Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Mobilní aplikace pro správu investičního portfolia

2023

Jakub Pilch

Vedoucí práce:
RNDr. Arnošt Večerka

Studijní program: Informatika,
Specializace: Programování a vývoj software

Bibliografické údaje

Autor:	Jakub Pilch
Název práce:	Mobilní aplikace pro správu investičního portfolia
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Informatika, Specializace: Programování a vývoj software
Vedoucí práce:	RNDr. Arnošt Večerka
Počet stran:	41
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Jakub Pilch
Title:	Mobile application for investment portfolio management
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Computer Science, Specialization: Programming and Software Development
Supervisor:	RNDr. Arnošt Večerka
Page count:	41
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

V rámci této bakalářské práce byla navržena a vytvořena mobilní aplikace pro správu investičních portfolií, s důrazem na jednoduchost a technologickou implementaci. Na základě analýzy existujících řešení byla navržena architektura aplikace, která umožňuje uživatelům snadno a efektivně spravovat svá portfolia a sledovat výkonnost svých investic. Celkovým cílem práce bylo přinést užitek začínajícím investorům, kteří budou díky této aplikaci schopni lépe spravovat svá portfolia.

Synopsis

In this bachelor thesis, a mobile application for investment portfolio management was designed and developed with an emphasis on simplicity and technological implementation. Based on the analysis of existing solutions, the architecture of the application was designed to allow users to easily and efficiently manage their portfolios and monitor the performance of their investments. The overall objective of the work was to benefit investors who will be able to better manage their portfolios through this application.

Klíčová slova: Android, mobilní aplikace, investice, portfolio

Keywords: Android, mobile application, investment, portfolio

Rád bych poděkoval panu RNDr. Arnoštu Večerkovi za jeho cenné rady a odborné vedení, které mi výrazně pomohly při psaní této bakalářské práce. Jeho znalosti a ochota mi byly v průběhu celého procesu psaní velkou oporou a pomohly mi dovést práci k úspěšnému dokončení.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1 Úvod 7

2 Rozbor zadání 8
2.1 Existující řešení . . . 8
2.1.1 My Stocks Portfolio - MSP . . . 8
2.1.2 Stock Events . . . 8
2.2 Vývoj mobilních aplikací . . . 8
2.3 Případy užití . . . 10

3 Použité jazyky a technologie 11
3.1 Kotlin . . . 11
3.2 Firebase . . . 12
3.3 Android Studio . . . 12

4 Technologické řešení 13
4.1 Autentifikace uživatele . . . 13
4.2 Data . . . 13
4.2.1 Získávání finančních dat . . . 13
4.2.2 Formát dat . . . 13
4.2.3 Uchování dat . . . 14
4.2.4 Grafové zobrazení dat . . . 15

5 Vývoj 16
5.1 Návrh vzhledu . . . 16
5.1.1 Wireframe . . . 16
5.2 Návrh architektury . . . 16
5.2.1 MVVM architektura . . . 17
5.3 Datové modely . . . 18
5.4 Implementace grafů . . . 19
5.5 Struktura API . . . 20
5.5.1 Vyhledání akcie . . . 20
5.5.2 Získávání dat o akcii . . . 21
5.5.3 Historická data . . . 22
5.6 Vyhledání akcie . . . 22
5.7 Úložiště . . . 23
5.7.1 Implementace interního úložiště . . . 23
5.7.2 Firestore . . . 24
5.8 Přihlášení a registrace . . . 25
5.9 Aktualizace dat na pozadí . . . 26
5.9.1 Využití korutin . . . 27
5.10 Notifikace . . . 27

6 **Uživatelská příručka** 29
6.1 Přihlašovací obrazovka 29
6.2 Obrazovka Portfolio 29
6.2.1 Vývoj portfolia 30
6.2.2 Predikce 31
6.3 Obrazovka Aktiva 32
6.3.1 Okno s detailem akcie 32
6.3.2 Okno přidání akcie 33
6.3.3 Tvorba upozornění 35
6.4 Obrazovka Možnosti 35
6.4.1 Uživatelské možnosti 35
6.4.2 Další funkce 35
6.5 Jazyk aplikace 36

Závěr 38

Conclusions 39

A **Obsah elektronických dat** 40

Bibliografie 41

1 Úvod

Investování je v dnešní době stále větší trend a pro mnoho lidí atraktivní způsob, jak zhodnotit své úspory nebo peníze alespoň ochránit před inflací. Není jednoduché každý den sledovat vývoj na trzích a to i přes to, že existuje nespočet nástrojů a aplikací, které v tom pomáhají. Tyto aplikace jsou často velmi složité a pro člověka, který nemá znalost v technologiích, to může být odrazující. Aplikace, která má jen ty nejdůležitější funkce a je jednoduchá pro každého člověka na trhu není. V dnešní době, kdy se inflace pohybuje v procentech dvouciferných čísel, by měl mít každý alespoň základní investiční znalost.

Cílem práce je proto vytvořit mobilní aplikaci, ve které si uživatel může sestavit virtuální portfolia a sledovat jejich vývoj. Ty si uživatel tvoří pomocí reálných akcií, které lze vyhledat ve vyhledávacím řádku aplikace. Informace o akciích jsou stahovány z API AlphaVantage a jsou tedy vždy aktuální. Uživatel si dále může prohlížet vývoj a historii daného portfolia, predikci pro růst v budoucnosti, ale také si může vytvořit požadavek na upozornění při vývoji ceny dané akcie. Všechna data jsou reprezentována graficky tak, aby byla pro uživatele co nejlépe čitelná. Tyto funkce by měli uživateli pomoct pochopit základ investovaní a složeného úročení, který je důležitý před nákupem reálných akcií.

Abychom uživateli umožnili správu portfolií z více zařízení, je vhodné implementovat funkci pro vytvoření uživatelského účtu, na který navážeme jeho data. Tímto účtem se poté bude moct přihlásit z jakéhokoliv mobilního zařízení s operačním systémem Android. Tím docílíme nejen synchronizaci mezi zařízeními, ale také zálohu a ochranu dat proti smazání z interního úložiště zařízení.

Aplikace je napsána v jazyce Kotlin, a proto je určena pouze pro operační systém Android. Tento jazyk nabízí i multiplatformní řešení (Kotlin Multiplatform), nicméně se nejedná vůbec o optimální volbu. Pro tento účel bych volil spíše Flutter, nebo v úplně ideálním případě nativní vývoj pro každou platformu zvlášť.

Uživatelské rozhraní by mělo působit moderně, jednoduše a intuitivně. Aplikace by v ideálním případě měla jít používat i bez předchozího zaškolení a měla by uživateli dávat dostatek informací o dění v aplikaci. Proto je vhodné dodržovat již ověřená pravidla pro vývoj uživatelského rozhraní.

2 Rozbor zadání

2.1 Existující řešení

Aplikací s tématikou investování a financí existuje nespočet. Nepodařilo se mi však najít takovou aplikaci, kterou bych mohl doporučit i úplnému začátečníkovi. Většina aplikací je buď nepřehledná nebo zbytečně složitá. Vybral jsem na ukázku tyto dvě aplikace, protože mají něco navíc oproti aplikacím ostatním.

2.1.1 My Stocks Portfolio - MSP

Prvním podobným řešením je aplikace *MSP*, která je vyobrazena na obrázku 1. Tato aplikace nabízí celou řadu funkcí a nástrojů, čímž se může zařadit do kategorie aplikací pro pokročilé uživatele. Nároční uživatelé jsou jistě nadšení z možnosti používat tolik funkcí, nicméně pokud je uživatel začátečník, tak tato aplikace jistě není pro něj. Aplikace disponuje nepřehledným uživatelským rozhraním a i po delším používání jsem často nevěděl, kam kliknout pro určitou funkcionalitu.

Na druhou stranu lze ocenit množství informací, které aplikace poskytuje. Jednotlivé akcie i portfolia jsou detailně popsány celou řadou statistik a dat, což je ale někdy na úkor přehlednosti.[1] Dále bych ocenil záložku s novinkami ze světa financí, kterou opravdoví nadšenci finančního světa jistě ocení.

2.1.2 Stock Events

Aplikace *Stock Events* oplývá o něco hezčím a modernějším uživatelským rozhraním (viz obrázek 2), zejména lze ocenit velké množství grafových reprezentací. Také nám umožňuje sledovat akcie a přidávat si je do pomyslného portfolia, nicméně nelze tvořit portfolií více. Dále je poskytnuto hodně informací o sledovaném portfoliu, včetně dividend a předpovědi do budoucna.

Hezkou přidanou funkcí je záložka s kalendářem, ve kterém jsou uložené důležité události týkající se finančního světa. Tato funkce přehledně shrnuje dění v investicích za celý rok, a to bez zbytečných článků a dlouhého čtení. [2]

2.2 Vývoj mobilních aplikací

Vývoj mobilních aplikací se dá rozdělit na dva základní způsoby – nativní vývoj a hybridní vývoj. Oba způsoby mají své pozitivní i negativní stránky a je na každém, co od aplikace očekává. Nelze ale říct, že výběr závisí jen na výsledném produktu vývoje. Důležité je zvážení i ostatních aspektů, jako údržba aplikace nebo třeba náklady na vývoj.

Nativní aplikace je aplikace vyvinutá pro konkrétní platformu (Android nebo iOS). Pro takovýto vývoj se používají jazyky k tomu určené. Pro vývoj na plat-

[1] Ke stažení zde: https://play.google.com/store/apps/details?id=co.peeksoft.stocks&hl=cs
[2] Ke stažení zde: https://play.google.com/store/apps/details?id=app.stockevents.android&hl=cs

Obrázek 1: Aplikace MSP

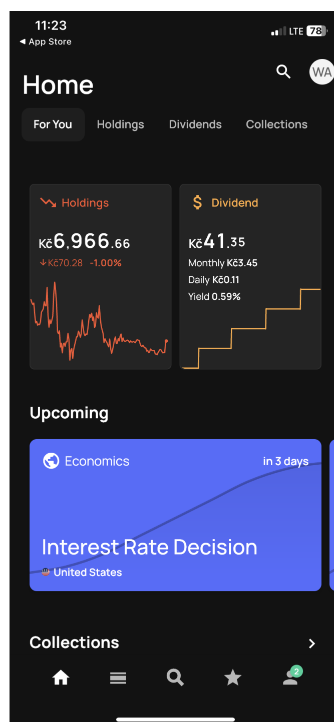
formu Android se používá například Java a Kotlin, pro iOS aplikace se pak používá jazyk Swift, potažmo jazyk Objective-C. Vývojáři také používají speciálně vyvinuté nástroje jako Xcode pro vývoj pro zařízení značky Apple, nebo Android studio pro vývoj aplikací pro zařízení s operačním systémem Android.

Jednoznačně největší výhodou nativního vývoje je fakt, že nemusíme dělat kompromisy mezi zařízeními. Při vývoji můžeme používat přímo systémové služby operačního systému, a tím také přistupovat přímo k hardwaru. Díky tomu je běh aplikace mnohem rychlejší a odezva plynulejší. Aplikaci také vyvíjíme pro výrazně menší počet zařízení, a proto je uživatelské rozhraní lépe optimalizované. Pro takovýto vývoj je ovšem potřeba programátor se znalostí konkrétního jazyka. V případě, že chceme vyvíjet aplikace pro obě platformy nativně, pak nás proces vyjde mnohem dráž. Při tomto scénáři je potřeba vytvořit tým pro každou platformu a tím se mnohonásobně zvedne cena.

Hybridní vývoj aplikací je způsob, kdy píšeme jeden kód pro obě platformy zároveň. Pro tento způsob se využívají frameworky[3] jako React-native, Flutter nebo Xamarin.

Výhodou hybridního vývoje je množství kódu potřebného pro sestavení aplikace pro obě platformy. Z důvodu psaní pouze jednoho kódu je potřeba mnohem méně vývojářů, a tedy i výsledná cena projektu je výrazně nižší než u vývoje

[3] Framework je soubor předpřipravených knihoven a nástrojů pro zjednodušení vývoje software.

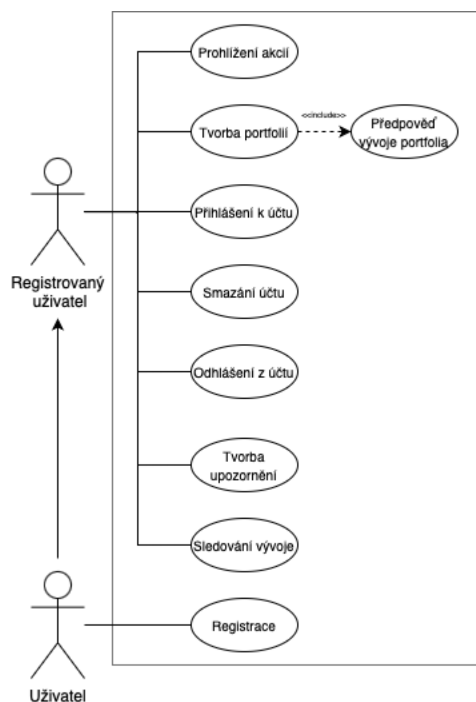


Obrázek 2: Aplikace Stock Events

nativního. Další výhodou je následná údržba a testování kódu, které je díky množství kódu mnohem jednodušší. Nevýhodou pak může být rychlost aplikace a přístup k hardwaru, který se zpracovává pomocí různých knihoven.

2.3 Případy užití

Použití aplikace z uživatelského hlediska je možné vyobrazit diagramem případů užití. Ten lze vidět na obrázku č. 3. Jedná se o základní diagram při modelování systémů a procesů. Popisuje nám, jak se systém používá a jaké funkce nabízí. Skládá se z tzv. aktérů (lidí nebo systémů, kteří s systémem komunikují) a případů užití (scénáře, které systém nabízí). Aktéři jsou znázorněni postavičkou a případy užití elipsou.



Obrázek 3: Diagram užití

3 Použité jazyky a technologie

3.1 Kotlin

Kotlin je moderní, staticky typovaný programovací jazyk, který byl vyvinut společností JetBrains v roce 2011. Jazyk je navržen tak, aby byl zpětně kompatibilní s již existujícími Java aplikacemi, a zároveň nabízel nové a moderní vylepšení. Mezi ně patří například podpora funkcionálního programování nebo jednodušší a bezpečnější práci s vlákny.

Jazyk Kotlin je přizpůsoben tomu, aby byl jednoduchý na učení a používání. Syntaxe jazyka je čitelná a jednoduchá, což zvyšuje rychlost psaní kódu. Kotlin podporuje funkcionální programování, lambda výrazy, rozšíření funkcí, korutiny, ochranu před null a další užitečné funkce.

Kotlin se stává stále více preferovaným jazykem pro vývoj mobilních aplikací pro Android, nahrazující tak tradičnější Javu. Pro vývojáře je díky své jednoduchosti a čitelnosti jasnou volbou. Dále tento jazyk nabízí podporu pro návrhové vzory jako je MVC, MVP nebo MVVM. Tento jazyk lze využít i pro vývoj desktopových, webových nebo třeba serverových aplikací. Jedná se o multiplatformní jazyk, a tím se jeho univerzalita prohlubuje. Kotlin je podporován spoustou IDE, jako například Android Studio, IntelliJ či Eclipse. Dále umožňuje integraci nástrojů, jako je Git či Maven.

Celkově lze říci, že Kotlin se stává čím dál víc populární mezi programovacími

jazyky. Díky jeho vlastnostem a univerzálnosti si jej vývojáři oblíbili. Stále je jeho největší využití ve vývoji mobilních aplikací, ale postupně proniká i do jiných odvětví vývoje. [**Kotlin**]

3.2 Firebase

Firebase je cloudová platforma vytvořená společností Google. V její nabídce nalezneme spoustu služeb a nástrojů, které lze využít při vývoji mobilních aplikací, webových aplikací či IoT projektů. Zásadní služby, které poskytuje, je autentifikace uživatelů a uchování dat.

Mezi její konkrétní nástroje patří Realtime database, Cloud firestore, Firebase Authentication, Firebase Cloud Messaging pro odesílání notifikací nebo Firebase Hosting pro hostování webových aplikací. Dále poskytuje nástroje pro analýzu uživatelského chování, testování aplikací a další.

Výhodou Firebase je, že poskystuje již hotové řešení pro mnoho aspektů vývoje aplikací. Díky tomu se vývojář může soustředit na samotný vývoj aplikace, namísto konfigurování vlastního backendu. Díky bezplatnému tarifu, který Google nabízí, lze aplikace jednoduše vyvíjet a testovat. Další tarify se liší podle toho, co od nich vývojář či firma očekává. [**Firebase**]

3.3 Android Studio

Android studio je integrované vývojové prostředí (IDE) vyvinuté společností Google. Díky funkcím a nástrojům, které toto prostředí obsahuje, se jedná o nejrozšířenější a nejoblíbenější prostředí pro vývoj aplikací pro platformu Android. Mezi důležité nástroje pro vývoj patří:

- Editor kódu – možnost psaní a editace kódu v Javě, Kotlinu ale i C++.
- Layout editor – možnost vizuálně tvořit a upravovat uživatelské rozhraní aplikace.
- Gradle-based build system – správa kompilací, sestavování a balení souborů do finálního souboru.
- Emulátor – virtuální zařízení, na kterém vývojář může testovat aplikace bez potřeby vlastnit fyzické zařízení.

Mezi další užitečné funkce patří také automatické formátování kódu, profiler pro sledování výkonu aplikace nebo zabudovaná podpora pro verzování kódu pomocí služby Git či Subversion. Všechny tyto funkce pomáhají vytvářet robustní a vyladěné aplikace pro Android.

4 Technologické řešení

4.1 Autentifikace uživatele

Firebase nám poskytuje službu autentifikace, díky které si uživatel může vytvořit vlastní účet pomocí e-mailu, Google účtu, Facebooku či jiných služeb. Tato funkce je důležitá pro zachování a přenos dat mezi různými zařízeními. Při případném vymazání paměti v mobilním zařízení jsou data zálohovaná na Cloudu, a tedy o ně uživatel nepřijde.

Pro vytvoření účtu lze použít spoustu variant, nicméně já jsem do aplikace implementoval pouze registraci pomocí e-mailové adresy. Abychom eliminovali registraci uživatele na cizí adresu, využiji funkci, která po zaregistrování odešle odkaz pro ověření na daný e-mail. Obsah potvrzovacího e-mailu si lze nakonfigurovat ve webové konzoli Firebase. Uživatel je vpuštěn do aplikace až po ověření, což docílí kliknutím na odkaz, který mu přišel do schránky. Možnosti registrace lze v budoucnu jednoduše rozšířit o další způsoby.

4.2 Data

4.2.1 Získávání finančních dat

Na internetu lze najít spoustu zdrojů odkud data čerpat, nicméně data společnosti Alpha Vantage [**AV**] a jejich API mi poskytli přesně to, co potřebuji. Jedinou limitací, která se naskytla bylo, že verze pro osobní použití poskytovala pouze 5 požadavků/min (500 požadavků/den). Bylo proto potřeba se obrátit na podporu a získat verzi pro studijní účely, která mi poskytuje 75 požadavků/min.

Může se zdát, že počet poskytnutých požadavků je veliký. Pro osobní účely testování tomu tak je, ale pokud by se aplikace měla dostat mezi lidi, je třeba koupit vylepšenou verzi. Když vezmeme v potaz, že vyhledáváme akcii podle jména, a s každým znakem se nám posílá nový požadavek, pak tento počet velmi rychle vyčerpáme.

Toto API poskytuje spoustu funkcí od základních informací, přes historická data až po data pro našeptávač při vyhledávání aktiv. Navíc lze získávat data o akciích, kryptoměnách, Forexu[4] ale i komoditách jako cukr, kafe či obilí. Lze vidět, že dat je nabízeno nespočet, nicméně pro naše účely budou stačit základní funkce. V případě budoucího rozšíření aplikace je rozsah dat velkou výhodou a věřím, že spoustu z nich v budoucnu využiji.

4.2.2 Formát dat

Návratová data služby jsou ve formátu, jež je určen v požadavku na API. Ten je určen parametrem, do kterého lze dosadit hodnoty jako je *json* nebo *csv*. V případě formátu *json* se jedná o typ klíč-hodnota. Naopak *csv* data repre-

[4]Forex je zkratka pro Foreign Exchange, což znamená trh s cizími měnami.

zentuje pomocí oddělovačů (většinou čárky). Každý řádek je řádkem pomyslné tabulky a čárky oddělují sloupce.

Samotná aplikace bere vstupní hodnoty ve formátu *json*. Díky knihovnám pro serializaci je následně velmi jednoduché z dat vytvořit objekty, se kterými lze poté jednoduše pracovat. Příklad dat ve formátu *json* lze vidět v kódu č. 1.

```
1  {
2     "Global Quote": {
3        "01. symbol": "AAPL",
4        "02. open": "165.0900",
5        "03. high": "165.3900",
6        "04. low": "164.0300",
7        "05. price": "165.2300",
8        "06. volume": "40713618",
9        "07. latest trading day": "2023-04-17",
10       "08. previous close": "165.2100",
11       "09. change": "0.0200",
12       "10. change percent": "0.0121%"
13    }
14 }
```

Zdrojový kód 1: Příklad dat ve formátu json

4.2.3 Uchování dat

V této části se změříme na dva problémy: uchování dat v interní paměti a uchování dat na vzdáleném úložišti.

Uložit data v telefonu je naštěstí díky knihovnám Androidu docela jednoduchá záležitost. V mém případě jsem použil knihovnu Room, která k práci s daty používá jazyk SQLite. Data jsou uložena v paměti a my s nimi pomocí SQL queries můžeme pracovat.

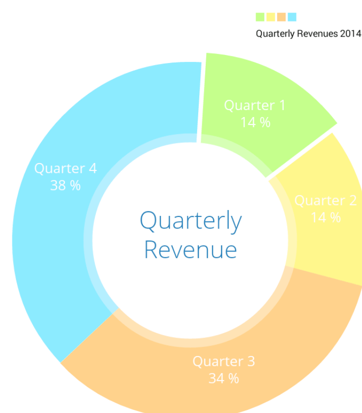
Volba vzdáleného úložiště byla o něco složitější. Ze začátku jsem chtěl data zálohovat na Google Drive, ale ani zdaleka se to nejevilo jako optimální volba. Tvorba vlastního úložistě mi pro tyto účely přišla zbytečná. Nakonec jsem tedy zvolil Firestore, který je pro tento scénář ideální volbou.

Ve Firebase konzoli si nejprve musíme založit projekt. Následně si vybereme typ databáze, kterou chceme použít. Na výběr máme ze dvou možností: Cloud Firestore a Realtime database. Výběr databáze se odvíjí od toho, co od ní očekáváme a na jaký účel bude použita. Cloud Firestore používá k organizaci dat dokumenty a kolekce a je stavěna pro větší množství dat. Naopak Realtime databáze používá stromovou strukturu spolu s dvojicema klíč-hodnota a je stavěna pro menší data. S tím se také pojí možnost dotazů – první zmíněná databáze nám dává širší možnost dotazování, než databáze druhá. Poslední důležitou vlastností je synchronizace dat. Realtime databáze umožňuje synchronizaci dat v reálném čase, což se hodí zejména u aplikací, kde potřebujeme změnu dat vidět ihned.

Já jsem si pro svoji aplikaci vybral Cloud Firestore, ačkoliv neočekávám, že uživatel bude ukládat velké obnosy dat. Pro tento výběr jsem se rozhodnul z důvodu struktury databáze. Každý uživatel má svoje ID, díky kterému lze jednoduše identifikovat dokumenty v kolekcích, které mu patří.

4.2.4 Grafové zobrazení dat

Je zvykem, že aplikace pracující s financemi zobrazují většinu dat pomocí grafů. Takto vhodně zvolená reprezentace je důležitá pro uživatele, aby se rychle zorientoval v datech a nemusel nad tím trávit spoustu času. Lineární, koláčové, sloupcové a jiné grafy jsou nedílnou součástí, a proto jsem použil knihovnu *MPAndroidChart*, která poskytuje hezké a jednoduše použitelné grafy. Ukázku koláčového grafu lze vidět na obrázku 4. Nejedná se o originální knihovnu Androidu, nicméně Android nativní knihovnu pro grafy neposkytuje. Tato knihovna je doporučená používat a je momentálně asi nejlepší volbou.



Obrázek 4: Příklad koláčového grafu [**PhilJay**]

5 Vývoj

5.1 Návrh vzhledu

Podoba uživatelského rozhraní (UI) je velmi důležitým prvkem celé aplikace. Může to uživatele nalákat na časté využívání a nebo ho úplně odradit. Proto je důležité tomuto aspektu každého projektu věnovat zvláštní pozornost.

Navrhování UI se může jevit jako triviální záležitost, realita je ale opačná. UI design je samostatné odvětví, které má přesah až do psychologie. Hlavní motivací je intuitivní a jednoduché ovládání tak, aby v ideálním případě uživatel nepotřeboval žádný manuál a uměl si s aplikací poradit sám.

5.1.1 Wireframe

Aby programátor nevymýšlel UI za pochodu, je vhodné si před vývojem vytvořit návrh aplikace (Wireframe). Jedná se o jednoduchý nákres aplikace, ve kterém je zobrazené rozložení obrazovky a veškerých komponent. Pro návrh lze využít mnoha grafických nástrojů jako je Figma, ale lze použít i tužku a papír. Já osobně jsem využil druhou možnost, akorát v digitální verzi.

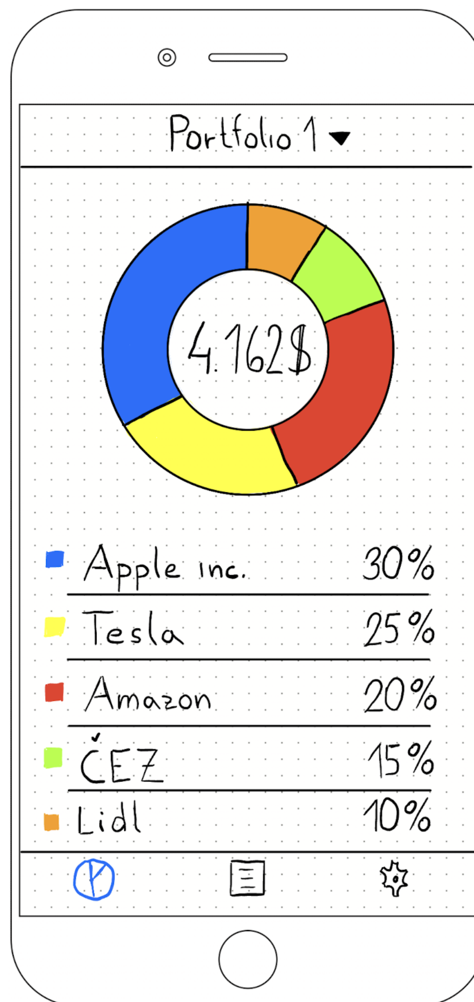
Abychom byli v grafických nástrojích jako Figma schopni vytvořit hezký wireframe, je třeba mít v programech už nějakou zkušenost. Jelikož se zameřuji spíš na technickou stránku, nepřišlo mi důležité trávit čas zkoumáním grafických nástrojů. Kreslení od ruky také není jednoduchá záležitost, ale v případě, že máte možnost kreslit digitálně, kde vás software asistuje a dělá vám rovné čáry, pak to není tak složité. Z internetu si lze stáhnout šablony, které vyobrazují zařízení a do kterého se pak dokreslují jednotlivé komponenty. Lze si také zapnout mřížku, díky které je jednodušší se orientovat a dělat symetrické rozhraní. Vzniklý návrh lze vidět na obrázku č. 5.

Není dobré se upínat pouze na jeden návrh. Pro spokojenost s rozhraním je vhodné si ke každé obrazovce nakreslit klidně dvě až tři rozložení a následně z nich vybrat to nejlepší. Tohle nám pomůže, abychom nenavazovali s dalšími obrazovkami stále na tu jednu původní. Samozřejmě být kreativní je skvělá věc, ale je důležité se držet ověřených postupů při návrhu UI.

Finální podoba mé aplikace pramení z návrhu, který jsem vytvořil. Nicméně v průběhu vývoje se naskytly nové nápady a funkcionality, které v návrhu nejsou vyobrazeny. I přes to si ale myslím, že je jasně vidět, že aplikace vychází z návrhu vyhotoveném před začátkem vývoje.

5.2 Návrh architektury

Architektura je další stěžejní částí každého softwaru. Softwarové inženýrství nám poskytuje již navrhnuté architektury, které lze implementovat v našem projektu. Některé z nich jsou vhodnější pro mobilní vývoj, jiné například pro web. Jejich úkolem je definovat vztah mezi jednotlivými komponenty a rozdělit zodpovědnost pro každou z nich. Kdyby každá komponenta mohla dělat všechno, vedlo by to



Obrázek 5: Wireframe obrazovky portfolio

k chaosu v kódu. Dalším důsledkem by také mohl být pomalý výsledný software, náročná testovatelnost a velmi složitá údržba kódu. Pro svou aplikaci jsem vybral *MVVM* model, který se skvěle hodí pro vývoj mobilních aplikací.

5.2.1 MVVM architektura

Model-View-ViewModel je softwarová architektura, která se používá pro vývoj desktopových i mobilních aplikací. Tato architektura odděluje logiku a uživatelské rozhraní, čímž usnadňuje testování a údržbu aplikace. Obsahuje 3 komponenty:

- Model
- View
- View-model

Model je komponenta, která má na starost logiku aplikace, zpracování dat a získávání dat – např. získávání dat z databáze nebo požadavky na API. Následně tato data poskytuje *viewmodelu*, který s nima dále pracuje.

Viewmodel je prostředníkem mezi *modelem* a *view*. Jeho úlohou je poskytnout logiku pro zobrazení a reprezentaci dat. *Viewmodel* komunikuje s *view* pomocí data bindingu, tzn. propojuje data ze zdroje s UI.

View zobrazuje data a obsluhuje změny, případně vstupy od uživatele. [**MVVM**]

5.3 Datové modely

Pro jednoduchou práci s daty a aplikací je důležité si navrhnout vhodné modely. Díky těmto modelům lze později jednoduše pracovat s databázemi a API. Modelem dat obecně myslíme třídu, která má vlastnosti a metody. Modely pro tuto aplikaci jsou:

- **Asset** je základní model pro práci s akciemi. Jako vlastnosti obsahuje všechny důležité informace ohledně ceny, symbolu, obchodních dnů a další. Celá třída je serializovatelná, takže je každá vlastnost modelu anotována pomocí @SerializedName. Abychom data tohoto modelu mohli uložit do databáze, je potřeba třídu anotovat pomocí @Entity a určit primární klíč. Ten musí být jedinečný, a jelikož data akcií budeme aktualizovat a nikoliv ukládat znovu, nabízí se označit symbol akcie jako primární klíč.

- **Portfolio** je složitým modelem, který obsahuje nejen vlastnosti, ale je rozšířen i o metody. Vlastnosti tohoto modelu jsou pouze tři – jméno portfolia, mapa akcií, které obsahuje, a počet do něj investovaných peněz. Komplikovanou vlastností je mapa akcií, protože aby mohla být třída serializovatelná, musí být i každá vlastnost serializovatelná. Pokud to datový typ vlastnosti nesplňuje, je potřeba pro něj naprogramovat konverter. Model dále obsahuje metody pro operace s akciemi a hodnotou portfolia. Pro operace s databází je opět potřeba anotovat a vybrat primární klíč, kterým je název portfolia.

- **Transaction** model zastupuje všechny transakce, které provádíme s akciemi. Třída obsahuje všechna data, která jsou o transakci potřeba znát, jako například datum, typ transakce, počet akcií, název portfolia nebo symbol akcie. Typ transakce nám udává, jestli se jednalo o nákup nebo prodej. Transakce nám sama o sobě nedává žádnou možnost unikátního identifikátoru, proto jsem využil anotace @PrimaryKey a parametru *autogenerate*, který za nás obstará automatickou generaci primárního klíče ve tvaru přirozeného čísla.

- **DailyDbEntry** je modelem reprezentujícím denní data, která se propisují do grafu vývoje portfolia. Abychom byli schopni sledovat vývoj, potřebujeme zapisovat data o jménu portfolia, aktuálním datu, investované hodnotě a hodnotě portfolia v daný den a rozdíl mezi nimi převedený na procenta.

Stejně jako u transakcí je potřeba generovat ID pro identifikaci denního zápisu.

- **PriceAlert** reprezentuje všechny uživatelské upozornění. Jeho vlastnostmi jsou symbol akcie, typ upozornění, cena akcie a unikátní ID, které je opět automaticky generované. Typem upozornění se myslí, jestli jsme upozorněni, když aktuální cena klesne pod určenou cenu, nebo překročí určenou cenu směrem nahoru. Pro tento účel jsem vytvořil výčtový typ, který obsahuje oba dva typy, takže si uživatel může vybrat.

Objekty každého z těchto tříd lze uložit do databáze, proto musí být všechny třídy řádně anotovány a obsahovat primární klíč. Příklad takové třídy lze vidět v ukázce kódu 2.

```
1  @Entity(tableName = AppConstants.ALERTS_TABLE)
2  data class PriceAlert(
3
4     @PrimaryKey(autoGenerate = true)
5     @ColumnInfo(name="id")
6     @SerializedName("id")
7     val id: Int,
8
9     @ColumnInfo(name = "symbol")
10    @SerializedName("symbol")
11    val symbol: String,
12
13    @ColumnInfo(name = "price")
14    @SerializedName("price")
15    val price: Float,
16
17    @ColumnInfo(name = "alert_type")
18    @SerializedName("alert_type")
19    val alertType: AlertType)
```

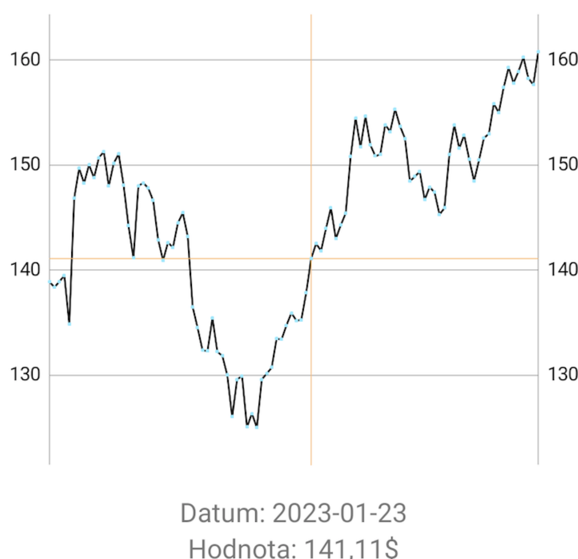
Zdrojový kód 2: Příklad modelu

5.4 Implementace grafů

Prvním krokem k implementaci je vhodné zasazení komponenty do uživatelského rozhraní. Do xml souboru příslušné obrazovky vložíme graf vhodným tagem[5], tj. *<PieChart/>*, nebo třeba *<LineChart/>*. Volba tagu grafu závisí na datech, které chceme reprezentovat. Lineární graf se hodí více pro data závislá na čase, koláčová reprezentace je vhodná pro data, které představují části jednoho celku. Pro identifikaci komponenty v kódu je potřeba jí přidělit unikátní identifikátor.

[5] Tag, neboli značka, je základním prvkem struktury jazyka XML.

Na tuto komponentu poté v příslušném view navážeme pomocí bindingu, kde jí najdeme pomocí identifikátoru. Aby měl graf vůbec co zobrazit, je potřeba mu přidělit vhodná data. Tato data musí být ve správné formě, které nám určuje samotná knihovna, ze které pochází i graf. Grafu nepřidělujeme surová data, ale vložíme je do datasetu, což je kolekce dat spolu s dalšími vlastnostmi, které jdou pro korektní zobrazení dat nastavit. Další vlastnosti pak lze nastavit u samotného grafu, které jsou ale spíš technického charakteru. Jedná se například o osy grafu, legendy nebo zpětné volání po kliknutí na graf. Příklad lineárního grafu lze vidět na obrázku č. 6



Obrázek 6: Lineární graf

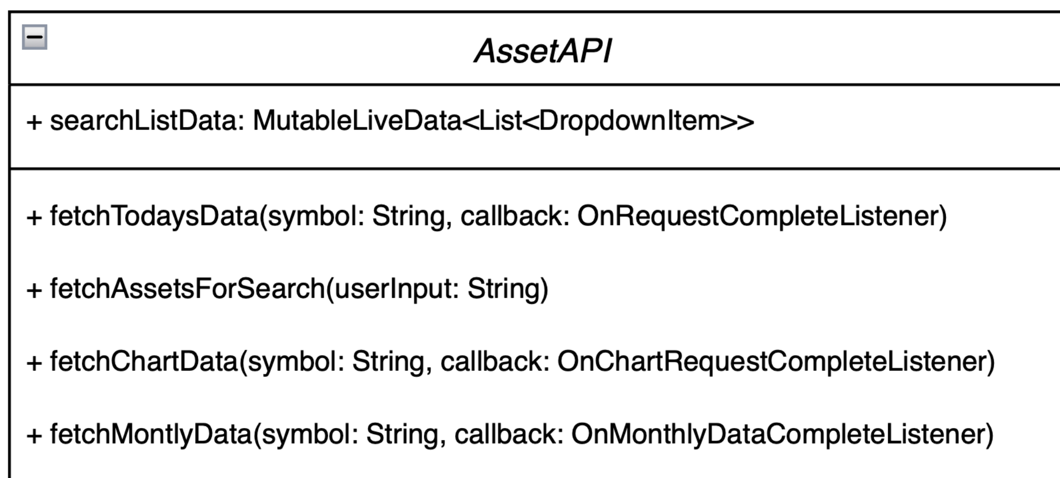
5.5 Struktura API

Pro získávání potřebných dat je vhodné si naprogramovat objekt, který bude obsluhovat všechny naše požadavky a přeposílat dále ke zdroji našich dat. Pro tuto funkcionalitu jsem vytvořil objekt AssetAPI, který obsahuje metody pro zisk dat, a je rozšířen o rozhraní, které při implementaci informuje volajícího o dokončení požadavku a vrácení dat. V opačném případě o neúspěchu. Strukturu třídy lze vidět na obrázku č. 7.

5.5.1 Vyhledání akcie

Alpha Vantage API nám poskytuje vyhledávání akcie pomocí symbolu[6] (např. AAPL) a pomocí plného názvu (např. Apple Inc.) zároveň. Díky této funkci ne-

[6]Symbol je posloupnost písmen, které označují danou akcii nebo cenný papír pro účel obchodování.



Obrázek 7: Diagram třídy AssetAPI

musíme rozlišovat, podle čeho uživatel vyhledává. Vstup uživatele lze jednoduše vložit do požadavku na API a ten nám vrátí korespondující výsledky.

Při větším počtu uživatelů by bylo vhodné implementovat omezení pro vstup z důvodu vysokého počtu požadavků (např. požadavek se pošle, až když jsou na vstupu alespoň 3 znaky). Jelikož mám k dispozici 75 požadavků/min, tak to pro účely testování a vývoje není potřeba. V případě, že by ale hledalo 10 uživatelů najednou, mohlo by dojít k rychlému vyčerpání tohoto limitu a následnému výpadku služby.

Po odpovědi se aktualizují data ve struktuře LiveData a spustí se akce všech pozorovatelů (observerů), které jsou na tuto strukturu navázány. Implementaci lze vidět ve zdrojovém kódu č. 3.

5.5.2 Získávání dat o akcii

Používané API nám poskytuje spoustu možností dat, které můžeme získat. Pokud požadujeme aktuální data, je vhodné použít funkci *Quote Endpoint*, která nám vrátí data jako: cena v daný moment, cena při uzavření trhu, cena při otevření trhu a jiné.

Povinnou složkou tohoto požadavku je symbol akcie, jehož informace chceme získat. O tento krok se uživatel nemusí starat, protože při kliknutí na akcii při vyhledávání se symbol uloží a následně použije při volání požadavku.

Dále si můžeme určit, v jakém formátu se nám data vrátí. Na výběr máme JSON a CSV formát. Nejen kvůli jednoduššímu zpracování, ale i kvůli podpoře knihoven Kotlinu jsem vybral formát JSON, který díky knihovně Kotlinu podporující serializaci lze jednoduše převést na objekt. Abychom mohli převést JSON na objekt, je nutné anotovat třídu objektu pomocí @Seralizable. V případě, že třída obsahuje atributy netriválních typů, jako například *map<key, value>*, je potřeba vytvořit konvertor, který určuje, jak data převádět.

```
1    fun fetchAssetsForSearch(userInput: String) {
2       val client = OkHttpClient()
3       val request = Request //Vytvoření požadavku
4          .Builder()
5          .url("${AppConstants.API_URL}/query?function=SYMBOL_SEARCH&
              keywords=${userInput}&apikey=${AppConstants.API_KEY}")
6          .build()
7
8    //Volání požadavku
9       client.newCall(request).enqueue(object : Callback {
10         override fun onFailure(call: Call, e: IOException) { ... }
11
12         override fun onResponse(call: Call, response: Response) {
13           ...
14             searchListData.postValue(dataList) // Aktualizace dat
15         }
16       }
17   }
```

Zdrojový kód 3: Dotaz na data pro vyhledávání

5.5.3 Historická data

Detailní informace o akcii obsahují i graf, který nám zobrazuje vývoj akcie za poslední půlrok. Tato data získáme pomocí funkce *Time series daily*, kterou spolu se symbolem akcie dosadíme do požadavku. Pokud získáváme data pro předpověď vývoje, potřebujeme data mnohem rozsáhlejší. Funkce *Time series monthly* nám poskytuje měsíční data s historií 20+ let. To je dostačující období na výpočet průměrného růstu.

Dalším parametrem, který si můžeme zvolit, je jak častá data požadujeme – tzn. jestli chceme záznamy na konci každého měsíce, nebo třeba týdne. Od toho se poté odvíjí přesnost predikce. Čím více dat bychom měli, tím lepší predikci bychom mohli udělat. Moje aplikace ale není matematicky zaměřená a odhad růstu je pouze orientační. Větší zpracování dat by vzalo více zdrojů zařízení a výsledný rozdíl by byl pravděpodobně nepatrný.

5.6 Vyhledání akcie

Implementaci vyhledávacího pole nám nabízí již samotné knihovny Kotlinu, které však po grafické stránce nanabízejí to, co potřebuji. Proto bylo potřeba si vytvořit vlastní xml soubor pro položku v rozbalovacím menu, a k tomu následně i adaptér.

Základní rozložení položky obsahuje pouze textové pole, což není dostatečné. Jak lze vidět na obrázku č. 8, moje implementace obsahuje textové pole pro symbol a druhé textové pole pro celý název aktiva. Abychom mohli vlastní rozložení použít, potřebujeme vytvořit vlastní adaptér. Tvorba adaptéru nemusí být

takový problém díky principům objektově orientovaného programování. Jednoduše lze zdědit implementaci třídy ArrayAdapter, u které stačí přepsat metodu getView tak, aby napasovala správná data na správné místo.

Abych vyhledávacímu řádku dal finální podobu, použil jsem knihovnu Material Designu, která obsahuje prvky moderního UI. Výsledná komponenta se skládá ze dvou částí – TextInputLayout a MaterialAutoCompleteTextView. První zmíněná nám dává možnost dosadit do řádku koncové a počáteční ikony (např. křížek na konci pro vymazání vstupu) a celkově vyladit grafickou podobu. Druhá položka má úlohu spíše funkční.

Obrázek 8: Vyhledávací řádek

5.7 Úložiště

5.7.1 Implementace interního úložiště

Před implementací databáze je dobré mít představu, jaká data budeme uchovávat. Tato data reprezentujeme takzvanými entitami, tj. třída, jejíž atributy budou reprezentovat sloupce ve výsledné tabulce databáze. Každý atribut můžeme označit různými anotacemi pro upřesnění (např. @ColumnInfo pro upřesnění atributu jako sloupce v tabulce nebo @PrimaryKey pro nastavení jako primární klíč). Samotná třída musí být anotována pomocí @Entity, kde do závorky definujeme název tabulky v databázi. Na obrázku č. 9 můžeme vidět data uchovaná v interní databázi a vztah mezi tabulkami.

Tvorba samotné databáze je díky knihovně Room triviálním krokem. Důležité je, abychom pracovali pouze s jedinou instancí. Toho docílíme tvorbou třídy AppDatabase, která bude při požadavku na databázi vždy vracet jedinou instanci[7], která je vytvořena při první žádosti. 4

[7] Jedináček, neboli singleton, je třída, která dovoluje vytvořit pouze jedinou instanci sebe samé.

```
1  abstract class AppDatabase : RoomDatabase(){
2
3     companion object{
4        private var INSTANCE: AppDatabase? = null
5
6        fun getDatabase(context: Context): AppDatabase{
7           if (INSTANCE == null){
8              synchronized(this){
9                 INSTANCE = Room.databaseBuilder(context, AppDatabase
                     ::class.java, "app_db")
10                   .build()
11             }
12          }
13          return INSTANCE!!
14       }
15    }
16 }
```

Zdrojový kód 4: Kotlin

Třída AppDatabase dědí z třídy RoomDatabase a obsahuje *companion object*[8], díky kterému můžeme volat jeho funkce bez potřeby vytvářet instanci třídy. Při zavolání funkce *getDatabase* se nejprve podívá, jestli už je vytvořená instance. Pokud ano, vrátí ji a pokud ne, vytvoří novou instanci, uloží do interní proměnné a vrátí.

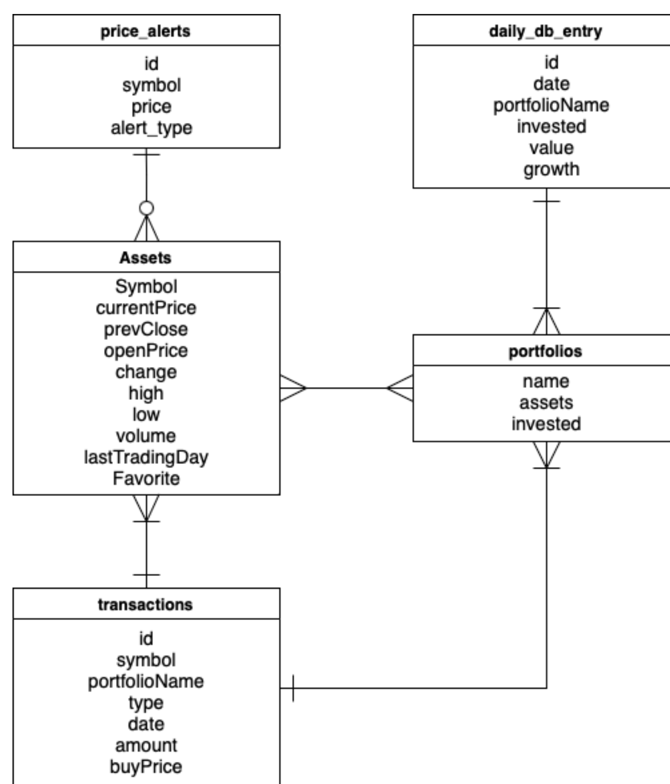
Ke každé tabulce databáze je potřeba vytvořit DAO (Data Access Object), což je rozhraní, které nám umožňuje přístup k záznamům v databázi. Toto rozhraní je anotováno pomocí @Dao a má již předdefinované funkce jako insert, update a delete (tyto funkce je nutné taky anotovat jako @Insert, @Update a @Delete). My si dále můžeme definovat vlastní funkce, které anotujeme jako @Query. Za tuto anotaci do závorky vepíšeme SQL dotaz, který se při zavolání provede a vrátí nám příslušná data z databáze.

5.7.2 Firestore

Struktura Firestore databáze je založená na kolekcích a dokumentech. Kolekci si jednoduše můžeme představit jako složku a dokument jako soubor s daty. Pro každou tabulku interní databáze zařízení jsem vytvořil vlastní kolekci, která obsahuje dokumenty podle ID každého uživatele.

Data jsou zde uložena v datové struktuře *Map<Klíč, Hodnota>*, kde klíč reprezentuje jedinečný identifikátor, podle kterého vyhledáváme data ve struktuře. V mém případě je klíč i hodnota datového typu *String*, přičemž hodnota je záznam databáze převedená do *JSON* formátu.

[8]Companion objekt je jedináčkem, jehož atributy a funkce jsou navázány na třídu, ale nikoliv na instanci třídy.



Obrázek 9: Diagram zobrazující vztahy mezi entitami

Pro převod záznamu na JSON používám knihovnu *Gson*. U převodu dat může dojít k problému s datovými typy. V takovém případě je nutné vytvořit třídu adaptéru, který definuje jakým způsobem se data převádí. V mém byl problém datový typ *LocalDate*. Definoval jsem proto třídu *LocalDateTypeAdapter*, který jsem k gsonu přidal pomocí funkce *registerTypeAdapter*.

Získání dat v praxi vypadá tak, že zavoláme funkci *collection* třídy FirebaseFirestore. Tím se pomyslně dostaneme do složky dané kolekce. Pro ty jsem vytvořil výčtový typ *Collections*, který zahrnuje názvy všech kolekcí, které naše databáze obsahuje. Jako argument volání funkce *collection* dosadíme jeden z těchto názvů. Poté na výsledek volání zavoláme funkci *document*, do jehož argumentu dosadíme uživatelské ID, které je uložené v proměnné *uid* každého uživatele. Tímto způsobem dostaneme data uložená v dokumentu – ta jsou ve tvaru JSONu a musíme je dále zpracovat. 5

5.8 Přihlášení a registrace

Tvorba uživatelského účtu je základní a důležitou funkcí pro fungování aplikace. Při prvním zapnutí aplikace je uživatel vyzván aby si založil účet. Po zadání uživatelských dat, tj. e-mail a 2x heslo, jsou data předána příslušnému viewmodelu, kde je zkontrolován formát e-mailu a shoda zadaných hesel. Pokud je vše v po-

```
1    fun getData(collection: Collections, callback:
        OnDataReturnedListener) = runBlocking {
2       firestore.collection(collection.value).document(
            getCurrentUserID()).get()
3           .addOnSuccessListener {
4              callback.dataReturned(it.data)
5           }
6    }
```

Zdrojový kód 5: Získání dat z Firestore databáze

řádku, je zavolána funkce *createUserWithEmailAndPassword* knihovny Firebase, která uživateli vytvoří účet. Při registraci je také zavolána funkce *sendEmailVerification*, která na registrovanou adresu zašle odkaz, přes který uživatel ověří, že je adresa opravdu jeho. Tento krok je důležitý z toho důvodu, aby si uživatel nemohl založit účet na cizí e-mail, a tím mu znemožnit používání aplikace. Pokud uživatel prokáže, že je adresa jeho, může pokračovat a přihlásit se do aplikace.

Při přihlašování do aplikace se formát dat validuje v příslušném viewmodelu. Mohli bychom data rovnou posílat na server Firebase, a tím ušetřit kód v naší aplikaci, nicméně odpověď serveru může být zdlouhavá. Heslo pochopitelně nelze kontrolovat lokálně, ale tvar e-mailové adresy lze jednoduše zkontrolovat alespoň přes přítomnost zavináče. Pokud je tedy e-mail ve správném tvaru, jsou data odeslána na Firebase, který nám zpět pošle odpověď o existenci uživatele a správnosti jeho hesla. V případě nesprávnosti je uživatel informován, stejně tak jako o chybách jiných – např. uživatel nemá přístup k internetu. Při správnosti je uživatel přesměrován do aplikace a může využívat její plnou funkčnost.

5.9 Aktualizace dat na pozadí

Aby se uživateli aktualizovala data i když aplikaci zrovna nepoužívá, je potřeba vytvořit Intent, který se spustí v daný čas.

Pro spuštění akce v určitou denní hodinu jsem použil třídu *Calendar* spolu se systémovou službou *AlarmManager*. Prvním krokem je vytvořit instanci kalendáře a nastavit čas, ve který chceme data aktualizovat. Já jsem nastavil čas na 9:10. Tento čas má jednoduchý důvod a to ten, že burza se otevírá v 9:00. *AlarmManager* má funkci *setInexactRepeating*, která mimojiné přijímá jako argument čas kalendáře a Intent, který bude v danou dobu spouštět.

Intentem je třída *DataUpdate*, která dědí z třídy *BroadcastReceiver*. Při zdědění z této třídy musíme přepsat metodu *onReceive*, ve které se bude dít všechen kód, který chceme na pozadí vykonat. V mém případě spouštím funkci pro aktualizace a zálohování všech dat.

5.9.1 Využití korutin

Jsou případy, kdy při používání aplikace potřebujeme vykonat nějaký kód na pozadí. Kdybychom tento kód vykonávali na hlavním vlákně, zbytečně bychom tím zmrazili fungování uživatelského rozhraní. Uživateli to není příjemné a vede to k čekání. Proto je potřeba rozdělit vykonávání kódu na vhodná vlákna. V Kotlinu si lze vyhradit vlastní vlákno na speciální účely. Obecně však máme připravené 4 typy, které lze používat:

- *Main* je typ, který spouští kód na hlavním vlákně. Říká se mu také UI vlákno a slouží pro účely, kdy potřebujeme z korutiny změnit uživatelské rozhraní.
- *Default* se používá v případě, kdy potřebujeme provést dlouhý výpočet. Takovýto výpočet by při vykonávání na hlavním vlákně zmrazil UI, což vede k nepříjemnému zážitku s používáním aplikace.
- *IO* už z názvu napovídá, že se bude jednat o Input/Output vlákno. To je speciálně optimalizované pro operaci se vstupy a výstupy, a proto slouží k práci s databází, komunikaci s API nebo zápis a čtení ze souboru.
- *Unconfined* nemá určené konkrétní vlákno, na kterém bude kód vykonaný. Podle kontextu může být spuštěn na *main*, ale třeba i na *default*. Pokud programátor není zkušený, pak není bezpečné tento typ používat. Speciálně pokud používáme nějaké sdílené proměnné, může dojít k chybám.

Obecně je tedy doporučeno používat *IO* a *default* a vyvarovat se tak zbytečným chybám.

5.10 Notifikace

Při rozkliknutí akcie si uživatel může vytvořit upozornění na vývoj ceny, které mu přijde, když daná akcie splňuje určenou podmínku. Tvorba upozornění se skládá ze dvou vstupů – cena a ukazatel, při jaké příležitosti chceme upozornit. Ukazatel se vybírá ze dvou možností, které jsou součástí výčtového typu *AlertType*. Při vložení vstupu se data pošlou do viewmodelu, kde se vytvoří nové upozornění a vloží se do databáze.

Kontrola upozornění se provádí při každé aktualizaci dat, to znamená každé ráno při automatické aktualizaci a nebo při manuální aktualizaci dat uživatelem. Tato kontrola spočívá pouze v porovnání dvou hodnot – hodnoty upozornění uložené v databázi a aktuální hodnoty akcie. Pokud je podmínka splněna, je potřeba vytvořit notifikaci a ukázat ji uživateli.

Pro tvorbu notifikací jsem vytvořil třídu *PriceAlertNotification*, která obsahuje metodu *notify* a bere jako argument text, který se má zobrazit v notifikaci. Prvním krokem je vytvoření notifikace samotné, které probíhá pomocí třídy *NotificationCompat*. Tato třída vytvoří notifikaci a umožňuje nastavit různé parametry. Druhým krokem je vytvoření notifikačního kanál. S tím nám pomůže

třída *Notification Channel*, která pro vytvoření kanálu potřebuje unikátní identifikátor, přes který spojíme kanál s danou notifikací. Kanál obsahuje důležité informace o notifikaci jako důležitost, popis nebo jméno. Na závěr zobrazíme notifikaci funkcí *notify*.

6 Uživatelská příručka

6.1 Přihlašovací obrazovka

Před vpuštěním do aplikace je uživatel vyzván, aby se přihlásil ke stávajícímu účtu, případně si vytvořil nový. Jednu z těchto možností si lze vybrat pomocí tlačítek nad formulářem přihlášení. Přihlášení proběhne po zadání validního e-mailu a příslušného hesla. Pokud je jedna z těchto informací nesprávná, je o tom uživatel informován a není do aplikace vpuštěn.

V případě, že uživatel nemá vytvořený účet, přepne se do registrace (viz obr. 10). Ta probíhá pomocí e-mailu, hesla a ověření hesla. Heslo musí být alespoň 6 znaků dlouhé, jinak je registrace neúspěšná. V případě, že se hesla neshodují nebo je jiný problém v registraci, je uživatel informován.

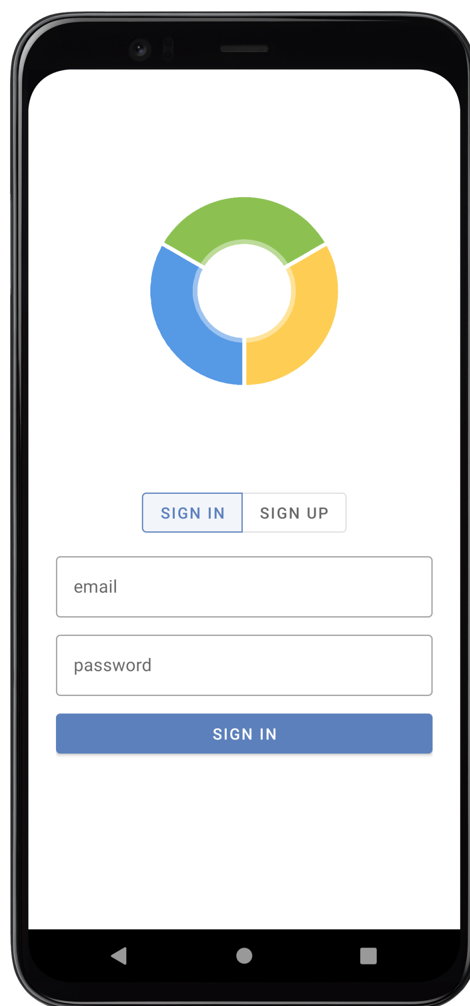
Po registraci je uživatel přesměrován na formulář přihlášení a vyzván, aby ověřil svoji e-mailovou adresu, pod kterou se registroval. Ověření se provede v e-mailové schránce, do které přišel e-mail s odkazem, na který je pro ověření nutné kliknout. Po kliknutí se otevře webová stránka s potvrzením ověření, po jehož úspěšném vykonání se uživatel může vrátit do aplikace. Dále již pokračuje v přihlášení pomocí e-mailu a hesla, kterým se registroval.

6.2 Obrazovka Portfolio

Tato obrazovka obsahuje tři hlavní prvky – vrchní hlavičku s názvem portfolia a možnostmi, grafickou reprezentaci portfolia a seznam aktiv, které portfolio obsahuje.

V hlavičce je umístěn název portfolia, na který když klikneme, objeví se nám seznam s dalšími portfolii, které máme, a které můžeme po kliknutí zobrazit. Další možností je kliknutí na samotnou hlavičku, po kterém se nám zobrazí další akce, jako jsou historie grafu, transakce a predikce. Rozdíl mezi rozbalenou a zabalenou hlavičkou lze vidět na obrázcích 11 a 12. Historie grafu nám zobrazí historický vývoj portfolia, včetně hodnoty portfolia, dat a procentuálního nárustu, případně poklesu. Položka transakce obsahuje výpis všech transakcí, které jsme v daném portfoliu provedli. Kliknutí na predikci nám podle historického vývoje všech aktiv v portfoliu vypočítá jeho průměrný roční nárust a graficky zobrazí predikovanou hodnotu v budoucnosti.

Koláčový graf je ideálním prvkem pro reprezentaci rozdělení dat jednoho celku. Každá část grafu ukazuje hodnotu aktiva portfolia v dolarech. Po kliknutí na konkrétní část koláčového grafu se ve spodní části obrazovky otevře karta obsahující informace o aktivu. Ve vrchní části této karty se nachází tlačítka s označením *plus* a *minus*. Po kliknutí na tlačítko *plus* se otevře nové okno, ve kterém má uživatel možnost přidání daného aktiva do stávajícího portfolia. Obdobná akce se stane po stisknutí tlačítka *minus*, pouze místo přidání se provede akce odebrání. Ve spodní části karty se nachází tlačítko *purchase history*, které uživateli zobrazí transakce provedené s daným aktivem v aktuálním portfoliu.

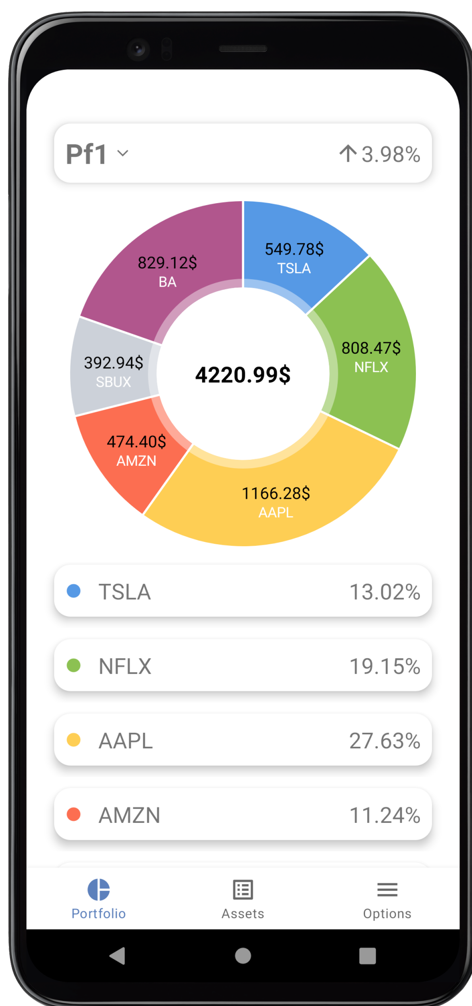


Obrázek 10: Přihlašovací obrazovka

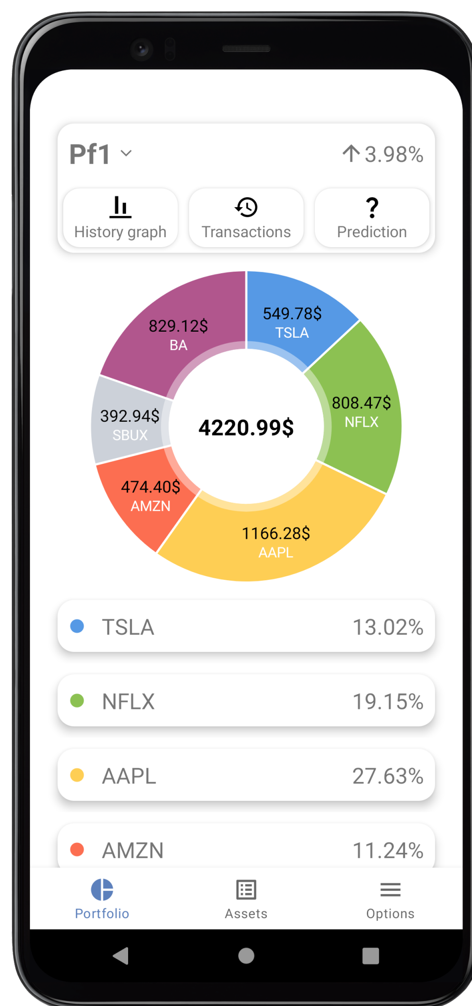
List aktiv ve spodní části obrazovky reprezentuje stejná data jako koláčový graf, jen místo dolarových hodnot jsou vyjádřena v procentech. Po kliknutí na položku v seznamu se opět dostaneme na kartu s informacemi o aktivu.

6.2.1 Vývoj portfolia

Statistický vývoj portfolia od jeho založení je pro uživatele důležitým ukazatelem. Vývoj je reprezentován spojnicovým grafem, viz obrázek 13. Každý bod osy x reprezentuje buď nový den nebo změnu hodnoty portfolia uživatelským zásahem – např. nákup akcie. Osa y poté reprezentuje aktuální hodnotu v dolarech. Po kliknutí do grafu se pod ním zobrazí data odpovídající danému záznamu. Jedná se o základní údaje jako datum, procentuální výnos, množství investovaných peněz a hodnota portfolia.



Obrázek 11: Obrazovka portfolio

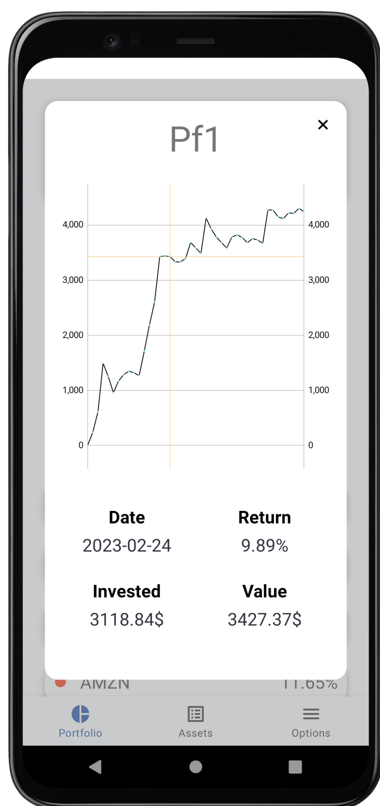


Obrázek 12: Obrazovka Portfolio s menu

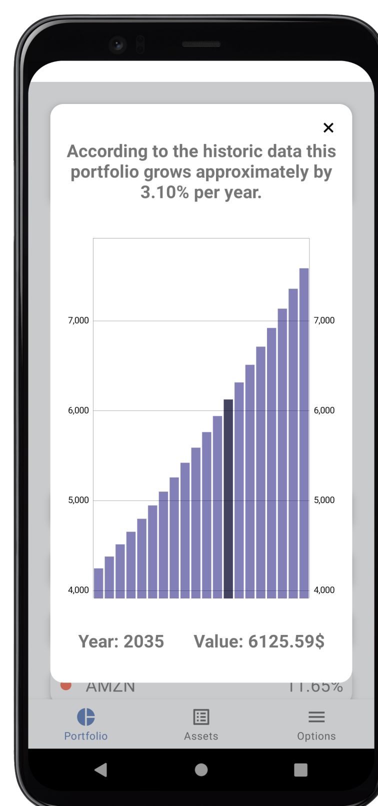
6.2.2 Predikce

Funkce predikce slouží uživateli jako jakýsi ukazatel kvality portfolia. Hodnota růstu je vypočítána jako průměrný roční nárust za posledních 20 let. Predikci tvoříme na základě historických dat s tím, že danou predikovanou hodnotu nemůžeme s jistotou zaručit. Finanční trhy jsou neuvěřitelně nepředvídatelné a ačkoliv můžeme dělat predikce, určitě nikomu nedoporučuji si na to vsázet. Tato hodnota tedy slouží pouze k orientaci.

Budoucí návratnost je reprezentována sloupcovým grafem, kde každý sloupec reprezentuje jeden rok a jeho výška odpovídá hodnotě portfolia v daném roce. Rok i hodnota portfolia je zobrazena pod grafem. Tyto hodnoty se zobrazí po kliknutí na jednotlivý sloupec grafu. Predikci lehce rostoucího portfolia lze vidět na obrázku č. 14.



Obrázek 13: Vývoj portfolia



Obrázek 14: Predikce

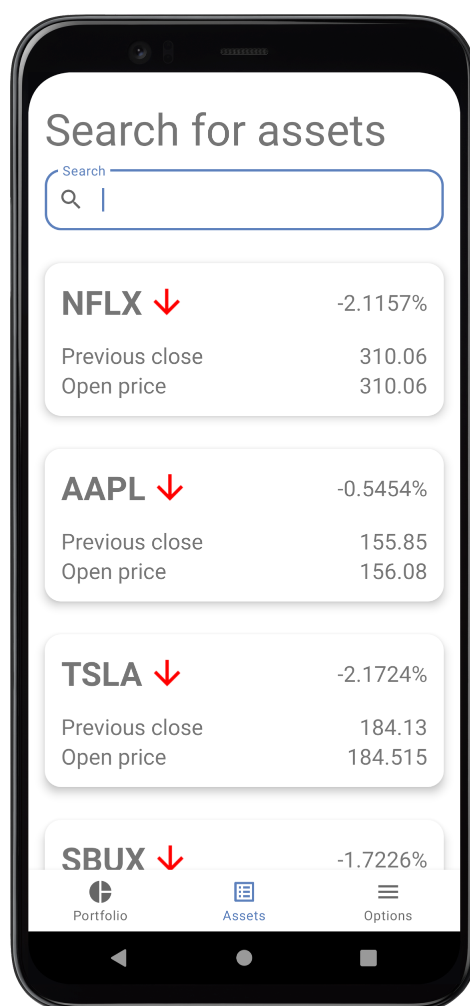
6.3 Obrazovka Aktiva

Tato obrazovka slouží k vyhledávání akcií. Ve vrchní části se nachází vyhledávací řádek, do kterého lze vepsat symbol nebo název akcie, kterou chceme vyhledat. Vyhledávání je omezeno, a proto začne až po zadání druhého znaku do vyhledávacího řádku. Poté se pod tímto řádkem objeví možnosti, které korespondují s uživatelským vstupem. Na levé straně možností jsou vypsány symboly a na straně pravé celý název akcie. Pokud žádné možnosti nejsou ukázány, pak pro zadaný vstup neexistuje akcie nebo telefon není připojen k internetu. Po kliknutí na jednu z možností se otevře okno s detailními informacemi.

Ve zbytku obrazovky je prostor pro akcie, které si uživatel přidá do oblíbených. Tyto akcie jsou zobrazeny v seznamu, přičemž o každé položce jsou zde zobrazeny důležité informace včetně ceny a růstu. Po kliknutí na jakoukoliv položku seznamu se opět otevře okno s detailem. Tímto způsobem lze jednoduše odebrat akcie z oblíbených.

6.3.1 Okno s detailem akcie

Při kliknutí na akcii se otevře okno, kde jsou vypsány podrobnější informace. Hlavní komponentou je graf, který zobrazuje historii vývoje dané akcie. Při podržení prstu a následném posouvání po grafu se uživateli zobrazí ceny v konkrétních



Obrázek 15: Zobrazení oblíbených položek

dnech v minulosti. Příklad detailu akcie zobrazuje obrázek č. 16.

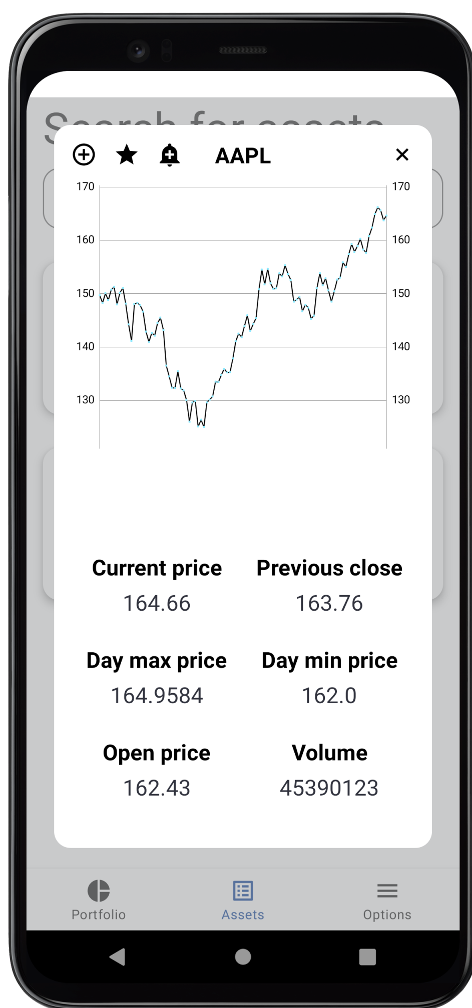
V levém horním rohu se nachází tři ikony, které reprezentují tlačítka pro určité akce. První ikona *plus* otevře nové okno, které uživatele vede k přidání akcie do portfolia. Po kliknutí na ikonu hvězdičky lze aktuální akcii přidat do oblíbených. Oblíbené položky se poté objeví na obrazovce s akciemi, která byla popsána v kapitole 6.3. Poslední ikona zvonečku umožňuje uživateli vytvořit požadavek na upozornění při zadaném vývoji ceny. I v tomto případě se otevře nové okno, které uživatele samo navede k tvorbě upozornění.

6.3.2 Okno přidání akcie

V okně přidání akcie má uživatel na výběr dvě možnosti – přidání akcie na základě množství nebo na základě hodnoty. V obou případech stačí vepsat číselnou hodnotu do příslušného pole a zbytek již aplikace dopočítá. Cenu akcie je možné zvolit aktuální podle dat stažených z internetu. V tomto případě je nutné zaškrt-

nou políčko pro použití aktuální ceny. Pokud uživatel chce zadat vlastní cenu, např. nákup v minulosti, tak cenu stačí vepsat do příslušného pole pro cenu akcie. V případě neplatných hodnot je uživatel upozorněn a není mu umožněno dalšího postupu.

Textové pole na nejspodnější pozici slouží pro výběr portfolia, do kterého chceme aktivum přidat. Po kliknutí na toto pole se objeví nabídka aktuálních portfolií, které jsou k dispozici. Uživatel si z těchto možností může vybrat a pole s názvem se automaticky vyplní. V případě, že chce uživatel vytvořit portfolio nové, pak stačí napsat unikátní název do textového pole. Kdyby se název shodoval s již existujícím portfoliem, přidá se akcie do existujícího portfolia. Příklad vyplnění formuláře pro přidání akcie se nachází na obrázku č. 17.



Obrázek 16: Detailní zobrazení aktiva

Obrázek 17: Přidání aktiva do portfolia

6.3.3 Tvorba upozornění

Je zřejmé, že uživatel nemůže sledovat vývoj cen během celého dne. Proto aplikace nabízí funkci upozornění na cenu. Tlačítkem zvonečku v okně detailu akcie se otevře nové okno s formulářem, kde takovéto upozornění lze vytvořit. Prvním políčkem pro vyplnění je typ upozornění. Máme na výběr dva – typ *NAD CENOU* a *POD CENOU*. Názvy typů mluví samy za sebe, nicméně první zmíněný nás upozorní, když cena akcie přesáhne zadanou cenu v druhém políčku. Druhý typ nás upozorní, když cena klesne pod zadanou cenu. Pro jednodušší porozumění je pod vstupními poli vypsáno upozornění srozumitelně ve větě. Tato věta se při každé změně uživatelského vstupu mění.

Notifikace pak uživateli přijde do telefonu a zobrazí se nahoře v notifikační liště. Po kliknutí na notifikaci je uživatel přesměrován do aplikace. Notifikace samozřejmě někdy přijít nemusí – to se může stát v případě, že cena nikdy nesplní danou podmínku. Tyto podmínky se kontrolují vždy, když se aktualizují data.

6.4 Obrazovka Možnosti

Obrazovka s možnostmi obsahuje další funkce, které nejsou pro uživatele tak důležité, aby byly na hlavních stránkách. V této sekci si tyto funkce rozebereme a popíšeme jejich účel.

6.4.1 Uživatelské možnosti

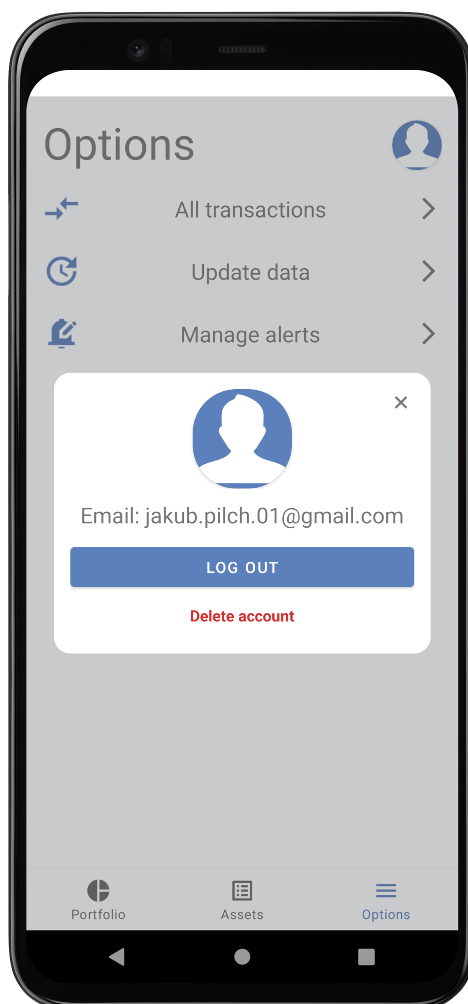
V pravém horním rohu nalezneme ikonu uživatele. Po kliknutí na tuto ikonu nám vyskočí okno s možnostmi. Tyto možnosti nabízí buď odhlášení uživatele z aplikace nebo smazání uživatelského účtu, viz obrázek č. 18. Po kliknutí na odhlášení je uživatel přesměrován na obrazovku přihlášení, data jsou zálohována na server a interní databáze je vymazána. Smazání uživatelského účtu vymaže všechna data o uživateli a je také přesměrován na přihlašovací obrazovku.

6.4.2 Další funkce

Přes tlačítko zobrazení transakcí se nám ukážou všechny transakce, které jsme udělali na aktuálně přihlášeném uživatelském účtu. Tohle je rozdíl oproti tlačítkům na hlavní stránce, kde se zobrazí transakce pouze v daném kontextu, tj. transakce v daném portfoliu, případně samotné akcie. Každá zobrazená transakce obsahuje informaci o symbolu akcie, typu transakce, datu, množství a názvu portfolia.

Další funkcí je aktualizace dat. V případě, že má uživatel pocit zastaralých a neaktuálních dat, má možnost kliknout na tlačítko *aktualizovat data*. Data se obecně aktualizují na pozadí, ale není technicky možné, aby byly aktuální v každý moment. Po aktualizaci dat se aktualizují grafy na hlavní stránce, detaily akcie, vývoj našeho portfolia a také se zkontrolují podmínky uživatelských upozornění. Je proto možné, že po kliknutí na aktualizaci dat nám přijde notifikace o ceně.

Položka *správy upozornění* slouží k prohlížení, případně odstranění upozornění o vývoji cen akcií. Každá položka obsahuje informaci o symbolu dané akcie, a také informaci o tom, kdy upozornit. Na pravé straně každého záznamu se nachází ikona odpadkového koše pro odstranění. Tato akce je nezvratná, proto je důležité být opatrný.



Obrázek 18: Možnosti uživatele

6.5 Jazyk aplikace

Obecně je doporučeno používat aplikaci v anglickém jazyce, nicméně aplikace podporuje také jazyk český. Aplikace automaticky využívá jazyk, který je nastavený v telefonu jako primární. Pro nastavení primárního jazyka je potřeba jít do Nastavení → Systém → Jazyky a zadávání → Jazyky a poté zvolit možnost *Přidat jazyk*. Pokud chceme jazyk nastavit jako primární, je důležité ho přesunout na začátek seznamu. Tuto akci provedeme tím, že umístíme náš prst na

položku v seznamu a posuneme se na jeho začátek. Po restartování aplikace se ihned promítnou změny a jazyk aplikace je změněn.

Závěr

V této bakalářské práci jsme se zaměřili na vývoj mobilní aplikace pro správu investičního portfolia. Vývoj proběhl v jazyce Kotlin s využitím služeb třetích stran, jako je Firebase. Hlavním cílem práce bylo vytvořit uživatelsky přívětivou aplikaci, která uživatelům umožní snadno sledovat své investice.

První čtyři kapitoly jsme věnovali teoretickým tématům této práce. Popsali jsme zadání, konkurenční aplikace a technologie, které lze využít v našem projektu. Zjistili jsme, že trh momentálně nenabízí jednoduchou aplikaci pro sledování portfolia.

V páté kapitole jsme se zaměřili na vývoj samotné aplikace. Prošli jsme celý průběh vývoje od návrhu aplikace až po dokončení finálního produktu. Popsali jsme úskalí a problémy, na které jsme v průběhu programování narazili. I přes pár problémů probíhal vývoj aplikace hladce, a to i s velmi malou předchozí zkušeností s vývojem mobilních aplikací. To i díky jazyku Kotlin, který je velmi intuitivní. Navíc využití služeb třetích stran vývoj výrazně zjednodušilo a zrychlilo.

Šestá a také poslední kapitola je věnovaná především uživatelům. Jsou v ní detailně popsány funkce aplikace, jak je používat a jak se k nim dostat. Uživatelská dokumentace je také doplněna obrázky tak, aby uživatel vše jednoduše našel a pochopil.

Výsledkem této práce je plně funkční mobilní aplikace, která je v aktuální podobě připravena pro publikaci. Aplikace nabízí tvorbu a sledování portfolií, sledování akcií, predikce a také tvorbu upozornění na změnu ceny. Určitě je prostor pro další funkcionalitu, nicméně již v úvodu práce bylo stanoveno, že aplikace má obsahovat jen základní funkce, aby byla vhodná pro začátečníky.

Celkově lze říci, že tato práce byla skvělou zkušeností a základem pro budování hlubších znalostí ohledně vývoje mobilních aplikací. V budoucnu je možno implementovat spoustu funkcionalit, které nebyly náplní této práce. Mezi ně patří například rozšíření cílových platforem, jako je iOS, nebo podpora dalších aktiv, jako jsou kryptoměny nebo třeba cenné kovy. Prostoru pro zdokonalení aplikace je spoustu a věřím, že v budoucnu tuto práci dotáhnu ještě o kus dál.

Conclusions

In this bachelor thesis we focused on the development of a mobile application for investment portfolio management. The development was done in Kotlin language using third party services such as Firebase. The main goal of the work was to create a user-friendly application that allows users to easily track their investments.

The first four chapters were devoted to the theoretical topics of this thesis. We reviewed the assignment, competing applications and technologies that can be used in our project. We found that the market does not currently offer a simple portfolio tracking application.

In chapter five we focused on the development of the application itself. We went through the entire development process from the design of the application to the completion of the final product. We described the pitfalls and problems we encountered during the programming process. Despite a few problems, the development of the app went smoothly, even with very little previous experience in mobile app development. This is also thanks to the Kotlin language, which is very intuitive. In addition, the use of third-party services made the development much simpler and faster.

The sixth and last chapter is mainly dedicated to users. It describes in detail the functions of the application, how to use them and how to access them. The user documentation is also supplemented with pictures so that the user can easily find and understand everything.

The result of this work is a fully functional mobile application that is ready for publication in its current form. The app offers portfolio creation and tracking, stock tracking, forecasting and also the creation of price change alerts. There is certainly room for additional functionality, however, it was established early in the thesis that the app should only contain basic functionality to make it suitable for beginners.

Overall, this work has been a great experience and a foundation for building deeper knowledge regarding mobile app development. In the future, it is possible to implement a lot of functionality that was not the focus of this work. These include, for example, extending target platforms such as iOS, or supporting other assets such as cryptocurrencies or precious metals. There is a lot of room for improvement of the app and I believe that I will take this work even further in the future.

A Obsah elektronických dat

app/

Složka obsahující mobilní aplikaci ve formátu .apk, která slouží pro instalaci.

text/

Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu, tj. zdrojový text textu a příloh, vložené obrázky, apod.

src/

Adresář obsahující kompletní zdrojový kód aplikace.

README.*

Textový soubor s informacemi o instalaci aplikace do mobilního telefonu.

Bibliografie

[1] Free Stock APIs in JSON & Excel | Alpha Vantage. Free Stock APIs in JSON & Excel | Alpha Vantage. Alpha Vantage Inc. 2017. [online] Dostupné z: https://www.alphavantage.co

[2] Vice, R. (2012). MVVM Survival Guide for Enterprise Architectures in Silverlight and Wpf. Packt Pub Limited. [cit. 2021-08-12]

[3] Google. (2021). Android Studio Dolphin | 2021.3.1 Patch 1 [Software]. Dostupné z: https://developer.android.com/studio

[4] Google. (2021). Firebase. [online] Dostupné z: https://firebase.google.com/.

[5] JetBrains. (2021). Kotlin dokumentace. [online]. Dostupné z: https://kotlinlang.org/.

[6] Kotlin: Basics of Companion Objects | by Mark Stevens | The Startup | Medium. Medium – Where good ideas find you. [online]. [cit. 2020-05-07]. Dostupné z: https://medium.com/swlh/kotlin-basics-of-companion-objects-a8422c96779b

[7] ChatGPT. (2023). Popis jazyka Kotlin [online]. Dostupné z: https://chat.openai.com/chat

[8] ChatGPT. (2023). Popis služby Firebase [online]. Dostupné z: https://chat.openai.com/chat

[9] ChatGPT. (2023). Co je to forex [online]. Dostupné z: https://chat.openai.com/chat

[10] PhilJay. MPAndroidChart. GitHub, 2023. [online] Dostupné z: https://github.com/PhilJay/MPAndroidChart