICKÁ ČASŤ

4 POROVNANIE NÁVRHOVÝCH VZOROV

V tejto kapitole sú porovnané návrhové vzory MVC, MVP a MVVM. Na konci kapitoly je zhrnuté porovnanie týchto návrhových vzorov.

4.1 MVC vs MVP



Obrázok 10: Porovnanie MVC a MVP [33]

V MVC môže byť pohľad upozornený na akúkoľvek zmenu v modeli samotným modelom. V MVP pohľad nevie nič o modeli a je úlohou prezentéra získavať aktuálne dáta z modelu a zistiť, či má byť pohľad aktualizovaný. [33, 34, 39]

Pohľady v MVC majú tendenciu mať v sebe viac logiky, pretože spracovávajú notifikácie s modelu. V MVP je táto logika zahrnutá v prezentéry, čo robí pohľad „hlúpym“. Úlohou pohľadu je vykresľovať dáta, ktoré mu dal prezentér a zachytávať užívateľské vstupy.

Tabuľka 3: Porovnanie MVC a MVP [34]

MVC	MVP
Jeden zo starších modelov používaných na delenie kódu.	Pokročilá forma MVC.
Pohľad spracováva gestá používateľov a dáva príkazy modelu.	Pohľad spracováva gestá používateľov a podľa potreby volá prezentér.

Pohľad má nejakú inteligenciu. Môže priamo dopytovať model.	Pohľad je úplne pasívny. Všetky interakcie s modelom musia prechádzať cez prezentér.
Obmedzená podpora jednotkových testov.	Vysoká podpora jednotkových testov.

4.2 MVP vs MVVM

V oboch prípadoch je Pohľad „ľahký“. Neobsahuje žiadny kód spojený s logikou aplikácie iba vizualizáciu dát. Oba návrhové vzory sú dobré pre jednotkové testy, keďže majú oddelenú business logiku od UI a oba rozdeľujú kód do rôznych vrstiev. [35]

Rozsiahlosť kódu

MVP môže produkovať viac tried a kódu. V MVVM existuje viac tried, ale menej kódu v každej triede. Veľkosť kódu v MVP je vcelku nadmerná oproti MVVM.

Fungovanie

MVP obsahuje veľké množstvo rozhraní (Interface) na interakciu medzi jednotlivými vrstvami. Každé rozhranie pokrýva interakciu do najmenších detailov. To môže mať za následok vytvorenie množstva metód.

VM vytvára toky udalostí, na ktoré sa pohľad môže viazať. Vytvára sa tak „voľné“ spojenie medzi pohľadom a VM. Ďalším veľkým rozdielom je, že prezentér musí pri vyvolaní (spustení) metódy skontrolovať, či je pohľad „živý“. Pri použití MVVM to nie je nutné, pretože ten používa LiveData, ktoré riešia tento problém.

Udržovateľnosť

MVP je vcelku ľahký na naučenie, úpravu a pridávanie funkcií. Pridanie nových funkcií do MVVM môže vyžadovať určité skúsenosti.

Logika

V aplikácii MVP je pohľad v skutočnosti aplikáciou, zatiaľ čo prezentér spracováva tok dát aplikácie. V MVVM je aplikácia v triedach kódu (VM), zatiaľ čo pohľad je rozhranie umožňujúce používateľom komunikovať s aplikáciou.

Zadávanie údajov

Vstup dát v MVP zažína v pohľade a nie v prezentéri. Rovnako v MVVM vstup začína v pohľade a nie vo VM.

Mapovanie a odkazy

Model MVP používa vzťah one-to-one medzi pohľadom a prezentérom zatiaľ čo MVVM používa medzi pohľadom a VM vzťah one-to-many. Ani jeden model nepoužíva referenciu medzi pohľadom a aplikačnou logikou (prezentérom alebo VM).

4.3 Zhrnutie

Je zrejmé, že návrhové vzory sa vyvíjajú. Z vyššie uvedených má asi najväčší potenciál MVVM, avšak aj model MVP je dostatočne flexibilný a využíva rôzne knižnice. Oba MVP a MVVM sú lepšie ako MVC pri delení aplikácie na moduly a komponenty s jedným účelom, ale taktiež pridávajú zložitosť. Pre veľmi jednoduchú aplikáciu s malým počtom obrazoviek (jedna alebo dve) môže MVC fungovať dobre. MVVM s databindingom je atraktívny model, pretože vychádza z reaktívnejšieho modelu a produkuje menej kódu. Tiež nie je striktne nutné držať sa len jedného návrhového vzoru. Vo veľkom počte prípadov nie je možné vytvoriť aplikáciu len pomocou jedného návrhového vzoru, a teda môžeme využiť kombináciu niekoľkých a ťažiť tak z ich výhod. Najdôležitejšie je v tomto prípade rozdeliť aplikáciu na pohľad, model a aplikačnú logiku. [13, 35, 39]

Nikde nie je napísané, ktorý návrhový vzor je najlepší. Záleží na konkrétnej aplikácií, aby vyťažila čo najviac z jednotlivých návrhových vzorov.

V našom prípade budeme programovať aplikáciu na zaznamenávanie príjmov a výdajov, kde použijeme návrhový vzor MVVM. Hlavným dôvodom výberu tohto návrhového vzoru bola možnosť využiť databinding a LiveData, ktoré v sebe skrývajú veľké výhody (ľahké na implementáciu, odpadá nutnosť kontrolovať či je pohľad „naživе“). Ďalej taktiež odpadá nutnosť písať zložité rozhrania (Interface) a možnosť viazať pohľad na udalosti, ktoré vyvolá ViewModel.

5 POROVNANIE DATABÁZ

Táto kapitola sa venuje porovnávaniu databáz Firebase, Realm a Firestore a to z pohľadu flexibility, aktualizácií dát, dopytovaniu, offline podpore a škáľovatelnosti.

5.1 Flexibilita

Pôvodné riešenie Firebase, ktoré uchováva všetky dáta ako jediný veľký strom JSON, neznie ako flexibilné riešenie. Je to skvelé pre jednoduché dáta, ale nie pre komplexné a hierarchické štruktúry. Preto Cloud Firestore používa štruktúru dokumentov, ktorá je podobná objektom. Ukladá údaje v dokumentoch, ktoré obsahujú mapovanie polí na hodnoty. Sú podobné ako JSON, ale organizujú sa v zbierkach a podsúboroch. Dátový model služby Cloud Firestore podporuje akúkoľvek štruktúru údajov, ktorá najlepšie funguje pre konkrétnu aplikáciu. Dokumenty podporujú mnoho rôznych typov údajov od jednoduchých reťazcov (string) a čísel po zložité vnorené objekty. [36]

Cloud Firestore stojí niekde medzi Firebase a Realm. Prístup Realm bol založený na plnom relačnom DBMS. Oblasti sú definované schémou, podobne ako tabuľky v bežnej databáze, ale sú „silnejšie“ - oblasť môže obsahovať viacero druhov objektov, každý s vlastnou vynútenou schémou. Napriek tomu na rozdiel od SQL databázy je Realm „ľahká“. Aplikácia môže využívať početné oblasti, lokálne aj vzdialené, s rôznymi oprávneniami pre rôznych používateľov.

5.2 Real-time aktualizácia dát

Všetky databázy využívajú synchronizáciu dát na aktualizáciu informácií na každom pripojenom zariadení. Firestore, na rozdiel od Firebase, je tiež navrhnutý tak, aby efektívne obsluhoval aj jednoduché dopyty. Na minimalizovanie využitia siete a načítanie iba zmien je možné vo všetkých prípadoch použiť listenery v reálnom čase. [36]

5.3 Dopytovanie

V aplikácii Cloud Firestore funguje vyhľadávanie tak, ako môžeme očakávať. Jednotlivé dokumenty je možné získať bez potreby vytiahnuť celú zbierku. Dopyty môžu obsahovať viaceré zreťazené filtre a kombinovať filtrovanie a triedenie. Sú tiež štandardne indexované, takže výkonnosť dopytu je úmerná veľkosti sady výsledkov, nie na množine údajov v databáze. [36]

Je to zlepšenie v porovnaní s Firebase, kde reťazové filtre a ich kombinovanie s triedením nebolo možné. Štruktúra údajov reprezentovaných ako jednotný JSON presadzovala dopyty tak, aby boli tieto dopyty štandardne hlboko, takže ako výsledok sa vráti celý subtree.

Realm Platform funguje podobne ako Cloud Firestore. Používateľ môže vytvoriť taký komplexný dopyt ako chce a načítať iba zodpovedajúce objekty.

5.4 Offline podpora

Realm presadzuje prístup offline-first. Synchronizačný kód je krátky, všetky aktualizácie sú automatické. Miestne úložisko je zdrojom pravdy, takže akékoľvek prerušenia siete je pre konečného používateľa väčšinou nepoznateľné. Všetky potenciálne konflikty sa automaticky vyriešia po opätovnom získaní spojenia, čo značne zjednodušuje celý proces.

Podobným spôsobom v zásade fungujú obe riešenie Firebase. Rozdiel je, že zdrojom pravdy v tomto prípade je databáza cloud a nie klientské zariadenie. Po obnovení konektivity po prerušení obdrží klientské zariadenie všetky zmeny, ktoré zmeškalo a synchronizuje ho s aktuálnym stavom servera. [36]

5.5 Škálovateľnosť

Cloud Firestore a Realm sú pripravené na rozšírenie vyvinutého produktu pre viac užívateľov alebo dát. Cloud Firestore propaguje automatickú multi-regiónovú replikáciu, silnú konzistenciu a operácie s atomickými dávkami. V prípade Realm môže byť atomické riešenie konfliktov zložité s veľkým počtom klientov pracujúcich na tých istých dátach. Rozšírenie Realm databázy spočíva v pridaní ďalšej inštancie Object Serveru. Čo sa týka Firebase je rozšírenie zložitejšie a vyžaduje delenie. [36]

5.6 Zhrnutie

Tabuľka 4: Porovnanie Cloud Firestore, Firebase a Realm DB [36]

Vlastnosť	Cloud Firestore	Firebase	Realm
Hosting	Cloud-base (BaaS)	Cloud-base (BaaS)	On-premises alebo verejný cloud
Dátový model	Dokumenty ako kolekcie	JSON	Kontajnery s natívnymi objektmi
Schéma	Bez schémy	Bez schémy	Áno

Podpora offline	iOS, Android, web	iOS, Android	iOS, Android, web
Dopytovanie	Indexované dopyty so zloženým triedením a filtrovaním	Hlboké dopytovani s obmedzenou funkčnosťou triedenia a filtrovania	Objektovo orientované dopytovanie pomocou rozhrania
Zapisovanie a transakcie	Atomické operácie zápisu a transakcie	Základný zápis a transakcie	Synchrónna a blokujúca operácia zápisu transakcií na strane klienta, bezproblémová a automatická neblokujúca synchronizácia zmien so serverom
Spoľahlivosť a výkon	Beta	"Zrelý" produkt	Populárny produkt
Škálovateľnosť	Automatické	Vyžaduje rozdelenie	Bezproblémová (viac inštancií serveru)
Bezpečnosť	Jednoduchšie a výkonnejšie zabezpečenie pre mobilné, webové a serverové SDK	Kaskádové pravidlá vyžadujúce samostatné overenie	Objekty sú vždy platné, pretože dodržiavajú schémy. Môžu byť šifrované AES-256
Autentifikácia	Firebase Authentication a Cloud Firestore Security Rules pre Android, iOS a JavaScript alebo Access Management pre server-side jazyky	Pravidlá databázy Firebase	End-to-end šifrovanie so ZeroKitom
Triggers	Áno, pomocou Cloud Functions	Pri zmene dát	Áno
Koncepty konzistencie	Okamžitá	Eventuálna, Okamžitá	Okamžitá
Podporované jazyky	Go, Java, JavaScript, JavaScript (Node.js), Objective-C, Python	Java, JavaScript, Objective-C	Java, Objective-C, React Native, Swift
Cena	Poplatky sú primárne pre operácie vykonávané vo vašej databáze (čítanie, zápis, vymazanie) a pri nižšej rýchlosti, šírke pásma a úložisko	Platí sa len za šírku pásma a pamäť, ale pri vyššej rýchlosti	Cena na základe používateľov a jadier, počnúc od 400 dolárov mesačne

Pre našu aplikáciu sme zvolili databázu Firebase jednak preto, že je jednoduchá na implementáciu, a taktiež že štruktúra našej databázy nebude zložitá, a teda model vo forme JSON-u nám vyhovuje. Ďalším dôvodom bolo to, že na rozdiel od Realm je zdarma (vzhľadom na náš rozsah) a nepotrebujeme si zaobstarávať server, na ktorom by táto databáza fungovala. Dôvod, prečo sme nezvolili Firestore je ten, že Firestore poskytuje navyše funkcie a vlastnosti, ktoré aj tak nevyužijeme a je zatiaľ vo verzii Beta.

6 FUNKČNÉ A NEFUNKČNÉ POŽIADAVKY

V tejto kapitole sú špecifikované funkčné a nefunkčné požiadavky pre mobilnú aplikáciu na správu financií. Popísaná je aj pomocou diagramu prípadov použitia.

6.1 Funkčné požiadavky

Aplikácia má:

- umožňovať identifikáciu a prihlásenie pomocou Google účtu
- umožňovať zobrazenie a úpravu údajov v profile
- umožňovať pridanie novej transakcie
- umožňovať zmazanie transakcie
- umožňovať zobrazenie transakcií podľa mesiaca
- rozdeľovať transakcie na príjmy a výdavky
- rozdeľovať transakcie podľa kategórií
- umožňovať zobraziť detail kategórie daného mesiaca
- umožňovať zobraziť detail transakcie
- fungovať aj v režime offline (bez prístupu na internet)
- ukladať transakcie do databázy na server
- umožňovať prihlásenie jedného užívateľa na viacerých zariadeniach súčasne

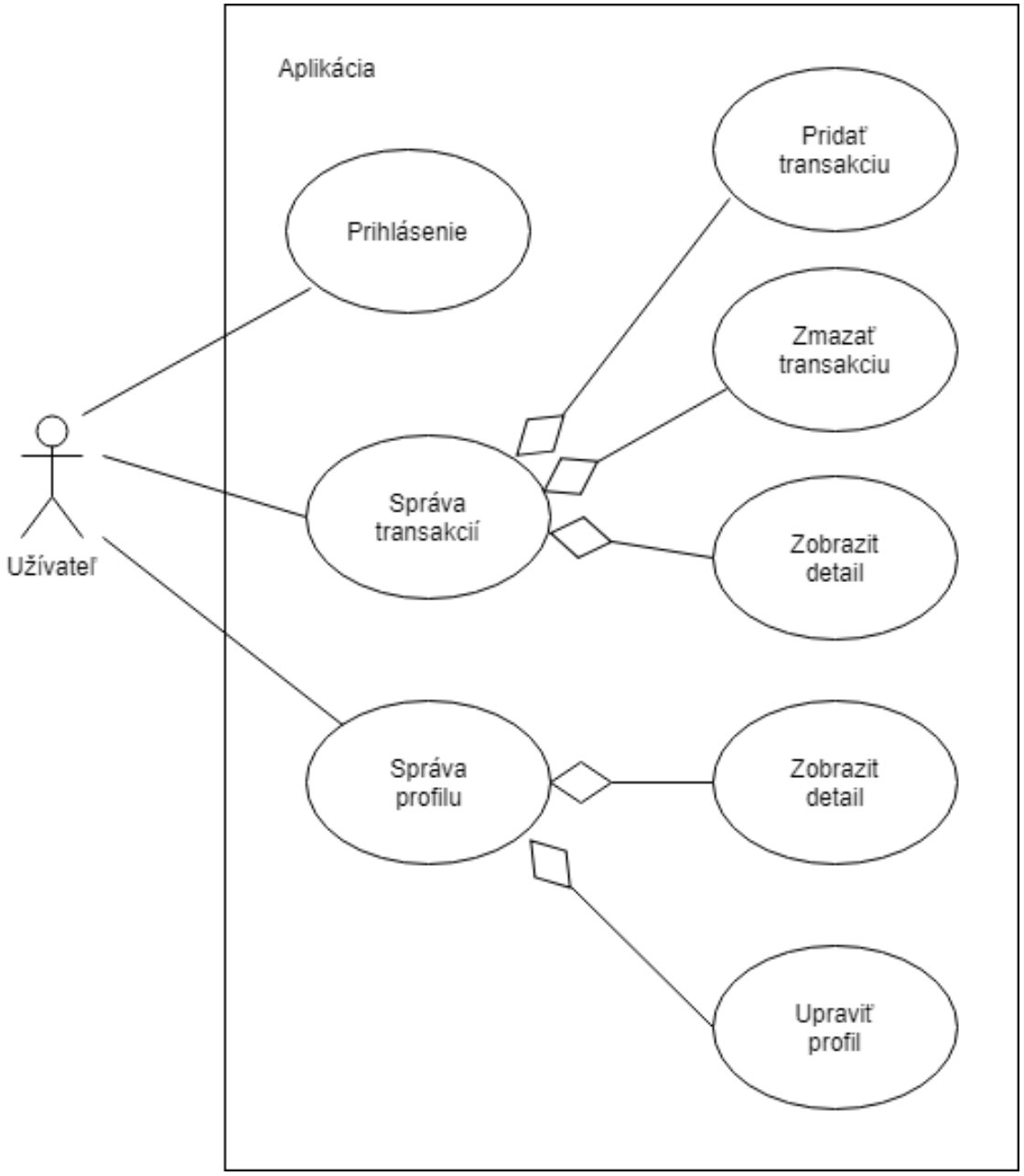
6.2 Nefunkčné požiadavky

Aplikácia bude:

- určená pre systém Android
- spustiteľná na zariadeniach s verziou API 21 a vyššou
- určená primárne pre smartphony, rozlíšenie do 1920x1080

6.3 Diagram prípadov použitia

V tejto podkapitole je popísaná funkcionalita aplikácie pomocou diagramu prípadov použitia znázorneného na nasledujúcom obrázku.



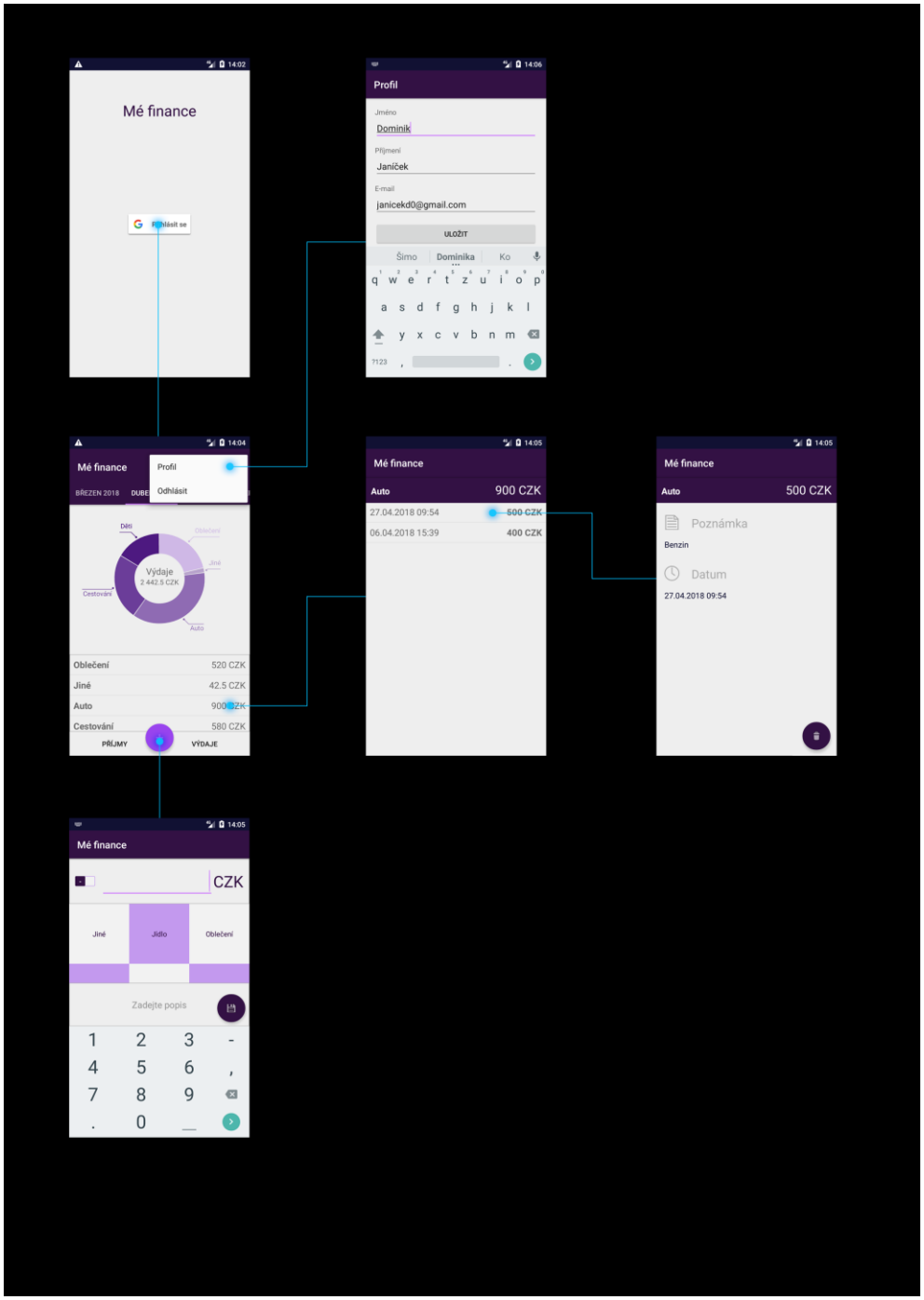
Obrázok 11: Diagram prípadov použitia

Aplikácia po prihlásení sa umožňuje správu transakcií a profilu. To znamená zobraziť už vytvorené transakcie podľa kategórií a mesiacov, zobrazovať detail transakcie, ďalej pridávať a mazať transakcie. Taktiež je užívateľovi umožnené zobraziť svoj profil a upraviť v ňom základné údaje.

7 MOBILNÁ APLIKÁCIA

V tejto kapitole je opísaná vytvorená Android aplikácia. Vývoj tejto aplikácie bol prispôsobený požiadavkám z predchádzajúcej kapitoly. Aplikácia je opísaná jak s pohľadu užívateľa, tak aj s pohľadu vývojára, kde sú popísané najdôležitejšie funkcie.

7.1 Popis aplikácie – s užívateľského pohľadu



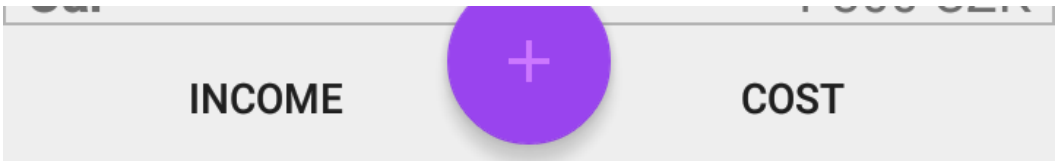
Obrázok 12: App flow

Po spustení aplikácie sa ako prvé zobrazí obrazovka s prihlásením. Prihlásenie je pomocou Google účtu. Po kliknutí na prihlasovacie tlačidlo sa na prihlásenie použije užívateľ, ktorý je prihlásený na zariadení. Ak je na danom zariadení viac kont, aplikácia vyzve užívateľa k výbere konta na prihlásenie. V prípade, že na danom zariadení nie je žiadne konto Google je užívateľ vyzvaní k prihláseniu sa alebo k vytvoreniu Google účtu. Ak sa užívateľ prihlasoval v minulosti a nebol odhlásený, použije sa naposledy prihlásené konto a úvodná obrazovka sa preskočí.

Po úspešnom prihlásení sa zobrazí obrazovka s transakciami v aktuálnom mesiaci. Mesiace sa prepínajú tzv. swapovaním zo strany na stranu alebo výberom mesiaca v hornej časti obrazovky.

Obrázok 13: TabLayout zo zoznam mesiacov

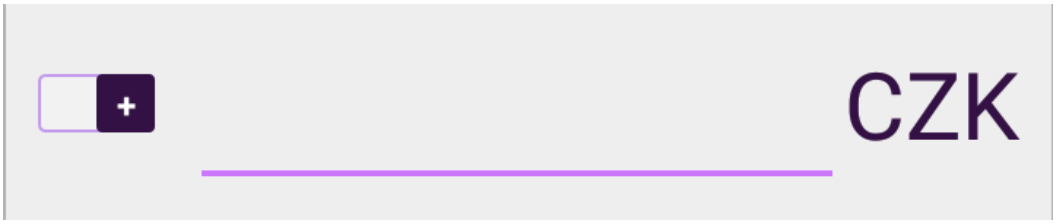
V strede obrazovky sa nachádza koláčový graf zobrazujúci percentuálny podiel jednotlivých kategórií s celkovej sumy. Pod grafom je zoznam jednotlivých kategórií so sumou. Medzi príjmami a výdavkami sa prepína pomocou tlačidiel v dolnej časti obrazovky.



Obrázok 14: Tlačidlá na výber medzi príjmami a výdavkami

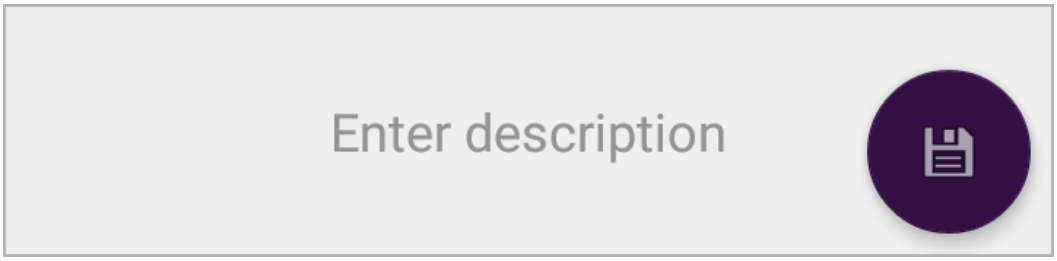
Po kliknutí na jednu z kategórií zo zoznamu sa zobrazí detail danej kategórie zo zoznamom jednotlivých transakcií a ich sumami. Pre zobrazenie detailu transakcie stačí kliknúť na jednu z nich. Obrazovka s detailom transakcie obsahuje názov kategórie a sumu v hlavičke a popis (ak bol nejaký popis zadaný pri tvorbe transakcie) a dátum vytvorenia v strede obrazovky. V pravej spodnej časti obrazovky sa nachádza tlačidlo na zmazanie aktuálnej transakcie. Po jeho stlačení sa zobrazí dialógové okno na potvrdenie zmazania. Transakcia sa zmaže a po návrate na hlavnú obrazovku sa aktualizuje stav transakcií.

Pre pridanie novej transakcie slúži fialové tlačidlo zo znakom + v spodnej časti hlavnej obrazovky. Po stlačení tohto tlačidla sa otvorí nové okno.



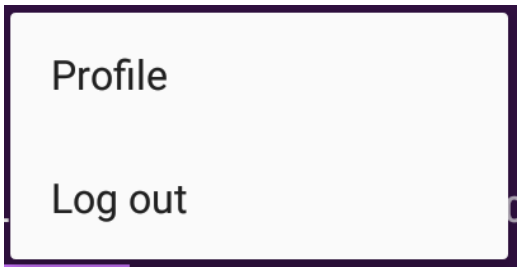
Obrázok 15: Voľba medzi príjmami a výdavkami a pole na zadanie sumy.

V hornej časti tohto okna je prepínač na voľbu medzi príjmami (+) a výdavkami (-). Vedľa tohto prepínaču je pole na zadávanie čiastky. V strede obrazovky sa nachádza zoznam s kategóriami, ktorý sa mení podľa toho, či sa jedná o príjmy alebo výdavky. V spodnej časti je pole na zadávanie popisu. To ako jediné nie je povinné. V prípade, že nie je zvolená kategória alebo zadaná suma, aplikácia vyzve aplikácia pri ukladaní užívateľa, aby dané pole vyplnil. Na uloženie transakcie slúži tlačidlo na pravej spodnej časti obrazovky (prípadne nad klávesnicou). Po uložení transakcie je užívateľ presmerovaný späť na hlavnú obrazovku.



Obrázok 16: Pole na vkladanie popisu a tlačidlo uloženia

Na hlavnej obrazovke sa ešte nachádza menu, kde užívateľ môže zobraziť svoj profil alebo sa odhlásiť. Po odhlásení sa aplikácia zavrie. Pri zobrazení profilu je užívateľ presmerovaný na obrazovku so základnými informáciami o jeho profile. Tieto informácie je možné editovať a následne uložiť.



Obrázok 17: Menu

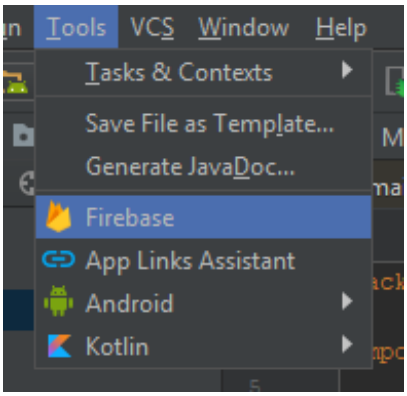
Celá aplikácia funguje po prihlásení aj v režime offline (bez internetu). Po spustení aplikácie v offline režime aplikácia upozorní užívateľa, že nie je pripojený k internetu. Všetky vykonané zmeny sa po opätovnom pripojení na internet aktualizujú zo serverom.

7.2 Popis aplikácie – opis kódu

V tejto časti sú popísané najdôležitejšie časti kódu aplikácie *My Finances*. Pre tvorbu tejto aplikácie sme si zvolili návrhový vzor MVVM a databázu Firebase. Ako prvé je popísaná implementácia prihlásenia Firebase Autentication pomocou Google účtu a funkcie na prácu z databázou. Ďalej je popísaný data binding a niektoré dôležité funkcie a vlastnosti.

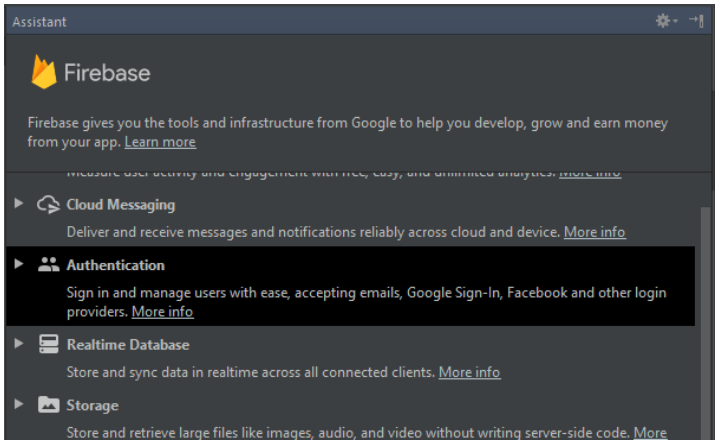
7.2.1 Login

Pridanie autentifikácie pomocou Google účtu do projektu je jednoduché. Android Studio obsahuje vstavaný nástroj pre Firebase, cez ktorý sa dá ľahko pridať Google autentifikácia. Stačí nasledovať sprievodcu, ktorého nájdeme na karte Tools pod názvom Firebase.



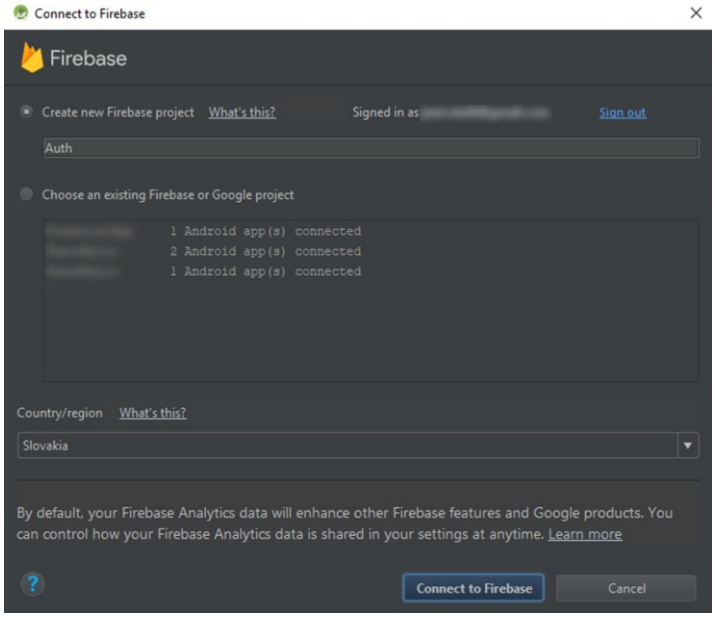
Obrázok 18: Umiestnenie Firebase asistenta

Po kliknutí na *Authentication* a *Email and password authentication* sa zobrazí ďalší krok, kde je nutné sa pripojiť k vytvorenej Firebase databáze tlačidlom *Connect to Firebase*. Po kliknutí sa otvorí prehliadač, v ktorom si zvolíme účet na pripojenie k Android Studiu, a potom službu Firebase.



Obrázok 19: Firebase assistant

Po pripojení k účtu Google sa otvorí okno s pripojením k projektu. Tu je možné zvoliť už existujúci projekt alebo vytvoriť nový. Po prihlásení alebo vytvorení nového Firebase projektu Android Studio stiahne potrebné súčasti ako google-services.json.



Obrázok 20: Okno pripojenia k Firebase

Po pripojení k Firebase projektu je potrebné pridať autentifikáciu, a to je možné buď ručne, alebo automaticky kliknutím na tlačidlo *Add Firebase Authentication to your app*. Android Studio pridá do build.gradle *classpath 'com.google.gms:google-services:3.1.0'* a do app/build.gradle *apply plugin: 'com.google.gms.google-services'* a *compile 'com.google.firebase:firebase-auth:11.0.4'*. Keďže sme si zvolili prihlásenie pomocou Google účtu ďalšie korky sme preskočili a v kroku 8 sprievodcu zvolili možnosť *Google Sign-in*, kde sa otvorí stránka s postupom pridania tohto spôsobu prihlásenia.

```
73    mAuth = FirebaseAuth.getInstance();
      ...
77    GoogleSignInOptions gso = new GoogleSignInOp-
↲     tions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
78          .requestIdToken(getString(R.string.default_web_client_id))
79          .requestEmail()
80          .build();
81
82    mGoogleApiClient = new GoogleApiClient.Builder(this)
83            .enableAutoManage(this, new GoogleApiC-
↲     lient.OnConnectionFailedListener() {
84                @Override
85                public void onConnectionFailed(@NonNull Connection-
↲     Result connectionResult) {
86                    Toast.makeText(LoginActivity.this,
↲     R.string.connection_failed,Toast.LENGTH_SHORT).show();
87                }
88            })
89            .addApi(Auth.GOOGLE_SIGN_IN_API, gso)
90            .build();
```

Zdrojový kód 1: Inicializácia FirebaseAuth, GoogleSignInOptions a GoogleApiClient

Postupovali sme podľa návodu na stránke a pridali všetky potrebné časti kódu ako sú ďalšie závislosti podľa aktuálnej verzie a iné. Do metódy *onCreate* sme pridali kód na inicializáciu *FirebaseAuth*, *GoogleSignInOptions* a *GoogleApiClient*.

Ďalej stačilo už len pridať zvyšné funkcie z návodu ako *signIn*, *onActivityResult* a *firebaseAuthWithGoogle*. Po úspešnom prihlásení sa vykoná kód vo funkcií *firebaseAuthWithGoogle* v časti *task.isSuccessful*. V našom prípade sme si tu stiahli aktuálneho užívateľa pomocou *mAuth.getCurrebtUser()* a uložili ho do databázy.

```
102   mAuthListener = new FirebaseAuth.AuthStateListener() {
103       @Override
104       public void onAuthStateChanged(@NonNull FirebaseAuth firebase-
↲     Auth) {
105           user = firebaseAuth.getCurrentUser();
106           if(user != null){
107               pending.set(false);
108
109               if(!MainTabActivity.isActive){
110                   Intent intent = MainTabActivi-
↲     ty.startIntent(LoginActivity.this);
111                   startActivity(intent);
112                   finish();
113               }
114           }
115       }
116   };
```

Zdrojový kód 2: Inicializácia AuthStateListener-a

Keďže po úspešnom prihlásení sa v našom prípade mala otvoriť nová aktivita, toto sme zabezpečili pomocou *AuthStateListener*. Tým sme zaistili aj to, že po opätovnom spustení aplikácie už nebude nutné sa znovu prihlasovať, ak sme sa predtým neodhlásili.

```
27    <com.google.android.gms.common.SignInButton
28        android:id="@+id/googleSigninBtn"
29        android:layout_width="wrap_content"
30        android:layout_height="wrap_content"
31        android:layout_centerHorizontal="true"
32        android:layout_centerVertical="true" />
```

Zdrojový kód 3: Google SignIn Button v activity_login.xml

```
95    signInButton.setOnClickListener(new View.OnClickListener() {
96        @Override
97        public void onClick(View v) {
98            pending.set(true);
99            signIn();
100       }
101   });
```

Zdrojový kód 4: Nastavenie OnClickListener-a na prihlasovacie tlačidlo

Do súboru xml sme už len pridali *SignInButton* z google balíku a nastavili naň *OnClickListener,* v ktorom sme zavolali funkciu *signIn().*

7.2.2 DB manager

Pre pridanie podpory Firebase Realtime Database sme postupovali podobne ako v predošlom prípade. Otvorili si Firebase asistenta, ale tentoraz si spustili sprievodcu pre Realtime Database. Po kliknutí na *Save and retriev data* sa otvorí step by step inštalácia. Keďže už sme mali aplikáciu spojenú s vytvoreným projektom, prvý krok už nebolo potrebné opakovať a sprievodca by teda mal ukazovať stav spojenia ako *Connected.* Po kliknutí na tlačidlo *Add the Realtime Database to your app* sa otvorilo okno, kde sme akceptovali zmeny a týmto krokom sme našu aplikáciu spojili s databázou Firebase.

Pre prácu s databázou sme si vytvorili novú triedu, ktorá obsahuje statické metódy na správu databáze. Ako prvé sme si vytvorili statické premenné, ktoré obsahujú referenciu na databázu a autentifikáciu.

```
29    private static DatabaseReference userRef;
30    public static DatabaseReference incomeRef;
31    public static DatabaseReference costRef;
32
33    private static FirebaseAuth firebaseAuth = Firebase-
↲     Auth.getInstance();
```

Zdrojový kód 5: Premenné DBManager-a

Ďalej bolo potrebné vytvoriť metódu na inicializáciu databázy, kde sa do vytvorených premenných uložili referencie na príjmy, výdaje a užívateľov a povolili ukladanie dát do zariadenia, aby sme tak mohli využívať aplikáciu aj v offline režime.

```
42    public static void initDB(){
43        FirebaseDatabase fire = FirebaseDatabase.getInstance();
44        fire.setPersistenceEnabled(true);
45
46        userRef = fire.getReference(USERS);
47        incomeRef = fire.getReference(INCOMES);
48        costRef = fire.getReference(COSTS);
49    }
```

Zdrojový kód 6: Inicializácia databázy

Potom sme si vytvorili funkcie na správu užívateľa a to na kontrolu, či je užívateľ v databáze a jeho prípadné pridanie.

```
82    public static void ckeckUserInDBandSave(FirebaseUser user, Context
      context){
83
84        final User me = new User(user);
85        final Context con = context;
86
87        user-
      Ref.child(user.getUid()).addListenerForSingleValueEvent(new Value-
      EventListener() {
88            @Override
89            public void onDataChange(DataSnapshot dataSnapshot) {
90
91                User loadedUser;
92
93                if(dataSnapshot.exists()){
94                    loadedUser = dataSnapshot.getValue(User.class);
95                    Toast.makeText(con,
      R.string.welcome_back,Toast.LENGTH_SHORT).show();
96                }else {
97                    saveUserToDB(me);
98                    loadedUser = me;
99                }
100               MyShared.setUser(con,loadedUser);
101           }
102
103           @Override
104           public void onCancelled(DatabaseError databaseError) {
105               Log.w("dbError", "loadPost:onCancelled", databaseEr-
      ror.toException());
106           }
107       });
108   }
```

Zdrojový kód 7: Kontrola existencie užívateľa a jeho prípadné pridanie

Do tejto funkcie vstupujú parametre ako užívateľ načítaný z autentifikácie Firebase a kontext aktivity. Tieto parametre sa uložia do lokálnych premenných spustí sa *ListenerForSingleValueEvent* podľa ID užívateľa. V metóde *onDataChange()* sa kontroluje, či sa vrátili nejaké údaje zo serveru. Ak áno, získa sa hodnota užívateľa a uloží sa do SharedPreferences. Ak zo serveru nič neprišlo, uloží sa tam užívateľ zo vstupného parametru a až ten sa uloží do SharedPreferences. Metóda *saveUserToDB()* uloží na server užívateľa podľa ID pomocou metódy *setValue()*.

Táto trieda obsahuje aj ďalšie metódy na uloženie a zmazanie transakcie.

```
116   public static void saveTransactionToDB(String userUid, Transaction
↵     transaction, Boolean income){
117       DatabaseReference dbRef = incomeRef;
118       if(!income){
119           dbRef = costRef;
120       }
121
122       String monthYear = MyDa-
↵     te.longTimeToMonthYear(transaction.getTransactionDate());
123
124       DatabaseReference ref = dbRef.child(userUid)
125               .child(monthYear)
126               .push();
217
128       transaction.setUid(ref.getKey());
129
130       ref.setValue(transaction);
131   }
```

Zdrojový kód 8: Funkcia na uloženie transakcie do DB

Táto funkcia uloží pod užívateľa transakciu ktorú príjme ako vstupný parameter. Ako prvé sa do premennej typu *DatabaseReference* uloží referencia na príjmy alebo výdaje. Ďalej sa z dátumu transakcie vytvorí kľúč, podľa ktorého sa daná transakcia uloží do DB. Následne sa podľa ID užívateľa a vytvoreného kľúča vytvorí referencia v DB. To je nutné kvôli získaniu unikátneho ID transakcie. Toto ID sa následne uloží do transakcie a tá sa potom uloží do DB.

```
137   public static void removeTransaction(String uid, Transaction
↵     transaction) {
138       String date = MyDa-
↵     te.longTimeToMonthYear(transaction.getTransactionDate());
139       String transID = transaction.getUid();
140       DatabaseReference dbRef;
141
142       if(transaction.getCategory() < Category.OFFSET){
143           dbRef = DBManager.incomeRef;
144       }else{
145           dbRef = DBManager.costRef;
146       }
147
148       dbRef.child(uid).child(date).child(transID).removeValue();
149   }
```

Zdrojový kód 9: Funkcia na zmazanie transakcie s DB

Funkcia *removeTransaction* zmaže užívateľovi transakciu, ktorú prijala ako vstupný parameter. Najskôr sa z transakcie získa dátum a s toho kľúč, podľa ktorého bude potom transakcia zmazaná. Ďalej sa získa ID transakcie a podľa kategórie sa do premennej typu *DatabaseReference* uloží referencia na príjmy alebo výdaje. Následne podľa získaných informácií sa transakcia zmaže z DB.

Pre realtime sledovanie zmien v databázy sme použili *ValueEventListener* vo VM fragmentu.

```
96    protected void onCleared() {
      ...
99        incomeRef.removeEventListener(listenerForIncomes);
      ...
101   }
      ...
106   listenerForIncomes = new ValueEventListener() {
107       @Override
108       public void onDataChange(DataSnapshot dataSnapshot) {
109           if(dataSnapshot.exists()){
110               processData(INCOMES, Transac-
      tion.transactionListFromFirebase(dataSnapshot));
111               setIncomesOrCost(isActiveIncomes);
112           }
113       }
114
115       @Override
116       public void onCancelled(DatabaseError databaseError) {
117           Log.w("dbError", "loadPost:onCancelled", databaseEr-
      ror.toException());
118       }
119   };
      ...
139   incomeRef.addValueEventListener(listenerForIncomes);
```

Zdrojový kód 10: Príklad listenera pre sledovanie zmien v DB

V tele listenera sme v metóde *onDataChange()* skontrolovali, či sa nám nejaké dáta vrátili a ak áno, tak sme zavolali funkciu na ich spracovanie a updatovali UI, ak to bolo potrebné. Následne sme tento listener priradil k referencií na Firebase DB. Taktiež bolo nutné odstrániť tento listener ak sa zničí VM, a to pomocou metódy *removeEventListener()* v metóde *onCleared()*.

7.2.3 Data Binding a LiveData

V tejto podkapitole je popísaný príklad DataBindingu v spojení s VM a taktiež sledovanie zmien pomocou LiveData a Observable. Ako prvé je však potrebné povoliť DataBinding v projekte a to pridaním nasledujúceho kódu do app/build.gradle.

```
3     android {
      ...
20        dataBinding {
21            enabled = true
22        }
23    }
```

Zdrojový kód 11: Povolenie DataBindingu

```
3       <data class="UserProfileDataBinding">
4           <import type="android.view.View"/>
5           <variable
6               name="vm"
7               ty-
↵   pe="cz.utb.fai.dodo.financialapp.ui.profile.UserProfileViewModel"/>
8       </data>
    ...
33  <EditText
34      android:id="@+id/editTextName"
        ...
38      android:inputType="textPersonName"
39      android:text="@={vm.name}" />
    ...
69  <Button
70      android:id="@+id/buttonSaveChanges"
    ...
75      android:onClick="@{vm::save}"/>
```

Zdrojový kód 12: Príklad Data Binding-u v xml súbore

Pri Data Bindingu je potrebné v xml súbore pridať na začiatok kód, ktorý vytvorí triedu z názvom, ktorý je v atribúte *class*. V tomto prípade to je *UserProfileDataBinding*. Ďalej je potrebné vytvoriť premennú (variable), ktorá bude odkazovať na triedu VM z atribútu *type.* V kóde potom už len stačí napísať názov premennej vytvorenej v xml súbore (*vm*) a za bodku názov premennej z VM. Tento kód musí byť obklopený špeciálnymi znakmi a to @{ } v prípade jednosmerného Data Binding-u a @={ } v prípade obojsmerného Data Binding-u ako je to na riadkoch 39 a 75 zdrojového kódu č. 12. Všetky bindované premenné musia mať vo VM verejné gettre a settre (v prípade že sa jedná o jednosmerný Data Binding stačí len getter), ak sú tieto premenné *private*. *Public* premenné nepotrebujú gettre ani settre. V oboch prípadoch ale pohľad nesleduje zmeny na premenných, len ich pri inicializácií načíta a ďalej ich už nesleduje, na to je potrebné použiť iný spôsob, ako napríklad Observable alebo LiveData. Ak sa jedná o bindovanie metódy používa sa namiesto bodky dvojbodka. Metóda potom musí mať nasledujúci tvar *public void methodName(View view).* Aby tento Data Binding fungoval bolo ešte potrebné pridať nasledujúci kód do aktivity.

```
32  UserProfileDataBinding userProfileDataBinding = DataBindingU-
↵   til.setContentView(this,R.layout.activity_user_profile);
33  UserProfileViewModel viewModel = ViewModelProvi-
↵   ders.of(this).get(UserProfileViewModel.class);
34  userProfileDataBinding.setVm(viewModel);
```

Zdrojový kód 13: Nastavenie DataBindingu v aktivite

Tento kód nastaví layout pohľadu a taktiež nami vytvorenú premennú *vm*, do ktorej uloží VM. Trieda s VM musela byť rozšírená o triedu *AndroidViewModel* a získaná pomocou *ViewModelProviders* aby bola zaručená jej správna činnosť.

Použite Observable je jednoduché. Kód v xml súbore a v aktivite ostáva rovnaký, menil sa len kód vo VM, kde sme namiesto klasickej premennej použili tzv. Observable premennú napr.: *ObservableBoolean* alebo *ObservableField*. V tomto prípade už nebolo nutné vytvárať getter ale podmienkou je, aby táto premenná bola *public*.

Ďalším spôsobom ako sledovať zmenu hodnoty akéhokoľvek typu bolo použite MutableLiveData. V triede VM sme si vytvorili premennú tohto typu napr.: *private MutableLiveData<List<String>> months = new MutableLiveData<>();* a k nej getter. Hodnota sa nastavuje funkciou *months.setValue( );*. V aktivite následne na vytvorený getter nastavíme *Observer*.

```
95    viewModel.getMonths().observe(this, new Observer<List<String>>(){
96        @Override
97        public void onChanged(@Nullable List<String> strings) {
98            init(strings);
99        }
100   });
```

Zdrojový kód 14: Nastavenie Observer-u

V tele potom môže byť hocijaký kód v našom prípade to bola inicializácia UI. Aby sme mohli LiveData a VM použiť, bolo nutné pridať ďalšie závislosti do app/build.gradle.

```
32    // ViewModel and LiveData
33    implementation "android.arch.lifecycle:extensions:1.1.1"
34    // alternatively, just ViewModel
35    implementation "android.arch.lifecycle:viewmodel:1.1.1"
36    // alternatively, just LiveData
37    implementation "android.arch.lifecycle:livedata:1.1.1"
38
39    annotationProcessor "android.arch.lifecycle:compiler:1.1.1"
```

Zdrojový kód 15: LiveData a ViewModel dependencies

7.2.4 RecyclerView a Adapter

Na zobrazovanie dát sme použili RecyclerView, ktorý sa do xml súboru vložil ako každý iný layout, bolo ale nutné k nemu vytvoriť ďalší xml súbor. Ten slúžil ako šablóna pri vkladaní položiek do RecyclerView-u.

```
8     <LinearLayout
9         android:id="@+id/listItemLayout"
10        android:layout_width="match_parent"
11        android:layout_height="wrap_content"
12        android:padding="8dp">
13
14            <TextView
15                android:id="@+id/textViewCategoryName"
16                android:layout_width="wrap_content"
17                android:layout_height="wrap_content"
18                android:textSize="18sp"
```

```
                android:textStyle="bold" />

            <TextView
                android:id="@+id/textViewPriceSum"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textAlignment="viewEnd"
                android:textSize="18sp" />
    </LinearLayout>
```

Zdrojový kód 16: Príklad šablóny pre RecyclerView

Ďalej bolo potrebné preň vytvoriť adaptér.

```
    public class AdapterCategory extends Recycler-
    View.Adapter<AdapterCategory.ViewHolder>{

        private List<CategorySimple> items;
        private IAdapterItemClicked<CategorySimple> listener;

        public AdapterCategory(List<CategorySimple> items, IAdapterI-
    temClicked<CategorySimple> listener) {
            this.items = items;
            this.listener = listener;
        }

        @Override
        public ViewHolder onCreateViewHolder(ViewGroup parent, int
    viewType) {
            View v = LayoutInflater.from(parent.getContext())
    .inflate(R.layout.list_item_category,parent,false);

            return new ViewHolder(v);
        }

        @Override
        public void onBindViewHolder(ViewHolder holder, int position)
    {
            final CategorySimple categorySimple = items.get(position);

            String categoryName = Catego-
    ry.getCategoryName(categorySimple.getCategory());
            String sum = NumberFormat-
    ter.formateNumber(categorySimple.getPriceSum(), ' ') + " " +
    Transaction.CURRENCY;

            holder.categoryName.setText(categoryName);
            holder.priceSum.setText(sum);

            holder.itemView.setOnClickListener(new
    View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    listener.onItemClicked(categorySimple);
                }
            });
        }

        @Override
        public int getItemCount() {
            return items.size();
```

```
61          }
62
63          public void setNewList(List<CategorySimple> list){
64              this.items=list;
65              notifyDataSetChanged();
66          }
67
68          class ViewHolder extends RecyclerView.ViewHolder{
69              TextView categoryName, priceSum;
70
71              ViewHolder(View itemView) {
72                  super(itemView);
73                  categoryName = item-
↲       View.findViewById(R.id.textViewCategoryName);
74                  priceSum = item-
↲       View.findViewById(R.id.textViewPriceSum);
75              }
76          }
77      }
```

Zdrojový kód 17: Adaptér pre RecyclerView

Na začiatku triedy sú uložené premenné a konštruktor, ktorý preberá list (v tomto prípade triedy *CategorySimple*, čo je pomocná trieda, ktorá v sebe drží informácie o zobrazovanej položke) a interface na obsluhu kliknutia na položku. Nasledujú prepísané (Override) metódy ako *onCreateViewHolder,* v ktorej sa nastavuje layout pre aktuálnu položku a *onBindViewHolder*, kde sa nastavuje obsah položky a *OnClickListener*. Ďalej je tu metóda *getItemCount*, ktorá vracia počet položiek v RecyclerViewe a nami vytvorená metóda *setNewList*, v ktorej nastavujeme nový obsah RecyclerViewu a aktualizujeme ho (napríklad pri načítaní nových dát z databázy). Na konci je vnorená trieda (*ViewHolder*), ktorá v sebe drží View ku každej položke RecyclerView-u.

Všetky adaptéry použité v našej aplikácií sú si dosť podobné, líšia s len v type zobrazovaných položiek a použitom layoute.

Po vytvorení adaptéra nám už len ostávalo ho priradiť k u konkrétnemu RecyclerViewu a to sme urobili v aktivite (prípadne vo fragmente).

```
97      public void onStart() {
        ...
100         viewModel.getPrices().observe(this, observer);
        ...
105     }
        ...
120     RecyclerView recyclerView = fragmentDataBin-
↲       ding.transactionRecycleView;
121
122     recyclerView.setHasFixedSize(true);
123     recyclerView.setLayoutManager(new LinearLayoutMana-
↲       ger(getActivity()));
124     recyclerView.addItemDecoration(new DividerItemDecora-
↲       tion(getActivity(), LinearLayoutManager.VERTICAL));
```

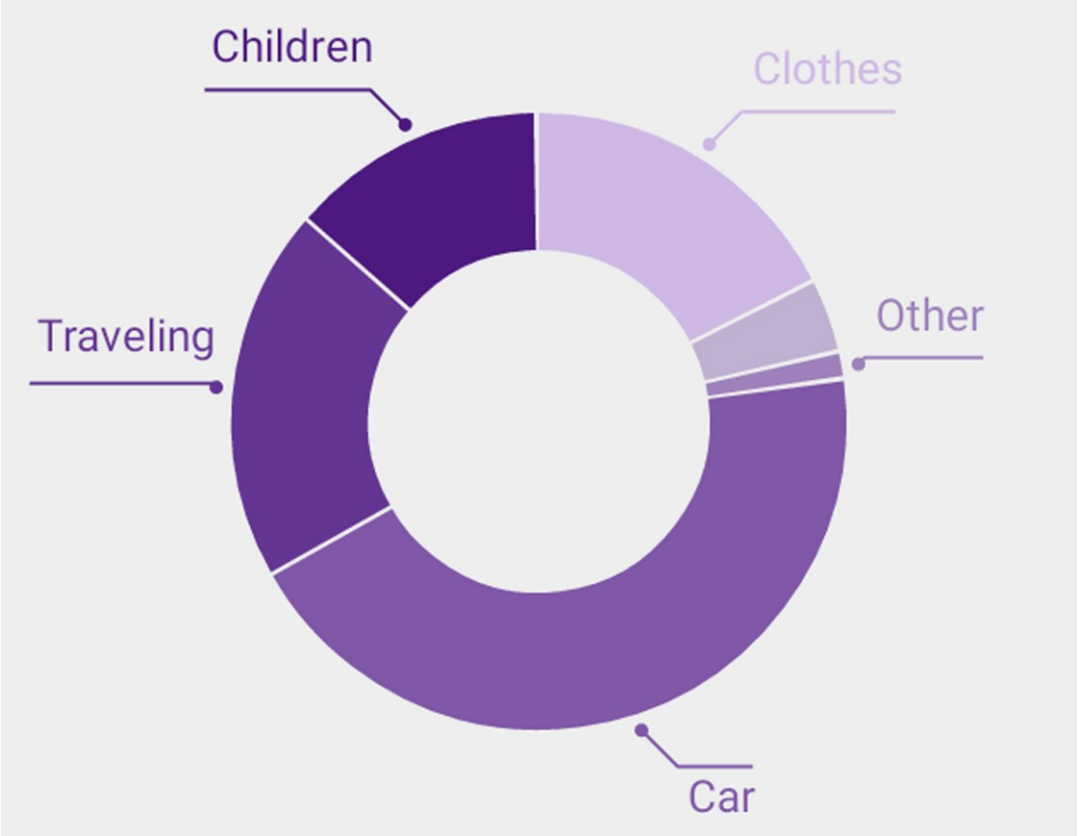
```
125   final AdapterCategory adapter = new AdapterCategory(new Array-
↲     List<CategorySimple>(), this);
126
127   recyclerView.setAdapter(adapter);
128
129   observer = new Observer<HashMap<Integer, Double>>() {
130       @Override
131       public void onChanged(@Nullable HashMap<Integer, Double> inte-
↲     gerDoubleHashMap) {
132           if (integerDoubleHashMap == null) {
133               adapter.setNewList(new ArrayList<CategorySimple>());
134           }else {
135               List<CategorySimple> categorySimples = CategorySim-
↲     ple.mapToList(integerDoubleHashMap);
136               adapter.setNewList(categorySimples);
      ...
139           }
140       }
141   };
```

Zdrojový kód 18: Nastavenie RecyclerViewu v aktivite

Ako prvé sme si uložili RecyclerView do lokálnej premennej a začali ho nastavovať. Parameter *setHasFixedSize* sa nastavuje na *true* v prípade, že vieme že sa veľkosť RecyclerViewu nebude meniť. Ďalej sme mu nastavili LayoutManagera. Vo všetkých prípadoch bol tento Manager typu Linear iba v aktivite na pridávanie transakcií bol typu Grid. Potom sme nastavili oddeľovaciu čiaru medzi položkami metódou *addItemDecoration.* Ako ďalšie sme si vytvorili adaptér, ktorý sme mu potom nastavili (*setAdapter()*). Nakoniec sme vytvorili *Observer* na sledovanie zmien, ktorý pri zmene nastavil adaptéru nový obsah a v *onStart()* metóde ho priradili. Taktiež bolo potrebné tento *Observer* odstrániť a v metóde o*nStop()* nasledujúcim kódom: *viewModel.getPrices().removeObserver(observer);*

7.2.5 PieGraph

Pre grafické znázornenie podielu jednotlivých kategórií na celkovej sume v jednotlivých mesiacoch sme sa rozhodli použiť koláčový graf od tvorcu „razerdp“.



Obrázok 21: PieView

Pridanie bolo jednoduché, stačilo do app/build.gradle pridať závislosť *compile 'com.github.razerdp:AnimatedPieView:{latestVersion}'*. Vytvorili sme si novú triedu, ktorá bude reprezentovať položku tohto grafu.

```
11    <public class PieGraph implements IPieInfo {
12
13        private float percentage;
14        private int color;
15        private String description;
16
17        public PieGraph(CategorySimple categorySimple, double sum, int
↵     color) {
18            this.percentage = ((float) categorySimple.getPriceSum() /
↵     (float) sum) * 100;
19            this.color = color;
20            this.description = Catego-
↵     ry.getCategoryName(categorySimple.getCategory());
21        }
22
23        @Override
24        public double getValue() {
25            return percentage;
26        }
27
28        @Override
29        public int getColor() {
30            return color;
31        }
```

```
32
33          @Override
34          public String getDesc() {
35              return description;
36          }
37      }
```

Zdrojový kód 19: Trieda reprezentujúca položku v grafe

Táto trieda obsahuje informácie o percentuálnom podiele z grafu, farbu a popis. V konštruktore si predávame informácie ako celková suma, farba a parameter typu *CategorySimple*. Z týchto údajov si vypočítame percentuálny podiel, uložíme farbu a taktiež popis. Ďalšie metódy sú gettre k týmto premenným. Následne sme do xml súboru pridali nový View *com.razerdp.widget.animatedpieview.AnimatedPieView*. Potom sme už len potrebovali v fragmente nastaviť údaje a vlastnosti tohto grafu.

```
169   private void setPieView(double sum, List<CategorySimple> category-
↲     Simples) {
170       animatedPieView = fragmentDataBinding.animatedPieView;
171       AnimatedPieViewConfig config = new AnimatedPieViewConfig();
172
173       List<Integer> colors = prepearColors(categorySimples.size());
174
175       config.startAngle(-90)
176               .splitAngle(1)
177               .strokeMode(true)
178               .drawText(true)
179               .textSize(26)
180               .duration(700);
181
182       int i = 0;
183       for (CategorySimple cs : categorySimples) {
184           config.addData(new PieGraph(cs, sum, colors.get(i)));
185           i++;
186       }
187       animatedPieView.applyConfig(config);
188   }
```

Zdrojový kód 20: Nastavenie PieView

Táto funkcia nastavuje PieGraph a spúšťa sa pri zmene obsahu (a teda aj pri prvom spustení). Ako prvé sa získa inštancia tohto grafu a vytvorí sa konfiguračná premenná pre graf. Ako ďalšie sa zistia farby, podľa počtu kategórií a uložia. Potom sa vytvorí konfigurácia kde sa nastaví počiatočný uhol, uhol medzery, mód (koláčoví alebo prstencoví) grafu, vykresľovanie textu (áno alebo nie), veľkosť textu a dĺžka animácie. Po vytvorení tejto konfigurácie v cykle pridajú jednotlivé položky a následne sa takto vytvorená konfigurácia aplikuje. Graf sa potom spustí metódou *start()*.

7.2.6 Triedy User, Transaction a iné

Trieda *User* je základná trieda, ktorá v sebe drží informácie o užívateľovi a to jeho meno, priezvisko, e-mail, adresu na fotku a jeho ID. Obsahuje aj niekoľko konštruktorov pre vytvorenie užívateľa, a taktiež gettre a settre pre všetky premenné (okrem premennej s ID, tá má len getter). Ďalej obsahuje aj metódy na konverziu medzi triedou *User* a reťazcom vo forme Json.

Trieda *Transaction* obsahuje niekoľko verejných konštánt typu *String* a privátnych premenných ako ID transakcie, dátum transakcie a dátum jej vytvorenia, kategóriu, popis a samozrejme sumu. Všetky tieto premenné majú vytvorené gettre a settre (okrem dátumu vytvorenia transakcie, ktorý má len getter). Taktiež táto trieda obsahuje konštruktor a niekoľko statických metód na konverziu na a z reťazca vo formáte Json ako aj konverziu z Json reťazca na zoznam transakcií. Ďalej obsahuje aj statickú metódu na konverziu z dátového typu *DataSnapshot*, ktorý dostane od databáze Firebase, na zoznam transakcií.

```
146   public static List<Transaction> transactionListFromFireba-
↲     se(DataSnapshot dataSnapshot) {
147
148       List<Transaction> transactions = new ArrayList<>();
149
150       for(DataSnapshot snapshot : dataSnapshot.getChildren()){
151           Transaction transaction = snap-
↲     shot.getValue(Transaction.class);
152           transactions.add(transaction);
153       }
154
155       return transactions;
156   }
```

Zdrojový kód 21: Funkcia na konverziu z DataSnapshot na zoznam transakcií

Táto funkcia prevádza výsledok z databáze Firebase vo forme *DataSnapshot* na zoznam transakcií. V tele funkcie je cyklus, ktorý postupne prechádza výsledok z databáze a prevádza tieto výsledky na triedu *Transaction*. Tie potom vkladá do zoznamu a vráti ho ako výstup z funkcie.

Ďalej sú v projekte aj ďalšie triedy ako *MyDate*, ktorá obsahuje len statické metódy na konverziu dátumu z typu *Long* na reťazec vo formáte *Rok_Mesiac* (v číselnej podobe – slúži ako kľúč pri ukladaní do DB) a zistenie čísla mesiaca z tohto reťazca. Taktiež na zisťovanie názvu mesiaca podľa tohto čísla. Potom je tu aj trieda *NumberFormatter* na formátovanie čísla pri zobrazovaní v UI. A trieda *MyShared* na ukladanie a získavanie informácií zo SharedPreferences. Napríklad ukladanie a načítavanie užívateľa (*User*), ukladanie

a načítavanie dát typu *Boolean* na zisťovanie, či bolo otvorené dialógové okno, alebo aký typ transakcií (príjmy alebo výdavky) sa má zobraziť.

Projekt obsahuje ešte ďalšie triedy, ktoré slúžia napr. ako konvertory alebo sú v nich uložené rôzne statické premenné. Taktiež obsahuje množstvo layoutov, aktivít a tried typu ViewModel, ktoré tu neboli opísané a rôzne zdroje ako farby, reťazce, štýly, obrázky a iné.

ZÁVER

Táto diplomová práca sa zaoberala porovnávaním architektúr pre tvorbu softwaru, rozdielom medzi klasickou (relačnou) databázou a databázou reálneho času, a taktiež porovnávaním predstaviteľov tohto typu databázy.

Na začiatku teoretickej časti bol popísaný systém Android jeho architektúra, a taktiež čo je to aktivita a fragment. Ďalej boli v teoretickej časti popísané návrhové (architektonické) vzory pre vývoj softwaru ako MVC, MVP a MVVM a ich jednotlivé časti. Ďalšia kapitola bola venovaná popisu a porovnávaniu databáz a to rozdielom medzi klasickou relačnou databázou a databázou reálneho času a popisom predstaviteľov tohto typu databáze. Konkrétne to boli databázy Firebase, Cloud Firestore a Realm.

Pri štúdiu rozdielov medzi relačnou a databázou reálneho času sme zistili, že databázy reálneho času sú veľmi efektívnym nástrojom pri čítaní dát v tom prípade, kde je nevyhnutné pozorovať zmeny v čase. Umožňujú sledovať zmeny na daných objektoch bez nutnosti opätovného dopytovania napr. pomocou REST API, pomocou ktorého sa obvykle pristupuje k relačným databázam. To je najväčšia výhoda oproti klasickým relačnom databázam, ktoré pracujú štýlom dotaz-výsledok. V prípade, že by naša aplikácia bola multiplatformná pre zápis dát do realtime databázy by bolo lepšie použiť REST API. A to z toho dôvodu, že pri zápise cez REST API možno vcelku efektívne vykonať rovnakú prácu pre všetky možné platformy (Android, iOS, web) na serveri. Inými slovami, Listener by sme použil u databázy reálneho času na čítanie dát a pre zápis by sme zvolil klasické REST API. Tým by sa nám skrátil kód a zjednodušila údržba.

Praktická časť sa skladala z niekoľkých častí, ako porovnávanie návrhových vzorov a predstaviteľov databázy reálneho času a tvorby demonštračnej mobilnej aplikácie. V prvej kapitole praktickej časti boli porovnané spomenuté architektonické návrhové vzory, ich výhody a nevýhody a rozdiely medzi nimi. Pre tvorbu mobilnej aplikácie bol zvolený návrhový vzor MVVM. Hlavným dôvodom výberu tohto návrhového vzoru bola možnosť využiť databinding pomocou LiveData. Ďalej taktiež odpadá nutnosť písať zložité rozhrania (Interface) a možnosť viazať View na udalosti, ktoré vyvolá ViewModel.

Ďalšia kapitola sa venovala porovnávaniu zvolených databáz reálneho času, rozdielom vo flexibilite, aktualizácií dát, škálovateľnosti, dopytovaniu a iným. Našou voľbou bola databáza Fiebase, a to hlavne s toho dôvodu, že nám vyhovovala štruktúra ukladania dát vo forme Json. Ďalším dôvodom bolo to, že na rozdiel od Realm je zdarma a nepotrebujeme si

zaobstarávať server. Dôvod, prečo sme nezvolili Firestore je ten, že Firestore poskytuje navyše funkcie a vlastnosti, ktoré aj tak nevyužijeme a je zatiaľ vo verzií Beta.

Zvyšok praktickej časti sa venoval mobilnej aplikácií, kde boli uvedené funkčné a nefunkčné požiadavky, a taktiež diagram prípadov použitia. Posledná kapitola bol venovaná aplikácií, a to z pohľadu užívateľa, kde boli popísané jednotlivé obrazovky a možnosti interakcie a aj z pohľadu vývojára. To obsahovalo popis tvorby jednotlivých častí aplikácie ako tvorba aktivity na prihlasovanie užívateľa, popis triedy na prácu s databázou, použité možnosti bindovania, tvorba adaptérov pre RecyclerView a iných pomocných tried na konverziu, ukladanie do pamäti telefónu a iné.

ZOZNAM POUŽITEJ LITERATÚRY

[1] Android. AandroidFórum.cz [online]. 2009 [cit. 2018-03-02]. Dostupné z: http://wiki.androidforum.cz/index.php/Android

[2] About the platform: Overview. Android Developers [online]. Mountain View: Google, 2018 [cit. 2018-03-04]. Dostupné z: https://developer.android.com/about/

[3] About the platform: Technology. Android Developers [online]. Mountain View: Google, 2018 [cit. 2018-03-04]. Dostupné z: https://developer.android.com/guide/platform/

[4] Kernel. Source [online]. Mountain View: Google, 2017 [cit. 2018-03-05]. Dostupné z: https://source.android.com/devices/architecture/kernel/

[5] Android Interfaces and Architecture. Source [online]. Mountain View: Google, 2017 [cit. 2018-03-10]. Dostupné z: https://source.android.com/devices/#Hardware%20Abstraction%20Layer

[6] ART and Dalvik. Source [online]. Mountain View: Google, 2017 [cit. 2018-03-10]. Dostupné z: https://source.android.com/devices/tech/dalvik/

[7] Distribution dashboard. Developers [online]. Mountain View: Google, 2017 [cit. 2018-03-11]. Dostupné z: https://developer.android.com/about/dashboards/

[8] Activity. Developers [online]. Mountain View: Google, 2017 [cit. 2018-03-15]. Dostupné z: https://developer.android.com/reference/android/app/Activity

[9] BOUBA, Radek. Seriál návrhových vzorů – 1. díl. Programujte.com [online]. , 1 [cit. 2018-03-16]. Dostupné z: http://programujte.com/clanek/2012032900-serial-navrhovych-vzoru-1-dil/

[10] SINHAL, Ankit. MVC, MVP and MVVM Design Pattern. In: Medium [online]. San Francisco: Medium, 2017 [cit. 2018-03-18]. Dostupné z: https://medium.com/@ankit.sinhal/mvc-mvp-and-mvvm-design-pattern-6e169567bbad

[11] MVC architektura. ITnetwork.cz [online]. 2013 [cit. 2018-03-18]. Dostupné z: https://www.itnetwork.cz/navrh/mvc-architektura-navrhovy-vzor/

[12] BERNARD, Borek. Úvod do architektury MVC. Zdrojak.cz [online]. 2009 [cit. 2018-03-18]. Dostupné z: https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

[13] MAXWELL, Eric. The MVC, MVP, and MVVM Smackdown. Realm [online]. Copenhagen: Realm, 2017 [cit. 2018-03-18]. Dostupné z: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

[14] BILÍK, David. Úvod do MVP na Androidu. In: Ackee [online]. Praha: Ackee, 2016 [cit. 2018-03-19]. Dostupné z: https://www.ackee.cz/blog/uvod-do-mvp-na-androidu/

[15] ViewModel Overview. Developers [online]. Mountain View: Google, 2018 [cit. 2018-03-19]. Dostupné z: https://developer.android.com/topic/libraries/architecture/viewmodel.html

[16] JIRAVA, Jarda. DataBinding a DataTemplate. In: Xaml.cz [online]. Xaml.cz, 2010 [cit. 2018-03-19]. Dostupné z: https://xaml.cz/wpf/databinding-a-datatemplate/

[17] Data Binding Overview. In: Microsoft [online]. Redmond: Microsoft, 2017 [cit. 2018-03-19]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview

[18] SALEH, Hazem. MVVM architecture, ViewModel and LiveData (Part 1). In: Pro Android Dev [online]. Medium, 2017 [cit. 2018-03-20]. Dostupné z: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1

[19] LiveData Overview. Developers [online]. Mountain View: Google, 2018 [cit. 2018-03-22]. Dostupné z: https://developer.android.com/topic/libraries/architecture/livedata.html

[20] Real-time database. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2018-03-22]. Dostupné z: https://en.wikipedia.org/wiki/Real-time_database

[21] Základní principy databází reálného času. Automa: časopis pro automatizační techniku [online]. Děčín: Automa, 2001 [cit. 2018-03-23]. Dostupné z:

http://automa.cz/cz/casopis-clanky/zakladni-principy-databazi-realneho-casu-2001_12_33758_2156/

[22] Relační databáze. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2018-03-23]. Dostupné z: https://cs.wikipedia.org/wiki/Rela%C4%8Dn%C3%AD_datab%C3%A1ze

[23] HOMAN, Jacqueline. Relational vs. non-relational databases: Which one is right for you? Pluralsight: The technology learning platform [online]. Farmington, Utah: Pluralsight, 2014, 6 April 2014 [cit. 2018-03-25]. Dostupné z: https://www.pluralsight.com/blog/software-development/relational-non-relational-databases

[24] Relačné (SQL) a nerelačné (NoSQL) modelovanie. Michal Macejko, Freelancer [online]. Macejko.sk, 2015 [cit. 2018-03-25]. Dostupné z: http://macejko.sk/articles/13-relacne-sql-a-nerelacne-nosql-modelovanie

[25] MALÝ, Martin. Web Sockets. Zdroják [online]. Praha: Devel.cz, 2009 [cit. 2018-04-01]. Dostupné z: https://www.zdrojak.cz/clanky/web-sockets/

[26] What are WebSockets?. Pusher [online]. London: Pusher, 2018 [cit. 2018-04-02]. Dostupné z: https://pusher.com/websockets

[27] Retrieving Data. Firebase [online]. Mountain View: Google, 2018 [cit. 2018-04-02]. Dostupné z: https://firebase.google.com/docs/database/admin/retrieve-data

[28] Build better apps, faster with real: An Overview of the Realm Platform. San Francisco (CA), 2017.

[29] Products. Firebase [online]. Mountain View: Google, 2018 [cit. 2018-04-04]. Dostupné z: https://firebase.google.com/

[30] ESPLIN, Chris. What is Firebase?. How To Firebase [online]. San Francisco (CA): Medium, 2016 [cit. 2018-04-04]. Dostupné z: https://howtofirebase.com/what-is-firebase-fcb8614ba442

[31] Products. Firebase [online]. Mountain View: Google, 2018 [cit. 2018-04-05]. Dostupné z: https://firebase.google.com/products/

[32] Cloud Firestore. Firebase [online]. Mountain View: Google, 2018 [cit. 2018-04-05]. Dostupné z: https://firebase.google.com/docs/firestore/

[33] ZUKANOV, Vasiliy. MVP and MVC Architectures in Android – part 1. TechYourChance [online]. 2016 [cit. 2018-04-25]. Dostupné z: https://www.techyourchance.com/mvp-mvc-android-1/

[34] T.EMMATTY, John. Differences between MVC and MVP for Beginners. CodeProject: For those who code [online]. Toronto: CodeProject, 2011 [cit. 2018-04-26]. Dostupné z: https://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners

[35] MVP vs MVVM: A Review of Architectural Patterns for Android. ThinkMobiles [online]. Uzhorod: ThinkMobiles, 2017 [cit. 2018-04-26]. Dostupné z: https://thinkmobiles.com/blog/mvp-vs-mvvm-android-patterns/

[36] ABRAMCZYK, Robert. Cloud Firestore vs Realm Platform. Netguru [online]. Poznań: Netguru, 2018 [cit. 2018-04-26]. Dostupné z: https://www.netguru.co/codestories/cloud-firestore-vs-realm-platform

[37] LACKO, Ľuboslav. Mistrovství - Android. Brno: Computer Press, 2017. Mistrovství. ISBN 978-80-251-4875-4.

[38] JACKSON, Wallace. Android apps for absolute beginners: covering Android 7. New York, NY: Springer Science Business Media, 2017. ISBN 978-1-4842-2267-6.

[39] PECINOVSKÝ, Rudolf. Návrhové vzory: [33 vzorových postupů pro objektové programování]. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[40] HILLS, Ted. NoSQL and SQL data modeling. Basking Ridge, NJ: Technics Publications, 2016. ISBN 978-163-4621-090.

[41] DEKA, Ganesh Chandra. NoSQL: database for storage and retrieval of data in cloud. Boca Raton, FL: Chapman and Hall/CRC, 2017. ISBN 978-149-8784-368.

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

2D	2-Dimensional - dvojrozmerný
3D	3-Dimensional - trojrozmerný
ACID	Atomicity, Consistency, Isolation, Durability – atomickosť, konzistencia, izolovanosť, trvácnosť
API	Application Programming Interface - rozhranie pre programovanie aplikácií
č.	číslo
DB	DataBase – databáza
DBMS	DataBase Management System - systém správy databáz
ES	Embedded Systems - vstavané systémy
HTML	HyperText Markup Language - Hypertextový značkový jazyk
ID	IDentification - identifikácia
MVC	Model View Controller
MVP	Model View Presenter
MVVM	Model View ViewModel
napr.	napríklad
OOP	Objektovo Orientované Programovanie
OpenGL	Open Graphics Library
OS	Operating System – operačný systém
pod.	podobné
REST	REpresentational State Transfer
RPC	Remote Procedure Call – vzdialené volanie procedúr
SDK	Software Development Kit - súbor nástrojov pre vývoj softvéru
SQL	Structured Query Language - štruktúrovaný dopytovací jazyk
tzv.	takzvaný
UI	User Interface – užívateľské rozhranie

VM	ViewModel
XML	eXtensible Markup Language - rozširiteľný značkovací jazyk

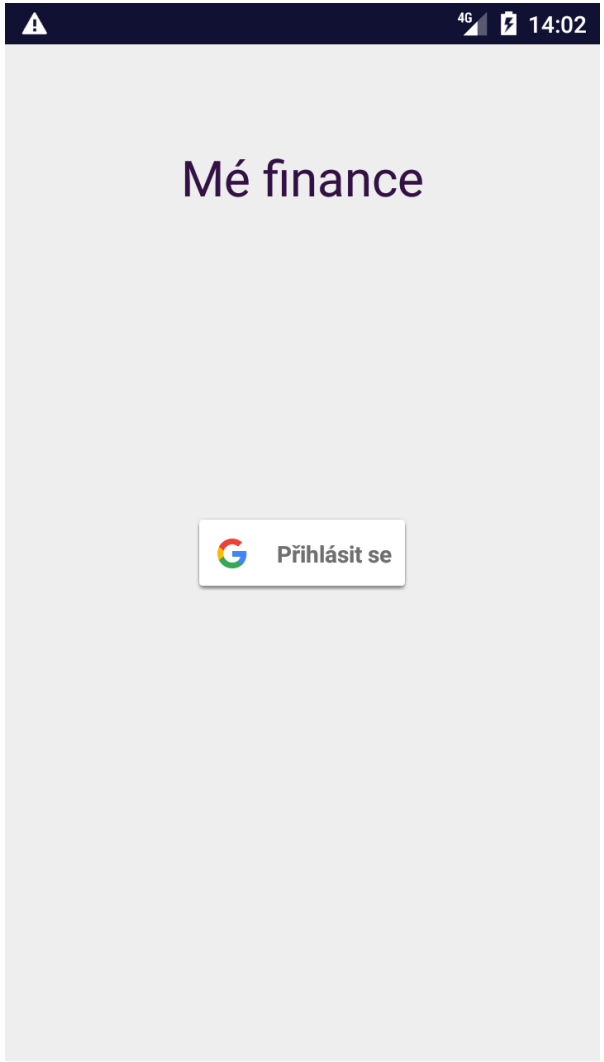
ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

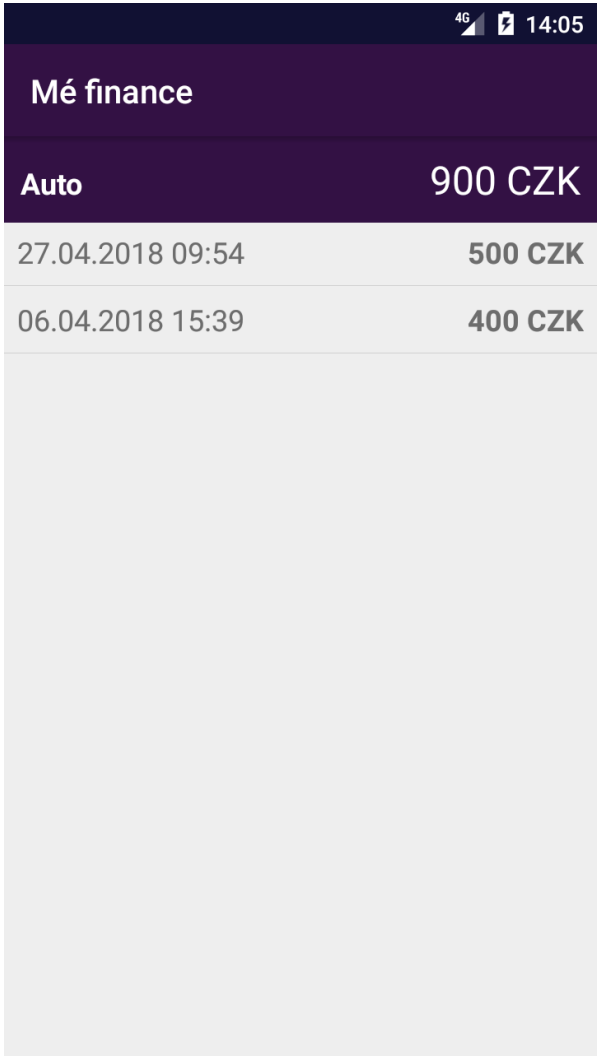
ZOZNAM ZDROJOVÝCH KÓDOV

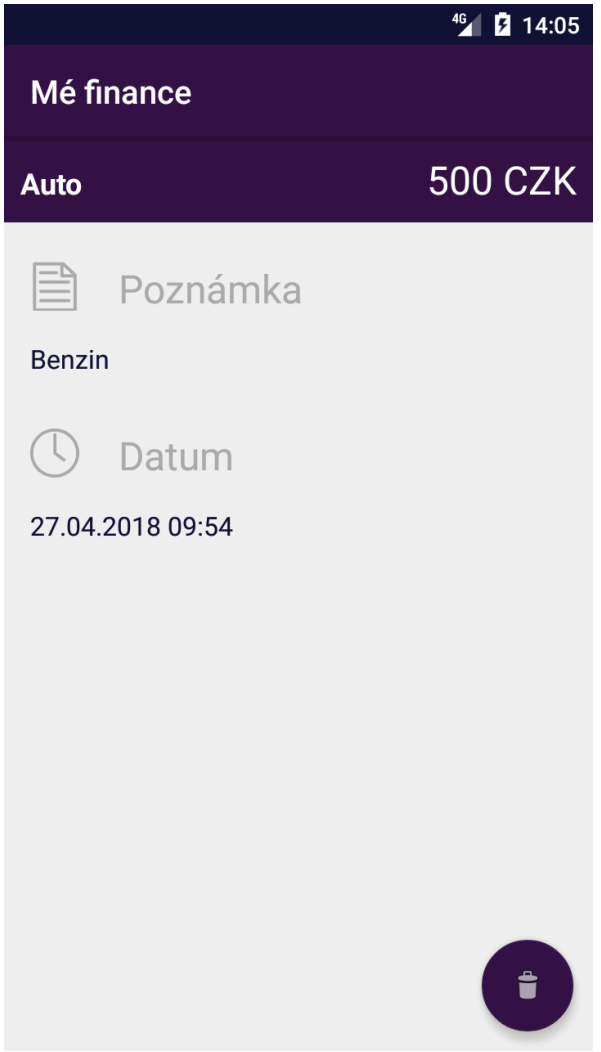
ZOZNAM PRÍLOH

PRÍLOHA 1: PRIHLASOVACIA OBRAZOVKA, HLAVNÁ OBRAZOVKA



PRÍLOHA 2: DETAIL KATEGÓRIE, DETAIL TRANSAKCIE





PRÍLOHA 3: PRIDANIE TRANSAKCIE, PROFIL