Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Vývoj mobilní Android aplikace pro monitorování změn kurzů kryptoměn

Bc. René Uhrin

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. René Uhrin

Informatika

Název práce

Vývoj mobilní Android aplikace pro monitorování změn kurzů kryptoměn

Název anglicky

Development of mobile Android application monitoring changes of cryptocurrencies exchange rates

---

Cíle práce

Cílem této diplomové práce je vývoj Android aplikace pro monitorování změn kurzů vybraných kryptoměn použitím moderních programovacích jazyků a technologií.

Dílčím cílem je vývoj klientské části použitím poměrně nového programovacího jazyka Kotlin v architektuře MVVM, graficky založeného na Material Designu. Aplikace bude periodicky komunikovat se serverem k získání aktuálních kurzů zvolených kryptoměn, které bude následně vykreslovat do grafu. Pro zobrazování push notifikací bude zvolena knihovna Firebase od firmy Google.

Pomocí této Android aplikace si uživatel bude moci zvolit určitou kryptoměnu, sledovat aktuální vývoj zvoleného měnového kurzu a rozhodovat se na základě těchto údajů pro nákup či prodej dané kryptoměny.

Metodika

Ke splnění uvedených cílů diplomové práce budou analyzovány vhodné technologie a možnosti jejich integrace do Android aplikace. Díky studiu aktuálních literárních zdrojů a využití poznatků získaných během dosavadní praxe bude vývoj této aplikace splňovat moderní postupy.

V rámci diplomové práce bude použit jazyk Kotlin, který je staticky typovaný programovací jazyk běžící nad JVM. Zvolená architektura pro tuto aplikaci je MVVM (Model, View, ViewModel). Tato diplomová práce bude popisovat vývoj moderní mobilní aplikace, přes výběr knihoven a frameworků, až po reálnou komunikaci se serverem třetí strany.

Doporučený rozsah práce

60-80str.

Klíčová slova

android, vývoj, kryptoměny, kurz, kotlin, monitorování, notifikace, knihovny, firebase, mvvm

---

Doporučené zdroje informací

ALLEN, G. *Android 4 : průvodce programováním mobilních aplikací.* Brno: Computer Press, 2013. ISBN 978-80-251-3782-6.

HERODEK, M. *Android : jednoduše.* Brno: Computer Press, 2014. ISBN 978-80-251-4298-1.

LACKO, Ľ. *Vývoj aplikací pro Android.* Brno: Computer Press, 2015. ISBN 978-80-251-4347-6.

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

VÁVRŮ, J. – UJBÁNYAI, M. *Programujeme pro Android.* Praha: Grada, 2013. ISBN 978-80-247-4863-4.

Předběžný termín obhajoby

2018/19 LS – PEF

Vedoucí práce

Ing. Martin Havránek, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 11. 9. 2018

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 09. 03. 2019

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj mobilní Android aplikace pro monitorování změn kurzů kryptoměn" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 25.3. 2019 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Martinovi Havránkovi, Ph.D. za odborné vedení mé diplomové práce. Následně bych rád poděkovat své manželce za podporu a trpělivost při mém studiu na ČZU.

Vývoj mobilní Android aplikace pro monitorování změn kurzů kryptoměn

Abstrakt

Diplomová práce se zabývá vývojem mobilní aplikace pro operační systém Android s využitím moderních technologií a postupů. Při vývoji je důraz kladen především na vytvoření architektury aplikace s ohledem na její životní cyklus a implementaci generických tříd pro přehlednější finální kód. Dále je při vývoji využíván návrhový vzor nazývaný Fasáda pro budoucí snadnější použití knihoven třetích stran. Navrhnutí a vytvoření aplikace je založeno na studiu aktuálních literárních zdrojů a využití vědomostí získaných během praxe v oboru informačních technologií.

V teoretické části diplomové práce je pozornost věnována především popisu použitých programovacích jazyků, moderních technologií a postupů. Dále jsou blíže specifikovány vybrané kryptoměny a v neposlední řadě je nastíněna technologie blockchain.

Praktická část diplomové práce obsahuje samotný vývoj mobilní Android aplikace. K naprogramování nativní aplikace je použit objektově orientovaný jazyk Java a objektově orientovaný funkcionální jazyk Kotlin. Reaktivní programovaní v aplikaci je umožněno přidáním knihovny RxJava. Aplikace poskytuje informace nejen o aktuálním kurzu, ale i historickém vývoji šesti předem vybraných kryptoměn a umožňuje zaslání notifikace uživateli při dosažení zvoleného kurzu.

Klíčová slova: Android, vývoj, kryptoměny, kurz, Kotlin, monitorování, notifikace, knihovny, Firebase, MVVM

Development of mobile Android application monitoring changes of cryptocurrencies exchange rates

Abstract

The diploma thesis is concentrated on development of mobile Android application with utilization of modern technologies and methods. During the development an emphasis is put primarily on creation of an application architecture with regard to its life cycle and an implementation of generic classes for an uncluttered final code. A design pattern called Facade is also used during the development for easier future usage of libraries of third parties. Design and creation of the application is based on current literary sources and usage of knowledge gained during practice in the field of information technology.

The theoretical part of the diploma thesis focuses primarily on a description of used programming languages, modern technologies and methods. Afterwards selected cryptocurrencies are specified in more detail and last but not least, the blockchain technology is outlined.

The practical part of the diploma thesis contains the development of the mobile Android application itself. Object oriented programming language Java and object oriented functional programming language Kotlin are used for programming of the native application. Reactive programming in this application is made possible by addition of RxJava library. The application provides information not only about current exchange rates, but also about historical development of the six beforehand selected cryptocurrencies and allows sending notifications to user, when elected exchange rate it achieved.

Keywords: Android, development, cryptocurrency, rate, Kotlin, monitoring, notifications, libraries, Firebase, MVVM

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

API – Application Programming Interface
JVM – Java Virtual Machine
MVVM – Model View ViewModel
REST – Representational State Transfer
SDK – Software Development Kit

1 Úvod

Mobilní telefony prošly za dobu své existence značným progresem. Dříve nebyly zdaleka tak běžné, jako jsou v dnešní době. Také funkce poskytované v raném stádiu vývoje telefonů byly omezeny pouze na psaní SMS zpráv a volání. V dnešní době je široká veřejnost zvyklá na postupně se navyšující hardwarové parametry mobilních telefonů, které jsou už téměř stejně výkonné, jako některé stolní počítače. Nespornou výhodou je každým dnem se zvyšující počet aplikací na mobilních platformách v obchodech, na jejichž vývoji pracuje velké množství vývojářů po celém světě. Díky výkonnému hardwaru a technologické vyspělosti jsou firmy schopné každým rokem přinášet nové funkce mobilních telefonů, na jejichž základě se stávají nedílnou součástí každodenního života široké veřejnosti.

Jedním z velkých milníků poslední doby je v případě mobilních telefonů jednoznačně možnost připojení na internet pomocí mobilního internetu nebo pomocí Wi-Fi. S touto funkcí má uživatel možnost být neustále připojený k síti a nepřetržitě sledovat nejen nejnovější zprávy ze světa nebo poslední výsledky sportovních utkání, ale i aktuální vývoj na burzách. Lidé tak mohou například hlídat své investice pomocí mobilního telefonu a internetového připojení při cestě do práce z jakéhokoliv části světa.

Stále častěji mobilní telefony nahrazují i kreditní karty, jelikož mohou být používány již i pro bezhotovostní platby, nejen pomocí fiat peněz, ale také pomocí kryptoměn. Tyto virtuální mince se zejména v poslední době používají také pro investování volných peněžních prostředků stejně jako např. akcie. Je možné s nimi nadále obchodovat na burze nebo měnit ve směnárně. V této situaci je každá vteřina rozhodující, a proto je důležité mít neustálý přístup k aktuálnímu kurzu a pravidelně sledovat jeho změny i v krátkém časovém úseku.

V rámci této diplomové práce bude pozornost zaměřena na výše uvedenou problematiku, konkrétně na vývoj mobilní Android aplikace pro monitorování změn kurzu vybraných kryptoměn. Práce je rozdělena do několika částí, přičemž hlavní pilíře

tvoří teoretická a praktická část. Jednotlivé části a jejich kapitoly jsou blíže popsány níže.

Teoretická část práce se nejprve bude zabývat moderními technologiemi, které jsou následně použité pro implementaci výsledné aplikace pro diplomovou práci. Konkrétně je pozornost zaměřena na popis použitých programovacích jazyků, jejich historie a hlavní vlastnosti. Dále je zmíněna architektura s přidanými knihovnami, se kterými se bude následně pracovat. V závěru teoretické části bude pozornost věnována vybraným kryptoměnám a technologii blockchain.

Praktická část práce bude zaměřena na implementaci projektu diplomové práce. V rámci této části bude popsán postupný vývoj zvolené architektury, generických tříd pro snadnější implementaci, využití návrhového vzoru fasáda pro usnadnění použití knihoven třetích stran a následná implementace logiky. Součástí praktické části budou nejen ukázky kódů a uživatelského rozhraní, ale také testování.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této diplomové práce je vývoj mobilní Android aplikace pro monitorování změn kurzů vybraných kryptoměn použitím moderních programovacích jazyků a technologií. Dílčím cílem je vývoj klientské části použitím poměrně nového programovacího jazyka Kotlin v architektuře MVVM, graficky založeného na Material Designu.

Aplikace bude periodicky komunikovat se serverem k získání aktuálních kurzů zvolených kryptoměn, které bude následně vykreslovat do grafu. Pro zobrazování push notifikací bude zvolena knihovna Firebase od firmy Google. Pomocí této Android aplikace si uživatel bude moci zvolit určitou kryptoměnu, sledovat aktuální vývoj zvoleného měnového kurzu a rozhodovat se na základě těchto údajů pro nákup či prodej dané kryptoměny.

2.2 Metodika

Ke splnění uvedených cílů diplomové práce budou analyzovány vhodné technologie a možnosti jejich integrace do Android aplikace. Díky studiu aktuálních literárních zdrojů a využití poznatků získaných během dosavadní praxe bude vývoj této aplikace splňovat moderní postupy.

V rámci diplomové práce bude použit jazyk Kotlin, který je staticky typovaný programovací jazyk běžící nad JVM. Zvolená architektura pro tuto aplikaci je MVVM (Model, View, ViewModel). Tato diplomová práce bude popisovat vývoj moderní mobilní aplikace, přes výběr knihoven a frameworků, až po reálnou komunikaci se serverem třetí strany.

3 Teoretická východiska

3.1 Android

Platforma Android je operační systém pro mobilní telefony, tablety a chytré televize založený na jádře Linuxu a dostupný pod licencí Apache 2.0 (open source). Platforma Android je vytvořena firmou Google pod hlavičkou společností Open Handset Alliance. Tento operační systém mohou modifikovat a využívat i jiní výrobci telefonů, ale musí dodržet předem dané podmínky. (ANDROID/HISTORY, 2018)

Ve Spojených státech amerických, konkrétně v Kalifornii, byla založena firma Android Inc. Andym Rubinem, Richem Minerem, Nickem Searsem a Chrisem Whitem v říjnu roku 2003. Tento projekt byl Googlem odkoupen v roce 2005 a stal se její dceřinou společností. Projekt nadále pokračoval pod vedením Andyho Rubina, kde jeho tým vývojářů vyvíjel operační systém a získal několik patentů v rámci mobilních technologií. Pouhé dva roky poté bylo vytvořeno výše zmíněné uskupení Open Handset Alliance, kde bylo více než 20 členů (mezi nimi byly firmy jako Google, HTC, Intel, LG, Motorola, Qualomm, Samsung a mnoho dalších). V den založení konsorcia byl uveden první produkt Platforma Android. První oficiální verzi operačního systému Android firma Google představila 23. září 2008. Jednalo se o T-Mobile G1 (HTC Dream). (ANDROID/HISTORY, 2018)

V současné době lze nalézt čistý (vyrobený přímo firmou Google bez přidaných komponent od jiných poskytovatelů) operační systém v telefonech Pixel. Vzhledem ke skutečnosti, že je každoročně pouze nízké procento telefonů aktualizováno na novější verzi, přistoupila firma Google na vytvoření programu Android One. Ten má zajistit včasnou aktualizaci systému i u jiných výrobců mobilních telefonů (např. Xiaomi, Motorola a několika dalších).

Platforma Android je jedna z nejpoužívanějších operačních systémů pro mobilní telefony na světě.

3.1.1 Java

Mezi nejvýznamnější data v historii Javy patří rok 1991, kdy představil tým vývojářů James Gosling, Bill Joy, Mike Sheridan a Patrick Naughton svůj nový projekt, který byl reprezentován s hlavním cílem vytvořit systém pro domácí spotřebiče. Projekt byl následně přejmenován na Green Project (Zelený Projekt) a týmu vývojářů se říkalo Green Team (Zelený tým). Za vedoucího projektu se považoval James Gosling. Pro svůj projekt nejdříve zvolili programovací jazyk C++, který však následně shledali za nevhodný pro jeho způsob použití. Z tohoto důvodu začali navrhovat a realizovat nový programovací jazyk, při jehož vytváření se, mimo jiné, inspirovali právě jazykem C a C++. (SCHILDT, 2017)

Nový programovací jazyk byl pojmenován anglickým názvem Oak (česky dub), protože tento typ stromu rostl před okny kanceláře, kde pracovali. V roce 1995 Green Team musel označení Oak přejmenovat na Java (americké slangové označení pro šálek kávy), protože jazyk s názvem Oak již existoval. (SCHILDT, 2017)

Java byla poprvé představena široké veřejnosti na mezinárodní konferenci SunWorld v roce 1995 jako produkt společnosti Sun Microsystems (později sloučené s Oracle Corporation). Následující rok byl vydán první Java Developement Kit (JDK 1.0), kde bylo vše potřebné pro tvorbu apletů. (SCHILDT, 2017)

V průběhu roku 2006 společnost Sun Microsystems zveřejnila většinu svého zdrojového kódu pod licencí GPL (General Public License). Zlomový moment pro vývojáře přisel s verzí SE 8, která umožňovala programování s lambda funkcemi, které otevřely dveře novým programovacím postupům. Poslední verzí je Java SE 10, která byla vydána na začátku roku 2018.

Vzhledem k velké použitelnosti se lze s tímto programovacím jazykem setkat v aplikacích všeho druhu (od mobilních aplikací a vestavěných systémů, až po servery a superpočítače na nejvyšší úrovni).

Mezi nejznámější platformy patří:

- Java SE (Standart Edition) – původní Java, která byla postupně rozšiřována a ze které vycházejí ostatní platformy.
- JavaCard - umožňuje běh aplikací na zařízeních, které mají vlastní paměť (např. paměťové karty).
- Java ME (Micro Edition) – hlavní využití této platformy bylo pro malá zařízení a zařízení s omezenými výpočetními možnostmi (dříve pro mobilní telefony).
- Java EE (Enterprise Edition) – určená pro vývoj a správu hlavně backend serverů, podnikových aplikací a informačních systémů.

Syntaxe tohoto programovacího jazyka je velice podobná dříve zmíněným C a C++, ale objektové vlastnosti jazyka jsou inspirovány Smalltalkem a Objective-C. Největší rozdíl od C a C++ je odstranění některých nízkoúrovňových konstrukcí (např. ukazatelů), které jsou nahrazeny tzv. haldou (součást správy paměti), kde jsou uloženy všechny potřebné reference. Správa paměti je automaticky řešená pomocí Garbage collectoru. Další velkou změnou je odstranění mnohonásobné dědičnosti. (SCHILDT, 2017)

Programovací jazyk Java je silně typovaný objektově orientovaný jazyk, který běží nad JVM (Java Virtual Machine). Hlavními výhodami jazyka jsou robustnost, výkonnost, dynamičnost a víceúlohovost.

Díky výše uvedené robustnosti Javy se může vývojář při psaní zdrojového kódu částečně vyvarovat potencionálním chybám (správa paměti a ukazatelé, příkaz goto apod.). Dále využívá dříve zmíněnou silně typovou kontrolu, což v praxi znamená, že vytvořená proměnná musí mít definovaný datový typ. Ztráta výkonu je téměř zanedbatelná, vzhledem ke skutečnosti, že je Java interpretovaný jazyk. Překladač může totiž pracovat v tzv. režimu „just-in-time“, což znamená, že je do strojového kódu přeložen pouze kód, který je opravdu potřeba. (SCHILDT, 2017)

Vytvořený program běží na jakémkoliv operačním systému, ale je zapotřebí mít k dispozici (nainstalovaný) požadovaný virtuální stroj, aby bylo možné program spustit.

Pokud počítáme s multiplatformním užíváním, lze v programu přizpůsobit vzhled a chování aplikace tak, aby UX odpovídalo dané platformě.

Velkou předností tohoto programovacího jazyka je skutečnost, že se Java SE neustále vyvíjí. Nové verze Javy jsou vydávány nepřetržitě, přičemž jejich autoři s každou další verzí usilují o to, aby se co nejvíce přibližovala moderním trendům v programování. V projektu této diplomové práce bude použita verze Java 8, která se v Androidu mírně liší od verze Java SE 8.

3.1.2 Kotlin

Společnost JetBrains představila svůj projekt, konkrétně nový programovací jazyk s názvem Kotlin, poprvé v červenci roku 2011. Firma si od projektu slibovala především větší variabilitu pro svůj vývojářský tým (alespoň stejná rychlost kompilace jako u čisté Javy, jednodušší syntaxe apod.). O rok později uvolnila firma JetBrains Kotlin jako open source pod licencí Apache 2. První oficiální stabilní vydání Kotlinu proběhlo až v roce 2015. Od tohoto roku je zachovávána zpětná kompatibilita. Vzhledem k velmi pozitivnímu ohlasu a popularitě v Android komunitě po celém světě, byl Kotlin Googlem oficiálně představen jako další podporovaný programovací jazyk na Google I/O v roce 2017. (KOTLIN, 2019)

Kotlin je staticky typovaný programovací jazyk, který běží nad JVM, stejně jako Java. Taktéž umožňuje používání lambda výrazů, což dělá z Kotlinu jak objektový, tak funkcionální programovací jazyk. Syntaxe jazyka je velice podobná Javě, C#, JavaScriptu a Groovy. Hlavní předností tohoto programovacího jazyka je interoperabilita, bezpečnost a jednoduchost. (KOTLIN, 2019)

Na základě interoperability Kotlinu lze v jednom projektu zároveň používat Kotlin i Javu (popřípadě Kotlin a JavaScript), tudíž je možné i nadále používat dodatečné knihovny psané v Javě. Hlavní motivací bylo navrhnout spolehlivý, objektově orientovaný jazyk, který je lepší než Java, ale je s ní v každém směru interoperabilní. (KOTLIN, 2019)

Kotlin je tzv. null bezpečný jazyk, protože pracuje s možnými nulovými situacemi v době kompilace, díky čemuž lze snadněji zabránit výjimkám za běhu aplikace. Aby mohl být objekt nulový, musí se explicitně tato možnost specifikovat. Programovací jazyk Kotlin bude použit i v rámci této diplomové práce.

3.1.3 Gradle

V začátcích vývoje Java aplikací neměli programátoři na výběr příliš mnoho buildovacích nebo kompilovacích nástrojů. Prvním větším projektem, který se touto problematikou zabýval, byl nástroj Apache Ant (Another Neat Tool), který vychází z konceptu Make a byl oficiálně představen roku 2000. Apache Ant umožnil vývojářům sestavení Java projektů, které jsou přenositelné a také rozšiřitelné pro stávající nebo další projekty. I v tomto projektu se však vyskytovaly nedostatky. Výpis chyb obsahoval nedůležité informace, buildovací skripty se nedaly znovu použít a řízení závislostí bylo nedostatečné. (VSKP.VSE, 2013)

Z uvedených důvodů se začal vyvíjet další buildovací nástroj, Maven, který je dodnes poměrně dost známý a oblíbený. Běžně známé konvence mezi programátory (hledání zdrojového kódu díky doporučené struktuře projektu, podpora continua, generované stránky projektu apod.) pochází právě z tohoto nástroje. I přes prvotní nadšení mezi vývojáři, nebyl však ani tento produkt dostatečně flexibilní. Otevřela se tak cesta novým buildovacím nástrojům, které by kombinovaly pozitivní vlastnosti předešlých verzí. Během následujícího období jich vzniklo spoustu, ale většina z nich se nakonec mezi programátory neprosadila. Za úspěšný projekt je považován až nástroj zvaný Gradle.

Během vývoje tohoto buildovacího systému se usilovalo o zkombinování flexibility Antu, řízení závislostí a konvencí jako v Mavenu a nejlepších vlastností z nástrojů Make, Ivy, Rake, Gant a Scons. Gradle má podporu ve vývojových prostředích Eclipse, IntelliJ IDEA (Android Studio) a Netbeans (přestože podpora není oficiálně uvedena v dokumentaci). (VSKP.VSE, 2013)

V Android aplikacích se hojně používají knihovny třetích stran, sestavují se balíčky pro různé typy obrazovek nebo speciální nastavení pro finální verzi aplikace.

K automatizování těchto procesů v Androidu se používá již zmíněný systém Gradle. Původně vznikl pro vyvíjení v jazyce Java, Groovy a Scale, ale v současné době se používá např. i pro platformu Android (tudíž i pro jazyk Kotlin). V této diplomové práci je Gradle použit pro automatizaci stahování knihoven nebo verzování aplikace. (VSKP.VSE, 2013)

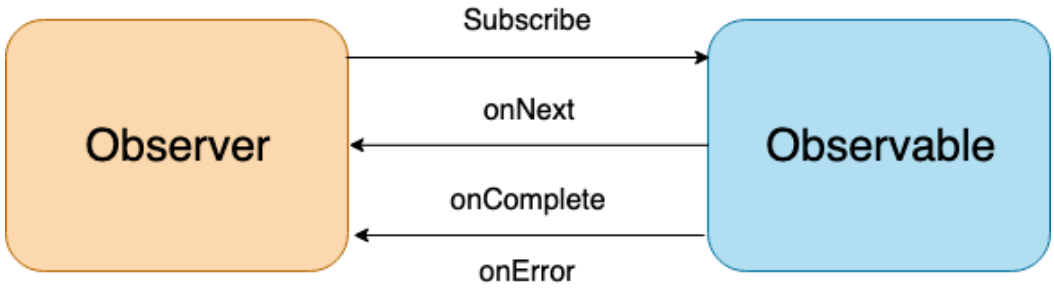
3.1.4 RxJava

Reaktivní programování poskytuje jednoduchý způsob programovat bloky kódu, které se hlavně využívají pro zpracování dlouhých operací, asynchronně. Mezi dlouhé a náročné operace, typické pro asynchronní zpracování, patří například komunikace se serverem (musí se provádět zásadně asynchronně, jinak aplikace za běhu spadne z důvodu *NetworkOnMainThreadException*), zpracování velkých souborů (např. konvertování obrázků) nebo náročné výpočty. Hlavním důvodem pro vznik reaktivního programování byla snaha nalézt způsob, jak vývojářům zjednodušit tvorbu interaktivního prostředí pro uživatele, animací nebo vykreslování předmětů v reálném čase.

Zajistit, aby vývojář programoval reaktivně, je možné více způsoby. První možností je výběr již existujícího ryze reaktivního programovacího jazyka (Clojure, Elixir, Elm, F#, Haskell, Idris, Scala). Další možností je zvolit si takový programovací jazyk, který umožnuje připojení knihovny, která tuto funkcionalitu přidá. Mezi takové jazyky patří například Java nebo Scala. V této diplomové práci bude mimo jiné využit jazyk Java, z tohoto důvodu je zvolena dodatečná knihovna RxJava (Rx je anglická zkratka Reactive extensions).

Hlavní dvě komponenty RxJavy, které jsou funkčně velmi úzce spjaté, se nazývají *Observable* a *Observer* (nazývané taky *Subscriber*). *Observable* je zdrojem dat a poskytuje je na cílené místo pomocí *Observeru*, který může vysílat libovolný počet objektů (nebo událostí). Posílání objektů může skončit úspěšně (vyvolá se metoda *onComplete()*), chybou (přejde se do metody *onError()*) nebo může být zdrojem posílání nekonečného sledu událostí. Na obrázku č. 1 je výše uvedený tok možných událostí vyobrazen.

Obrázek 1 – Tok událostí mezi *Observable* a *Observer* (*Subscriber*)



Zdroj: vlastní zpracování

Na *Observer* může být připojen libovolný počet *Observable* proměnných. Přes *Observable* je možné vysílat 0 až n událostí, které končí úspěchem nebo chybným stavem. Pokud se nová událost nebo objekt posílá z *Observable*, metoda *onNext()* bude volána na každém *Observeru*. Toto pravidlo platí i pro výše zmíněné metody *onComplete()* a *onError()*. Použití metody *onNext()* značí, že bude posílání událostí nadále pokračovat. *Observers* a *Observable* se rozdělují dle typu plánované komunikace. (REACTIVEX, 2019)

Flowable se používá, stejně jako u předchozího typu, při vysílání 0 až n událostí, které skončí úspěchem nebo chybou. Dále podporuje Backpressure, který slouží pro kontrolu rychlosti vysílání. (REACTIVEX, 2019)

První větší změna přichází s typem *Single*, který dokáže pracovat pouze s jedinou událostí. Pokud událost proběhne v pořádku, vrátí se zpracovaná položka. V opačném případě je vrácena chyba. (REACTIVEX, 2019)

V reaktivním programování lze také použít volitelný argument, který se nazývá *Maybe*. Při zpracování události může programátor jako návratovou hodnotu očekávat zvolenou položku, prázdný argument nebo chybu. (REACTIVEX, 2019)

Posledním základním typem v reaktivním programování RxJava je *Completable*. Jedná se o reaktivní verzi *Runnable*, který vysílá pouze událost reprezentující úspěch nebo chybu. (REACTIVEX, 2019)

3.1.5 Databinding

Snahou programátorů je především co nejvíce minimalizovat finální kód, jednak kvůli přehlednosti v projektu, ale také z důvodu snížení pravděpodobnosti chyb, které by mohly vzniknout v rámci nadbytečného množství kódu. Data binding poskytuje způsob spojení uživatelského rozhraní a logiky obrazovky, které dokáže automaticky aktualizovat hodnoty bez dodatečného dotazování nebo ukládání reference na grafický element. Na základě nepřímé propojenosti uživatelského rozhraní a logiky obrazovky umožňuje programátorům rozdělení zdrojového kódu do vrstev. Data binding je také úzce spjatý s architekturou MVVM, která je popsána níže.
(DEVELOPER.ANDROID/DATA-BINDING, 2019)

Skutečnost, že díky data bindingu již není nutné používat metodu *findViewById*, usnadňuje Android vývojářům komunikaci s grafickými elementy. Stačí pouze vytvořit danou *Observable* proměnnou, která bude komunikovat s uživatelským rozhraním a poté jen aktualizovat hodnoty, které chceme poskytnout uživateli. Jak napovídá obrázek č. 2, pro možnost použití data bindingu stačí pouze přidat blok kódu do Gradle souboru projektu. (DEVELOPER.ANDROID/DATA-BINDING, 2019)

Obrázek 2 – Zpřístupnění data bindingu v projektu

```
android {
    dataBinding {
        enabled = true
    }
}
```

Zdroj: vlastní zpracování

Ke každému XML souboru, který definuje vzhled dané obrazovky (komponenty apod.) je potřeba explicitně přidat tag, který reprezentuje přítomnost data bindingu. Mezi tyto tagy patří *layout* a *data*. První jmenovaný (tag) reprezentuje celý XML soubor. Data slouží pro posílání informací, které lze v XML použít. Mezi těmito informacemi nemusí být jen zmíněný ViewModel, ale také context, handlery nebo prosté primitivní typy (int, long, float apod.). Proměnná *name* reprezentuje název, se kterým se nadále pracuje v XML souboru, zatímco type je přesné umístění třídy

v projektu. (DEVELOPER.ANDROID/DATA-BINDING, 2019). Tato skutečnost je zobrazena na obrázku č. 3.

Obrázek 3 – Užití data bindingu v XML souboru

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
>
    <data>
        <variable
                name="viewModel"
                type="cz.reneuhrin.masterthesesproject.screen.main.CryptoListViewModel"
        />
    </data>

</layout>
```

Zdroj: vlastní zpracování

Vytvořené proměnné mohou být použity dvojím způsobem. Jedním je režim čtení a druhým čtení se zápisem (tzv. two-way data binding). Při kompilaci kódu je potřebná syntaxe v XML souboru pro zjištění atributů používajících přímé čtení z proměnné pomocí data bindingu. Musí být použit znak @ a následně závorky {}, kde se použije název proměnné a pomocí tečkové anotace si vybereme požadovanou informaci. Pokud chceme i zápis do *Observable* proměnné, jako druhý znak uvedeme znak =. (DEVELOPER.ANDROID/DATA-BINDING, 2019). Příklad užití obou režimů je demonstrován níže na obrázku č. 4.

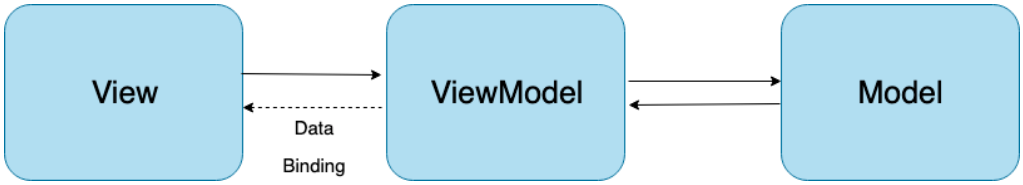
Obrázek 4 – Porovnání režimu čtení s two-way data bindingem

```
android:text="@={viewModel.dateRange}"
app:visibility="@{!viewModel.calendarPickerVisibility}"
```

Zdroj: vlastní zpracování

Celý proces začíná u interakce uživatele (kliknutím na tlačítko, napsáním textu a mnoha dalšími způsoby komunikace). Jak je patrné z obrázku č. 5, *Observable* zašle notifikaci o změně do ViewModelu, který na to zareaguje naprogramovanou logikou a vrátí výsledek zpět pomocí data bindingu uživatelskému rozhraní, kde se změny projeví.

Obrázek 5 – Tok událostí v data bindingu



Zdroj: vlastní zpracování

Data binding byl představen na Google I/O v roce 2016, který je podporován přímo firmou Google. Výše uvedená knihovna je v rámci této diplomové práce využívána.

3.1.6 MVVM (Model, View, ViewModel)

Hlavním nedostatkem architektur používaných dříve byla úzká spjatost mezi logikou a grafickými komponentami. Potíže s těmito architektonickými vzory nastávaly ve chvíli, kdy uživatel např. zadal pokyn k rotování obrazovky a zároveň současná akce odkazovala na již neexistující grafický prvek. Při životním cyklu aplikace se totiž v metodě *onDestroy()* celá současná aktivita zničí a v metodě *onCreate()* znovu vše vytvoří. Díky architektuře MVVM je možné těmto problémům předejít.
(DEVELOPER.ANDROID/VIEWMODEL, 2019)

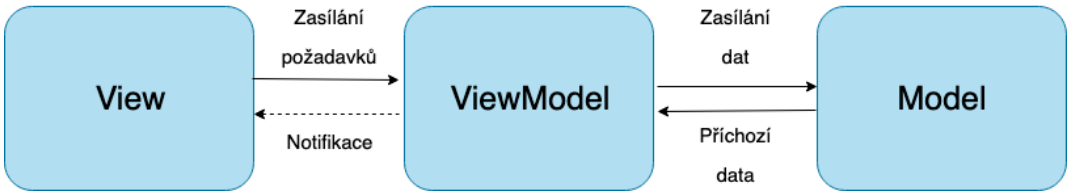
Při vývoji architektury aplikace pro platformu Android se odděluje uživatelské rozhraní od logiky dané obrazovky. Předností je testovatelnost jednotlivých metod pomocí unit testů, opakované použití jednoho ViewModelu napříč různými obrazovkami a škálovatelnost kódu. Přístup MVVM se skládá ze tří základních složek, z nichž má každá svou vlastní samostatnou a jedinečnou roli.
(DEVELOPER.ANDROID/VIEWMODEL, 2019)

Modelová vrstva představuje data ze serveru, cache nebo databáze a business logiku aplikace. Doporučenou implementací této vrstvy je možnost připojení pomocí *Observable* proměnné, díky které se tato vrstva naprosto oddělí od zbylých vrstev, přičemž je i nadále možné komunikovat pouze s vrstvou ViewModel.
(DEVELOPER.ANDROID/VIEWMODEL, 2019)

ViewModel je prostředníkem mezi vrstvou Model a View. Data stáhnutá vrstvou Model jsou připravena ke zpracování ve ViewModelu (lze data seřadit, některá smazat, poupravit apod.), který pošle už upravená data do vrstvy View. Důležitou implementační podmínkou v této architektuře je nepřítomnost jakékoliv reference grafických komponent z View vrstvy. Veškerá komunikace probíhá skrz notifikace. (DEVELOPER.ANDROID/VIEWMODEL, 2019)

Poslední vrstvou v této architektuře je View. Jejím jediným a hlavním cílem je pouze vykreslit data pro uživatele, popřípadě předat informaci ViewModelu o interakci uživatele. (DEVELOPER.ANDROID/VIEWMODEL, 2019). Všechny tyto vrstvy a jejich vzájemné vztahy vyjadřuje obrázek č. 6.

Obrázek 6 – Vrstvy architektury MVVM a jejich vzájemné vztahy



Zdroj: vlastní zdroj

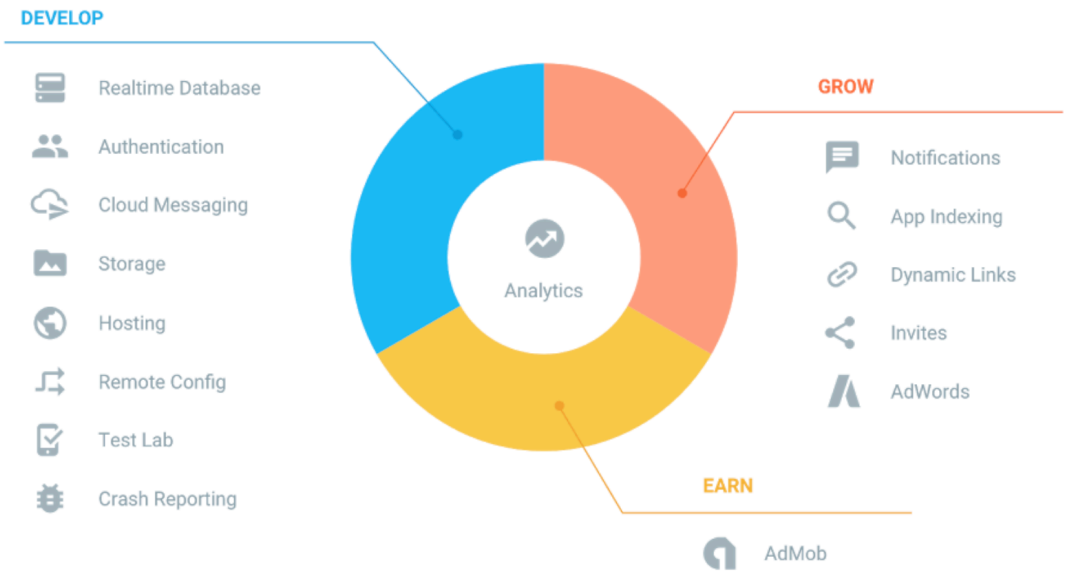
3.1.7 Firebase

Platforma Firebase slouží jako doplněk při vývoji mobilních a webových aplikací. Poskytuje vývojářům pestrou paletu nástrojů a služeb, díky kterým jsou schopni rozvíjet funkcionalitu dané aplikace.

Koncept, který měl Firebase původně, byl naprosto odlišný. V roce 2011 vznikla společnost Envolve, která poskytla programátorům API pro integraci online chatu do webových stránek. Pro firmu Envolve bylo velkým překvapením, že programátoři API používali k jinému účelu, než byl vytvořen, a to k synchronizování dat v reálném čase (např. skóre ve hrách, stav hry). Tento důvod přivedl zakladatele Jamese Tamplina a Andrewa Leeho k myšlence oddělit chatovací systém od architektury pro synchronizování dat v reálném čase.

V dubnu roku 2012 byla vytvořena firma Firebase jako nová společnost poskytující výše uvedenou službu. Dva roky poté byla firma zakoupena společností Google a rychle se vyvinula do své dnešní podoby. Na obrázku č. 7 níže je ilustrace služeb, kterými nyní knihovna disponuje. V diplomové práci budou využívány notifikace, které poslouží k informování uživatele o dosažení předem zvoleného kurzu.

Obrázek 7 – Přehled služeb platformy Firebase



Zdroj: https://www.spaceotechnologies.com/implement-firebase-push-notification-android-app

Koncept rozdělení mobilní aplikace na serverovou a klientskou část je nejčastějším schématem, podle kterého se budují současné aplikace s dynamickým obsahem. Klientské aplikace (frontend) bývají připojovány k serverové části (backend), který s daty provádí největší část práce (autentifikace uživatele při přihlášení, uložení dat v databázi, operace s daty apod.). Propojení aplikace s Firebase serverem není předmětem této diplomové práce, tudíž budou při simulaci práce serveru využívány jen mockované notifikace (vytvářeny lokálně), podle uživatelem předem zadaných údajů. Třída zabývající se notifikacemi bude připravena pro potencionální rozšíření o serverovou část.

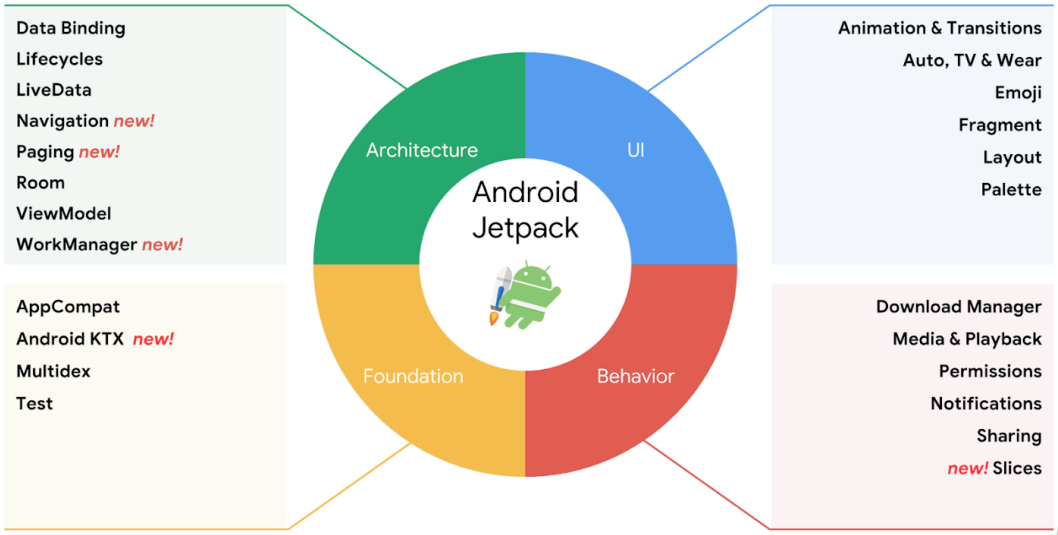
Pro práci s notifikacemi je potřeba dědit *FirebaseMessagingService* a poté přetížit metodu *onMessageReceived()*, která přijme zprávu přicházející ze serveru a upraví ji do výsledné podoby (nastaví titulek, přidá zprávu a ikonu). Poté je notifikace připravena na

finální zobrazení uživateli. K vytvoření notifikace se používá návrhový vzor Stavitel (anglicky Builder), který nastaví vše potřebné díky tečkové anotaci při vytváření objektu. (FIREBASE.GOOGLE, 2019).

3.1.8 Jetpack

Google představil Android Jetpack na veletrhu Google I/O v roce 2018. Jedná se o sadu softwarových komponent, která vývojářům pomáhá vytvářet mobilní aplikace. Jetpack spojuje již existující komponenty Support Library a Architecture Components, které rozděluje do následujících čtyř kategorií, viz obrázek č. 8.
(DEVELOPER.ANDROID/JETPACK, 2019)

Obrázek 8 – Sada softwarových komponent Android Jetpack



Zdroj: https://android.jlelse.eu/what-is-android-jetpack-737095e88161

V kategorii Architecture jsou přítomny moduly, které pomohou vyřešit problém s persistencí dat a životního cyklu aplikace. Nezbytnou částí v programování aplikace pro platformu Android je Fundation, který pokrývá základní funkce systému (běžně tuto funkci zastával AppCompat). Kategorie UI je zaměřená na uživatelské rozhraní, včetně fragmentů a rozvržení grafických komponent na obrazovce. Tato součást není omezena jen pro vývoj aplikací pro mobilní telefony, ale lze ji použít i pro Android TV a Wear OS. (DEVELOPER.ANDROID/JETPACK, 2019)

Aplikace vytvořené pro platformu Android mají specifické chování (zobrazování oprávnění uživateli za určitých okolností, oznámení nebo sdílení), na které jsou uživatelé Androidu zvyklí, přičemž programátoři některá doporučení nedodržují. Kategorie Behaviour se snaží o podobné (ideálně stejné) chování u všech aplikací, tak aby nedocházelo k matení uživatele.

Jetpack je poskytován jako balíček komponent, který není přímou součástí platformy Android. Z tohoto důvodu se může zaintegrovat do projektu jakákoliv komponenta podle potřeby vývojářů, kteří aplikaci spravují. Hlavní myšlenkou celého tohoto projektu od Googlu je především možnost přidat potřebnou funkcionalitu kdykoliv během vyvíjení, nasadit aplikaci do prostředí Google Play Store a poskytnout ji uživatelům v co nejkratším časovém horizontu.

Vzhledem ke skutečnosti, že většina oficiálně vydaných aplikací používá Support Library, není přítomnost zpětné kompability, která poskytuje výhodu nezávislosti na konkrétní verzi, žádným překvapením. Další motivací při vývoji této knihovny bylo podnítit Android programátory, aby své aplikace stavěli na moderních postupech. (DEVELOPER.ANDROID/JETPACK, 2019)

V této diplomové práci budou využity pouze vybrané komponenty z Jetpacku, konkrétně Data Binding, Lifecycles, LiveData, Navigation, ViewModel, Fragment, Layout a Notifications.

3.1.9 Použité knihovny třetích stran

Platforma Android poskytuje vývojářům velké množství vlastních komponent pro vytváření aplikací. Nicméně mohou nastat situace, kdy nejsou oficiálně poskytované komponenty dostačující. Pokud se jedná o malou změnu v chování ve srovnání s již existující komponentou, poté je výhodnější tu současnou pouze změnit. Při velké změně chování je však obvyklé použít knihovny třetích stran, které požadované chování obsahují. Aplikace diplomové práce obsahuje několik knihoven třetích stran, které jsou použity z důvodu zajištění správného chování aplikace. Téměř všechny knihovny (kromě MPAndroidChart) pocházejí od firmy Square, která vytváří open source projekty s licencí Apache 2.0.

První knihovnou od firmy Square použitou v aplikaci je Retrofit. Jedná se o HTTP klienta pro projekty psané v Javě nebo aplikace pro platformu Android. Knihovna umožňuje stahování dat ve formátu JSON nebo XML z webového rozhraní API. Po stáhnutí je objekt konvertován do objektu (POJO), který musí být definovaný pro každé volání.

Pro vykreslení grafů, které jsou nedílnou součástí aplikace diplomové práce, je použita knihovna MPAndroidChart. Hlavní výhoda při použití této knihovny tkví ve variabilitě dostupných grafů a možnosti jejich následného dodatečného upravování. Problémem při použití knihoven vykreslujících obrazce bývá především zatěžování hardwaru mobilního zařízení a s tím související neplynulé používání aplikace. Zvolená knihovna tyto problémy neobsahuje, tudíž nedošlo při implementaci projektu k žádnému neuspokojivému chování nebo zpomalování výsledné aplikace.

Android API sice poskytuje zobrazení kalendáře, avšak pouze jako informativní prvek. Naplánovanou funkcí aplikace diplomové práce byla možnost označit období, které bude následně reprezentováno v grafu s výslednými hodnotami. Pro tento účel byla vybrána externí knihovna TimesSquare, jelikož komponenta zobrazující kalendář existující v Android API má primárně jinou funkci.

3.2 Kryptoměny

Kryptoměny se dají považovat za výsledek třicetileté cesty hledání nezávislého digitálního prostředku pro placení. Digitální měny boří zaběhlé vnímání peněz jako takových, mění se jejich forma a veřejnost se postupně seznamuje s tím, jak s nimi lze vynaložit. Dnešní generace je svědkem digitálního vlastnictví, bez nutnosti zainteresování prostředníka. Kryptoměny usilují o to, aby se staly globální digitální měnou, pro kterou nebudou hranice států důležité.

3.2.1 Druhy kryptoměn a jejich historie

Počet kryptoměn celosvětově stále roste a v současné chvíli existuje již více než 1000 druhů. Největší a nejznámější kryptoměny se liší technologickými aspekty, které jsou popsány níže. V rámci této diplomové práce bylo vybráno celkem šest druhů

kryptoměn, jejichž kurzový vývoj ve svíčkovém grafu bude mít uživatel aplikace k dispozici.

Bitcoin

Za tzv. "vlajkovou loď" kryptoměn se považuje Bitcoin. Síť blockchain vznikla v roce 2009. První zmínka proběhla již v roce 2008 na e-mailové konferenci metzdowd.com, kde bylo zveřejněno osmistránkové pojednání o digitální měně Bitcoin: Peer-to-Peer Electronic Cash Systém. Tento dokument byl zveřejněn přes anonymní účet s pseudonymem Satoshi Nakamoto. (STROUKAL, SKALICKÝ, 2018)

Celkový počet Bitcoin mincí, které lze vytěžit, je stanoven podle algoritmu na 21 000 000 (přesně 20 999 999, 9769). Mince se stahují stabilním, avšak klesajícím tempem z důvodu rostoucí algoritmické složitosti pro vyřešení jedné transakce. Přestože je v současnosti každých 10 minut vytěženo 12,5 nových Bitcoinů, průměrné množství vytěžených Bitcoinů stále klesá. Podle výpočtů se má poslední Bitcoin vytěžit v roce 2140. (STROUKAL, SKALICKÝ, 2018)

Původně byl s Bitcoinem spojován americký občan, Dorian Prentice Satoshi Nakamoto, žijící v Kalifornii, který ovšem všechny spekulace vyvrátil s tím, že se poprvé o existenci Bitcoinů dozvěděl od novinářů Newsweek. V dalších letech se za tvůrce kryptoměny Bitcoin vydával australský podnikatel Craig Wright, který předložil několik důkazů o tom, že právě on je tvůrcem Bitcoinu s pseudonymem Satoshi Nakamoto. Jedním z důkazů potvrzujících jeho tvrzení bylo doložení klíče PGP, o kterém se mnozí domnívali, že patří právě Nakamotovi. Dalším důkazem měl být dokument, který by elektronicky podepsal jako Satoshi Nakamoto. Nicméně Craig Wright dokument podepsat odmítl. Předložené důkazy však nepovažují odborníci za pádné, jelikož je lze vytvořit i dodatečně. (STROUKAL, SKALICKÝ, 2018)

Jméno Nick Szabo bylo odhaleno veřejností díky neutuchající snaze o odhalení osoby, která stojí za jednou z nejzajímavějších technologických inovací za poslední roky. Tato osoba se považuje za zakladatele chytrých kontraktů, protože v roce 1998 vyvinul algoritmus pro digitální měnu Bit Gold. I když se tato kryptoměna nikdy

neprosadila, je i tak považována za základ pro Bitcoin. Nick Szabo však svůj pracovní podíl na vývoji Bitcoinu popřel.

Litecoin

Po výše zmíněném Bitcoinu je nejvíce známá kryptoměna Litecoin. Poprvé ji představil 7. října 2011 její tvůrce Charlie Lee (s přezdívkou Satoshilite), bývalý zaměstnanec společnosti Google. Litecoin je přímým potomkem Bitcoinu, avšak hlavními rysy této kryptoměny jsou použití algoritmu scrypt a rychlejší provedení transakcí. Díky algoritmu scrypt je možné těžit Litecoiny na méně výkonných strojích, protože hlavní výpočetní náročnost vyžaduje paměť stroje. (LITECOIN, 2018)

Celkový počet Litecoin mincí je 84 milionů, což je čtyřnásobně větší počet, než dovoluje Bitcoin. Velkou výhodou je také kratší časová prodleva vyhodnocování transakcí, která je v případě Litecoinu až čtyřikrát rychlejší, což dělá z této kryptoměny vhodnější platidlo pro každodenní život. S tím souvisí i odměny za proběhlé transakce, které nejsou z důvodu menší náročnosti tak vysoké. (LITECOIN, 2018)

Ethereum

Digitální měnou budoucnosti je dle spekulací kryptoměna nazývaná Ethereum (nebo též Bitcoin 2.0). Tvůrci Etherea Vitalik Buterin a Gavin Wood poprvé koncept představili v roce 2013. Koncept zahrnuje virtuální stroj Ethereum Virtual Machine (EVM) pro fungování chytrých kontraktů, který se snaží zajistit co možná nejméně komplikací při fungování sítě. Následující rok v projektu pokračoval s názvem Yellow Paper. První oficiální spuštění sítě proběhlo 30. července 2015 v režimu, který nesl název Frontier (veřejně dostupný beta testing). (EDDISON, 2017)

Největší rozdíl Etherea od ostatních kryptoměn je v síti samotné (tzv. blockchain ethereum). Jde o decentralizovaný systém pro uchovávání a práci s výše zmíněnými chytrými kontrakty. Smlouvy v chytrých kontraktech nelze podepsat neoprávněnou autoritou, jelikož jsou bloky kódu naprogramované tak, aby byly ověřované celou sítí. To dělá z Etherea a Blockchain Etherea nejpoužívanější technologii u společností pro primární úpis akcií. Bloky kódu (smlouvy) lze za běhu upravit a změnit tak viditelné informace o investorech v účetní knize. (EDDISON, 2017)

Bitcoin Cash

Bitcoin se začal potýkat s problémy způsobenými přehlcením sítě při provádění transakcí. V dnešní době trvá transakce i několik hodin (pokud je trh stabilní) a vzhledem ke stoupající náročnosti dochází k prodloužení intervalu až na řády dnů. Jedním z řešení, se kterou souhlasila valná většina bitcoinové komunity, by bylo použití technologie „Segregated Witness“ (označována též SegWit2x), která snižuje objem dat v bloku odstraněním signature data a přidáním dat do doprovodného bloku. Někteří odpůrci však tuto technologie nepovažovali za přijatelné řešení, jelikož SegWit2x nebyla dostatečně transparentní a její použití by vedlo k porušení decentralizace a narušení demokratického charakteru měny. (BITCOINCASH, 2018)

Tyto konflikty a nesouhlas s většinovou politikou vyústily v rozhodnutí Bitcoin 1.srpna 2017 „rozvětvit“ (originálně pojmenováno fork) a vytvořit Bitcoin Cash, který založil novou větev celé blockchain databáze. Vzhledem k přímému „rozvětvení“ má Bitcoin Cash podobné vlastnosti jako Bitcoin. Maximální množství mincí je 21 milionů.

Monero

V současnosti je za jednu z nejvíce anonymních považována měna Monero. Na trhu jsou k dispozici i jiné anonymní měny, ale jejich anonymita je pouze volitelnou možností. Monero neumožňuje tuto funkci vypnout a ve výchozím stavu skrývá většinu informací (adresy odesílatele a příjemce, hodnotu proběhlé transakce apod.). Velký rozdíl je i v používání klíčů k peněžence. Zatímco např. u Bitcoinu je potřeba pouze veřejný a privátní klíč, u Monera jsou zapotřebí soukromý view key, veřejný view key, soukromý spend key a veřejný spend key. (GETMONERO, 2018)

Za touto měnou stojí sedm vývojářů, z nichž však pouze dva, David Latapie a Riccarda Spagni, odhalili svoji identitu. Monero bylo poprvé oficiálně představeno 18. dubna 2014. Vývojáři Monero založili na CryptoNote protokolu, které má zásadní algoritmické vlastnosti, které je potřeba zohlednit k úplné anonymitě v platební síti. (GETMONERO, 2018)

Počet mincí v síti Monero nemá maximální hodnotu, nicméně odměny se stále zmenšují. Během roku 2022 bude v oběhu 18 132 000 mincí, přičemž odměna za jeden vytěžený blok nepřesáhne hodnotu 0,6 Monero. Každé dvě minuty je vytěžen nový blok. (GETMONERO, 2018)

Dogecoin

Hlavním důvodem vzniku Dogecoinu nebylo zlepšení současných technologií jako u ostatních kryptoměn. Tato kryptoměna vznikla pouze jako recese. Měnu vytvořil programátor Bill Markus a marketingový specialista Jackson Palmer v prosinci roku 2013. Dogecoin vychází z Litecoinu, tudíž obsahuje algoritmus scrypt a dokáže zpracovat desetkrát více transakcí než výše zmíněný Bitcoin. Dogecoin nemá žádný limit pro maximální počet mincí, na rozdíl od Litecoinu. (THOMPSON, 2015)

3.2.2 Vlastnosti

Kryptoměny jsou i přes kolísavý vývoj kurzu stále populárnější. Hlavní zásluhu na vzrůstající popularitě a rostoucí komunitě mají především jejich unikátní vlastnosti, které běžné platební metody (peníze, šeky nebo platební karty) nemají. Většinu z níže popsaných vlastností mají téměř všechny veřejně známé kryptoměny. Několik vybraných hlavních vlastností kryptoměn je představeno blíže.

Jedna z nejvíce diskutovaných vlastností kryptoměn je decentralizace. Kryptoměny nemají žádnou státní autoritu nebo centrální bod, tudíž je nelze kontrolovat. Všechny uzly v síti Blockchain jsou si rovny, což znamená, že i když přestane pracovat libovolná část uzlů, ostatní uzly nebudou ovlivněny. (KALISKÝ, 2018)

Všechny transakce jsou naprosto transparentní. Každý uzel obsahuje informace o provedených transakcí. Jelikož má každý uzel v průměru informaci o 500 transakcích, je možné vyhledat i jednotlivé transakce. (KALISKÝ, 2018)

Další důležitou vlastností je samostatnost sítě. Odměny za transakce získají pouze ti, kteří vypočítají vzniklé transakce, tudíž nejsou potřeba žádné organizace. Výše

odměn lze s různými transakcemi měnit, ale čím vyšší je odměna, tím větší je pravděpodobnost, že bude daná transakce vyřešena rychleji. (KALISKÝ, 2018)

Anonymita hraje ve světě kryptoměn velkou roli a tato jejich vlastnost je jedním z důvodů, proč kryptoměny původně vznikly. Dříve se dala považovat téměř každá měna za anonymní. Nicméně v některých případech je potřeba se legitimovat dokladem (občanský průkaz, řidičský průkaz apod.), především při koupi většího počtu kryptoměn nebo při registraci na burzách a dalších situacích, při kterých již platí určitá pravidla. Přestože není další obchodování přímo spjaté s osobními údaji, transakce i nadále probíhají na uživatelově účtu, kde jsou tyto informace přiřazené. V dnešní době se za nejvíce anonymní měnu dá považovat Monero nebo Anoncoin.

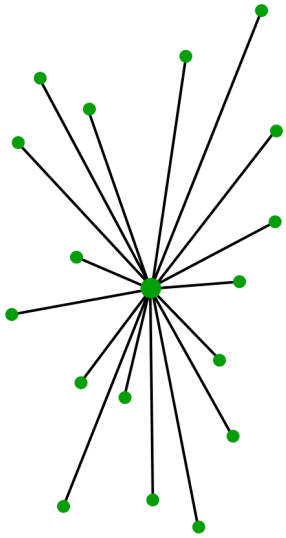
3.2.3 Blockchain

Hlavní technologií kryptoměn je blockchain, který slouží jako "speciální" druh databáze. Mezi informatiky se databáze popisuje jako prostor, kde jsou uloženy informace z obchodů o produktech, komentáře a fotografie nebo jednoduše informace o uživateli a jeho pohybu díky zapnuté lokaci v telefonu.

V současnosti se běžně používají centralizované databáze, což v praxi znamená, že existuje jedno úložiště s velkým množstvím pevných disků a zajištěním stabilním rychlým internetovým připojením, kde se pracuje se všemi daty dané společnosti.

Dalším obvyklým postupem bývá pronájmem záložních serverů pro případ napadení nebo výpadku elektrického proudu (zákazník je jednoduše přesměrován na jiný server), přesto ani tímto způsobem není možné problém s centralizovanou databází vyřešit. Další možné problémy spjaté s centralizovanými databázemi souvisí s možností jejich zneužití. Může se jednat jak o nekalé praktiky prováděné menší skupinou lidí přímo na databázi, tak i o prodej nasbíraných dat o uživatelích jiným organizacím nebo sledování uživatelů díky získaným datům. Možností zneužití je mnoho a riziko je velmi vysoké. (DRESCHER, 2017). Příklad centralizované sítě je možné vidět na obrázku č. 9 níže.

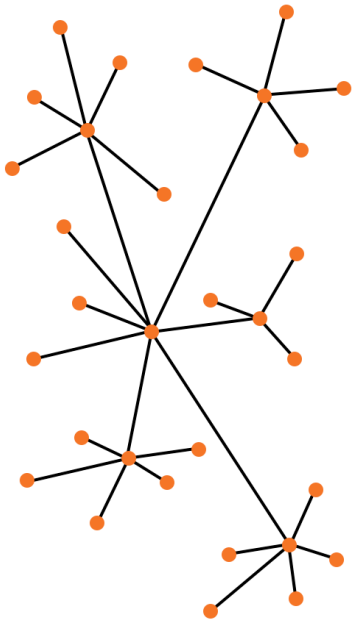
Obrázek 9 – Centralizovaná databáze



Zdroj: vlastní zpracování

Výše uvedené důvody byly hlavní motivací pro vytvoření decentralizované databáze (obrázek č. 10 níže) neboli Blockchainu pro kryptoměny. Jedná se o typ distribuované databáze, která je soběstačná a neexistuje v ní slabý bod, který by se dal zneužít. Databáze není nikým ovládaná, tudíž zde není prostor pro útok. Ani pronájem externích serverů pro správu sítě není nutný, jelikož je síť tvořena miliony osobními počítači po celém světě, které vlastní obyčejní lidé, tzv. „mineři“, kteří vypočítávají vzniklé transakce. (DRESCHER, 2017)

Obrázek 10 – Decentralizovaná databáze



Zdroj: vlastní zpracování

V síti Blockchain je bezpečnost prioritou, proto jsou veškerá data šifrována hashovacím algoritmem. Součástí sítě se může stát každý, ale musí poskytnout část výpočetního výkonu síti. Poté už lze „těžit“ tím, že uživatelův počítač bude náhodně zkoušet různé hashe a pokud těžař jako první použije správný hash, dostane odměnu. (DRESCHER, 2017)

3.2.4 Transakce v síti Blockchain

Důležitou roli v transakcích hrají veřejné a privátní klíče. Zatímco privátní klíč umožnuje přístup ke kryptoměně uložené v peněžence, veřejný klíč nám slouží jako cílová destinace, kam mají být prostředky přesunuty. Na začátku transakce je zvolena částka k poslání a pomocí privátního klíče je proveden digitální podpis, do kterého se zapíší všechny detaily o transakci (adresa příjemce a množství kryptoměn k odeslání). Tímto je transakce připravena k odeslání. V této chvíli je transakce již v progresu a uživatel musí počkat na její vyřešení. Jako důkaz, že operace není falešná, je nutné přiložení veřejného klíče k transakci, který slouží k ověření digitálního podpisu. Jakmile vše proběhne v pořádku, transakce je kompletní a měna se přesune např. z peněženky do peněženky. Transakce neprobíhají pouze mezi pěněženkami, ale například i na burze a podobně. (DRESCHER, 2017)

4 Vlastní práce

4.1 Vzhled aplikace

V roce 2014 vydala firma Google operační systém Android Lollipop (Android 5.0), který jako první nesl prvky Material Designu. Pro intuitivní ovládání a všeobecnou oblíbenost tohoto vzhledu se společnost rozhodla během dalších let graficky sjednotit své další aplikace (webové i mobilní). V dnešní době jsou v Material Designu všechny uživatelem běžně používané aplikace (Google Maps, Gmail, YouTube, Google Disk, Překladač a mnoho dalších).

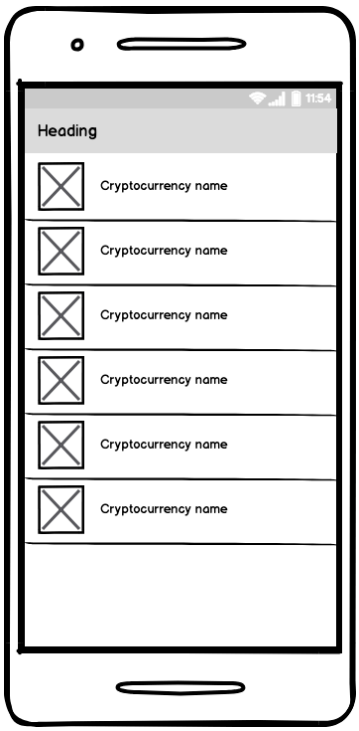
Aplikace, která vznikla pro účely této diplomové práce taktéž nese grafické prvky Material designu pro snazší orientaci a adaptaci pro potencionální uživatele, kteří jsou na tento vzhled již zvyklí ze svého operačního systému. K tomu, aby byla aplikace úspěšná je důležité podchytit i ty nejmenší detaily. Pro uživatele je vždy prvních pár desítek vteřin po spuštění mobilní aplikace rozhodujících. Pokud je uživatel z aplikace zmatený nebo není na první pohled jasné, jak aplikace funguje, obvykle tato zkušenost končí odinstalováním aplikace. Vzhled aplikace je tedy stejně důležitý jako správně naimplementovaný program.

4.1.1 Drátové modely

První drátový model představuje úvodní obrazovku (třída *CryptoListFragment*), která se uživateli zobrazí po spuštění aplikace. Hlavní funkcí této obrazovky je, v rámci této konkrétní aplikace, vybrání jedné z šesti dostupných kryptoměn (Bitcoin, Litecoin, Bitcoin Cash, Ethereum, Monero a Dogecoin), jejíž aktuální kurz si chce uživatel prohlédnout. Taktéž si může uživatel nastavit hlídání konkrétní výše kurzu, při jejímž dosažení aplikace zašle uživateli upozornění.

Záhlaví je v mobilních aplikacích důležitým faktorem a ani v rámci této diplomové práce tomu není jinak. Funkcí záhlaví v této aplikaci je především zobrazení názvu aktuální obrazovky, což slouží pro přehlednější průchod aplikací. Jednotlivé položky v seznamu obsahují název dané kryptoměny a obrázek reprezentující její logo (obrázek č. 11 níže). Po kliknutí na položku je uživatel přesměrován na následující obrazovku, která obsahuje podrobnější informace.

Obrázek 11 – Drátový model úvodní obrazovky

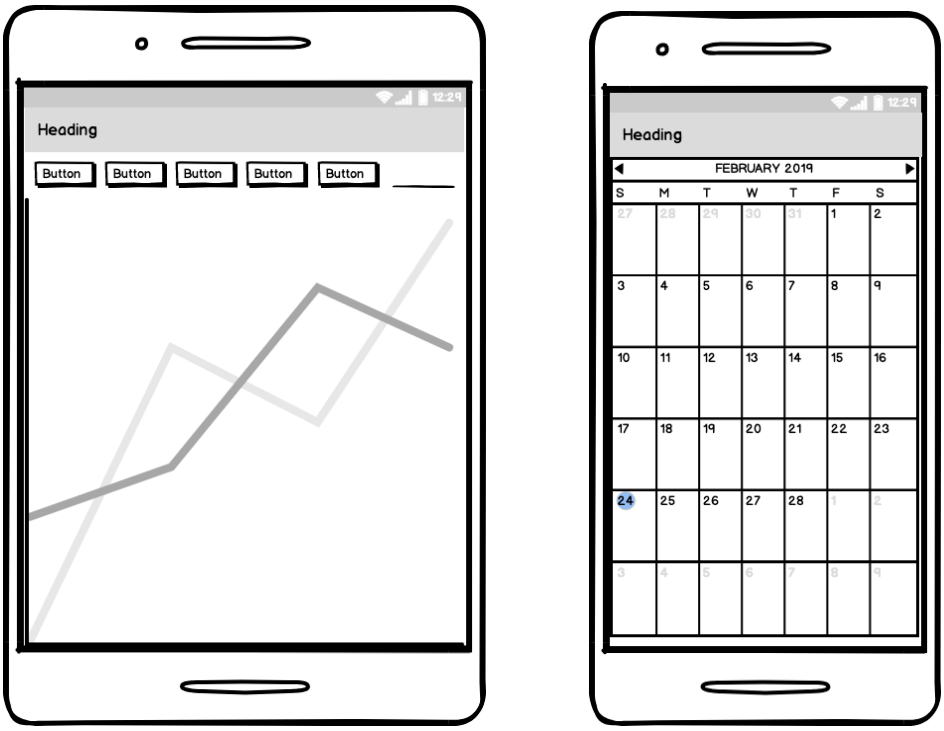


Zdroj: vlastní zpracování

Po vybrání konkrétní kryptoměny je k dispozici obrazovka (třída *CryptoInformationFragment*), ve které se nachází svíčkový graf s informací o aktuálním vývoji kryptoměny. Uživatel má možnost si zvolit období, ve kterém chce vývoj kurzu zobrazit. K dispozici je možnost zvolení defaultního režimu (posledních 540 dní), 24 hodin, 48 hodin a volitelného časového období. Funkce volitelného období umožnuje uživateli vybrat si manuálně časový horizont pomocí kalendáře zobrazeného níže na obrázku č. 12. Není však možné zvolit datum starší než 540 dní (defaultní nastavení).

Poslední možností je zvolení konkrétní výše kurzu, při jejímž dosažení se uživateli zobrazí notifikace.

Obrázek 12 – Drátový model grafu a funkce volitelného období



Zdroj: vlastní zpracování

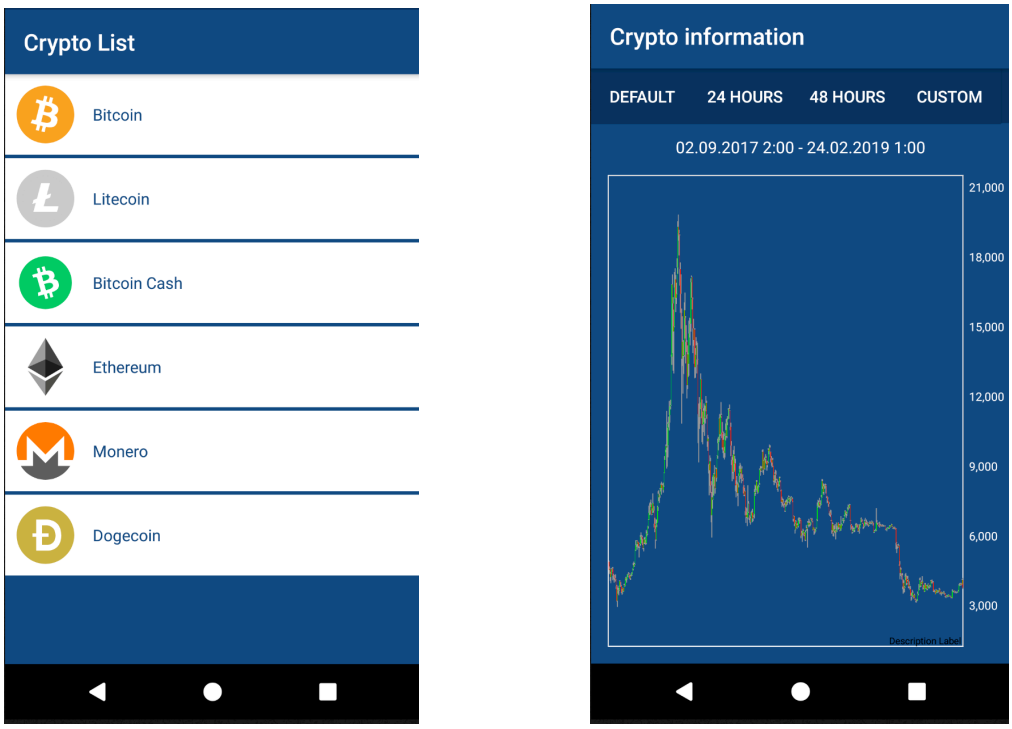
4.1.2 Finální vzhled aplikace

Finální vzhled aplikace vyvíjené v rámci této diplomové práce reprezentuje obrázek č. 13 níže. Pro demonstraci funkčnosti aplikace jsou na snímcích použita reálná data ze serveru třetí strany.

Pro větší přehlednost při používání aplikace jsou mezi vybranými kryptoměnami vloženy jednotlivé separátory. Již na první pohled je patrné, kam uživatel může kliknout, aby mohl přejít na další obrazovku.

Při větším množství dat (např. druhý screenshot), lze graf libovolně přibližovat a následně zjistit přesnou cenu v určitém období. Osa Y reprezentuje cenu kryptoměny, za kterou se v daném období prodávala/kupovala, zatímco osa X je představitelem časového období. Pokud uživatel při používání aplikace mobilní telefon přetočí, obrazovka se přizpůsobí horizontální orientaci a veškerá data se zobrazí opět napříč celou obrazovkou.

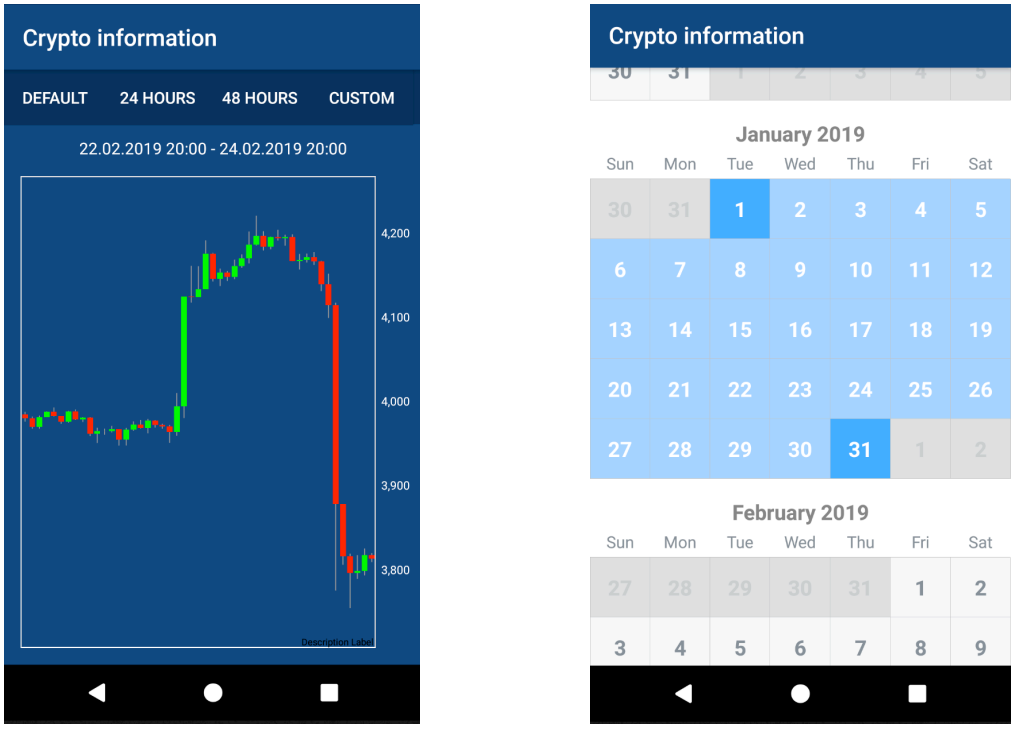
Obrázek 13 – Finální vzhled aplikace



Zdroj: vlastní zpracování

Obrázkem č. 14 je další snímek, který reprezentuje zobrazení dat v rámci 48 hodin. Na posledním snímku je vyobrazen kalendář, díky kterému si potencionální uživatel může navolit přesné datové rozmezí.

Obrázek 14 – Zobrazení dat v aplikaci v rámci 48 hodin



Zdroj: vlastní zpracování

4.2 Konfigurace buildovacího systému Gradle

Vývoj aplikací pro platformu Android je také o správném nastavení buildovacího systému. Gradle umožňuje programátorům automatizovat procesy např. při vytváření release verze pro konzoli Google Play, zprovoznění speciálně naimplementovaného kódu pro testování zvolené komponenty nebo volbě verze, při jaké se aktivuje ProGuard a mnoho dalších situací.

Na začátku každého Gradle souboru musí být definovány doplňky (pluginy), které budou použity v daném modulu. Existuje ovšem i několik výjimek, kdy musí být daný doplněk až na úplném konci souboru. V diplomovém práci se jedná o *com.google.gms.google-services*, který bude popsán v další části. Doplněk *com.android.application* je přítomen ve všech aplikacích. Dalšími doplňky použitými v aplikaci jsou *kotlin-android* a *kotlin-android-extensions*, které se do aplikací přidávají pouze v případě, kdy chce vývojář vyvíjet jak v Javě, tak i v programovacím jazyku Kotlin, viz obrázek č. 15. (DEVELOPER.ANDROID/GRADLE-TIPS, 2018)

Obrázek 15 – Doplňky v Gradle souboru

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

Zdroj: vlastní zpracování

Další na řadě je v Gradle souboru Android blok. Správné nastavení je velice důležité pro další fungování aplikace, pro automatizaci inkrementálního zvětšování verze pro publikaci na Google Play a mnoho dalších funkcí.

První nastavenou informací v aplikaci je *compileSdkVersion*, která je v diplomové práci nastavená na hodnotu 28. Jedná se o verzi Android SDK, který je pro IDE (nebo jiný prostředek pro kompilaci) použit k běhu, vyvíjení nebo pro vytvoření aplikace a finálního .apk souboru. Obvykle má stejnou hodnotu nastavený i *targetSdkVersion.* V rámci diplomové práce je hodnota stejná, ale může se i lišit, jelikož se nejedná o žádné dané pravidlo. Hodnota *targetSdkVersion* značí verzi Androidu SDK, na který je aplikace cílena a tím zajišťuje optimální běh aplikace.
(DEVELOPER.ANDROID/GRADLE-TIPS, 2018)

Tímto nastavením se neomezuje chod aplikace na předchozích verzích systému Android. Hodnota *minSdkVersion* označuje nejnižší verzi sady Android SDK, na které může být aplikace spuštěna. Dané ohraničení řeší problémy vzniklé s dřívějším rozhraním API, chybějícími funkcemi nebo jinými problémy s chováním předchozí verze. Nejnižším možným SDK, na které je možno aplikaci nainstalovat, je SDK verze 24, tedy Android 7.0 Nougat. (DEVELOPER.ANDROID/GRADLE-TIPS, 2018)

Následující hodnota, kterou je potřeba vyplnit, je *versionName*. Jedná se o název, jehož jediným účelem je zobrazení uživatelům. Tento atribut lze nastavit jako stringový řetězec. Naprosto odlišnou funkcionalitu zastává atribut *versionCode*, který obsahuje číslo verze používané interně. Jediným pravidlem při jeho použití je inkrementálně zvyšovat danou verzi, aby mohla např. developerská konzole Google Play poznat, že se jedná o novou verzi aplikace (první verze může začínat na čísle 255, následující verze musí však být 256). V diplomové práci jsou oba atributy nastaveny na 1 (respektive

u *versionName* je hodnota nastavena na 1.0), což symbolizuje první a také jedinou vydanou verzi aplikace (vytvořenou v rámci této diplomové práce).
(DEVELOPER.ANDROID/GRADLE-TIPS, 2018)

AndroidJUnitRunner je nástroj pocházející z testovací knihovny Android Support, který spouští JUnit testy systému Android. Pokud vývojář chce testovat logiku dané funkcionality, je potřeba přidat argument typu *testInstrumentationRunner* s informací v jakém adresáři je obsažen. Jedná se o adresář obsažený v Android knihovnách, tudíž při používání Jetpack a AndroidX bude umístění v adresáři stejná, jako je na obrázku č. 16 níže.

Každá aplikace operačního systému Android má jedinečné ID aplikace (má stejný tvar jako název java balíku např. cz.myapp.application). Toto označení jednoznačně identifikuje každou aplikaci v obchodě Google Play. Pokud chce vývojář nahrát novou verzi aplikace, musí se *applicationId* (a cerifikát, se kterým je aplikace podepsaná) shodovat s původně nahranou verzí APK. Pokud je *applicationId* rozdílné, obchod Google Play považuje aplikaci za zcela jinou a nepřiřadí si nově nahrané APK již k původní aplikaci. (DEVELOPER.ANDROID/GRADLE-TIPS, 2018)

Při založení nového projektu v Android Studiu se *applicationId* přesně shoduje s názvem balíku, který byl zvolen při inicializaci projektu, přičemž jsou další názvy naprosto nezávislé a mohou se měnit. Při přejmenování ID aplikace je nutno dodržet restriktivní pravidla. První zmíněné omezení je nutnost mít alespoň dva segmenty (mít mezi slovy alespoň jednou tečku např. cz.app). Další pravidlem je, že počáteční znak každého segmentu musí začínat písmenem. Závěrečným pravidlem od firmy Google je omezení povolených znaků na alfanumerické nebo podtržítka (zapsáno v regulárním výrazu [a-zA-Z0-9_]). (DEVELOPER.ANDROID/GRADLE-TIPS, 2018)

Obrázek 16 – Konfigurace Android aplikace

```
android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "cz.reneuhrin.masterdegree"
        minSdkVersion 24
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
```

Zdroj: vlastní zpracování

Parametry aplikace pro vyvíjení nebo finální verze pro uživatele by měla být z hlediska nastavení co možná nejpodobnější. Díky podobnému nastavení při vyvoji aplikace, lze odhalit jinak těžko reprodukovatelné chyby, které je dobré minimalizovat ještě před vydáním do produkčního prostředí. Nicméně se najde i nastavení, které není při vývoji potřeba nebo může dokonce stížit programátorům jejich práci. V aplikaci diplomové práce je ve verzi pro vyvíjení vypnut ProGuard (obrázek č. 17), který slouží mimo jiné i pro obfuskaci kódu. (DEVELOPER.ANDROID/GRADLE-PLUGIN, 2018)

Obrázek 17 – Vytvořené buildovací typy debug a release

```
buildTypes {
    debug {
        minifyEnabled false
        useProguard false
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.txt'
    }
    release {
        minifyEnabled false
        useProguard true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                'proguard-rules.pro'
    }
}
```

Zdroj: vlastní zpracování

Dříve bylo pro Android vývojáře běžné používat pro vývoj Javu 6 (nebo předchozí verze Javy), která již nesplňovala moderní prvky programování. Z tohoto důvodu byla posléze vytvořena a využívána tzv. Retrolambda, sloužící alespoň pro částečnou minimalizaci vznikajícího kódu.

Společnost Google se snaží neustále naslouchat požadavkům Android komunity a proto během roku 2017 vydala ve stabilní verzi vývojové prostředí Android Studio

3.0, která s sebou přinesla mnoho nových funkcí a možností. Mezi těmito novými funkcemi byla i možnost používat Javu ve verzi 8, která již nabízí nativní použití lambda funkcí. Tato funkce však není defaultně nastavena a musí být explicitně přidána v *compileOptions* pomocí *sourceCompatibility* a *targetCombatibility* s definicí, o jakou verzi se jedná (v tomto případě *JavaVersion.VERSION_1_8*), viz obrazek č. 18. (DEVELOPER.ANDROID/GRADLE-PLUGIN, 2018)

Obrázek 18 – Přidání Javy 8 do projektu

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

Zdroj: vlastní zpracování

Unit testy nejenže pracují s konkrétní logikou dané problematiky (podrobněji rozebráno v další kapitole diplomové práce), ale mohou i vyvolat výjimky v rozhraní API ze sady Android SDK. To může být problematické pro testy a vyústit ve zkreslení testovací funkcionality. Důvodem vzniku vyjímek je nenamockování metody ve třídě, která není inicializovaná, protože používání unit testu neobsahuje žádný skutečný kód (API ze sady Android SDK je poskytnuta pouze systémovým obrazem na konkrétním zařízení). Preventivní namockování potencionálně využité metody není vhodným postupem a proto je v těchto situacích lepší vrátit defaultní hodnoty. Tohoto je možné docílit přidáním bloku kódu *testOptions*, jak je zobrazeno na obrázku č. 19 níže.

Nicméně nastavení defaultních hodnot při testování s sebou nese i určitá rizika. Hodnoty s návratovou hodnotou nula nebo null mohou v testech zavést regrese, které se obtížně opravují či způsobit selhání testů. Proto je opravdu důležité, aby vývojář zvážil, jaké výše zmíněné nastavení je v konkrétní situaci nejvhodnější.
(DEVELOPER.ANDROID/GRADLE-PLUGIN, 2018)

Obrázek 19 – Nastavení defaultních hodnot v rámci unit testů

```
testOptions {
    unitTests.returnDefaultValues = true
}
```

Zdroj: vlastní zpracování

Jak už bylo v diplomové práci zmíněno, existují i doplňky, které je potřeba přidat právě na konec Gradle souboru. Hlavním důvodem, proč je tento doplněk potřeba přidat až po zanalyzování ostatních závislostí v souboru, je zásuvný modul, který přidá jádro knihovny Firebase, pokud již není v aktuálním projektu. Také zkontroluje verzi závislostí Firebase a Google Play Services. Aby však mohla být všechna tato práce provedena bez konfliktu s ostatními doplňky, musí být *com.google.gms.google-services* (obrázek č. 20) spuštěn proti projektu až poté, co budou ostatní závislosti projektu již nadefinovány. Přidáním doplňku na konec souboru se lze těmto problémům vyhnout a řádně synchronizovat závislosti mezi sebou.

Obrázek 20 – přidání doplňku google-services v buildovacím skriptu

```
// Add to the bottom of the file
apply plugin: 'com.google.gms.google-services'
```

Zdroj: vlastní zpracování

4.3 Výběr REST API pro konzumaci dat

Mezi plánované funkce mobilní aplikace vyvíjené v rámci této diplomovou práce patří komunikace se serverem třetí. Dalším krokem při vývoji bude výběr vhodného REST API, které splňuje předem dané požadavky.

První stanovenou podmínkou, která značným způsobem omezuje výběr dostupných služeb při finálním rozhodování, je počet bezplatných volání z klienta na server. Tento počet volání je omezován záměrně, aby měli vývojáři či firmy větší motivaci ke koupi a jejich následného používání i pro komerční účely.

Další vlastností API, která je důležitá při finálním výběru, je možnost získat historická data předem vybraných kryptoměn, zmíněných výše, alespoň za poslední dva roky. Neméně důležitá je i možnost pracovat s daty nejen v intervalu dní či hodin, ale i minut, pokud to API umožňuje.

Pro dostatečně deskriptivní popis zpracovávaného kurzu a spolehlivější předpověď jeho budoucího směru růstu je v aplikaci použit svíčkový graf. Pro korektní vykreslení grafu je nutné znát čtyři hodnoty. Jedná se o cenu při otevření trhu, cenu při zavření trhu, horní a spodní stín (anglický překlad pro tyto hodnoty je open, close, high

a low). Poslední podmínkou při výběru vhodné služby pro konzumaci dat je možnost mít všechna dostupná data ve více fiat měnách.

V tabulce č. 1 je znázorněno porovnání mezi jednotlivými poskytovateli, ze kterých byl nakonec vybrán ten finální.

Tabulka 1 – Porovnání poskytovatelů REST API

Poskytovatel	Použití zdarma množství/měsíc	Historická data	Intervaly v rámci dnů a hodin (minuty)	Obsahující hodnoty (high, low, open, close)	Více fiat měn
www.coinapi.io	Ano – 100	Ano	Ne	Ne	Ne
min-api.cryptocompare.com	Ano – 100000	Ano	Ano	Ano	Ano
docs.nomics.com	Ano – bez limitu	Ano	Ne	Ano	Ano
coinmarketcap.com	Ano - 10000	Ano	Ano	Ano	Ano

Zdroj: vlastní zpracování

Zelená barva reprezentuje, že daný poskytovatel konkrétní funkci podporuje a červená vyjadřuje opak. První poskytovatel v tabulce (www.coinapi.io) umožňuje využívat REST API bezplatně, avšak pouze prvních sto zavolání serveru, což nemusí být pro vývoj aplikace dostatečné. Zároveň neposkytuje hodinové intervaly, potřebná data pro svíčkový graf a ani přepočet na jiné fiat měny. Další poskytovatel (docs.nomics.com) jako jediný podporuje komunikaci se serverem, která je při volném klíči bez limitu, což neposkytuje žádný jiný z uvedených možností. Celkově se poskytovatel docs.nomics.com zdá být lepší než www.coinapi.io, nicméně ani zde není k dispozici intervalový výběr. Finální volba probíhala mezi poskytovateli min-api.cryptocompare.com a coinmarketcap.com, přičemž oba poskytovatelé splňují předem dané atributy pro používání. Pro účely této diplomové práce byl však nakonec vybrán poskytovatel min-api.cryptocompare.com, jelikož povoluje desetkrát více volných volání na server měsíčně než poskytovatel coinmarketcap.com.

4.4 Schéma architektury

Správné navržení architektury je základním stavebním kamenem pro vytvoření škálovatelné mobilní aplikace a minimalizace opakování stejného kódu. Jasně

definovaná pravidla architektury jsou důležitá zejména v projektech, na kterých spolupracuje více programátorů nebo dokonce i různé týmy.

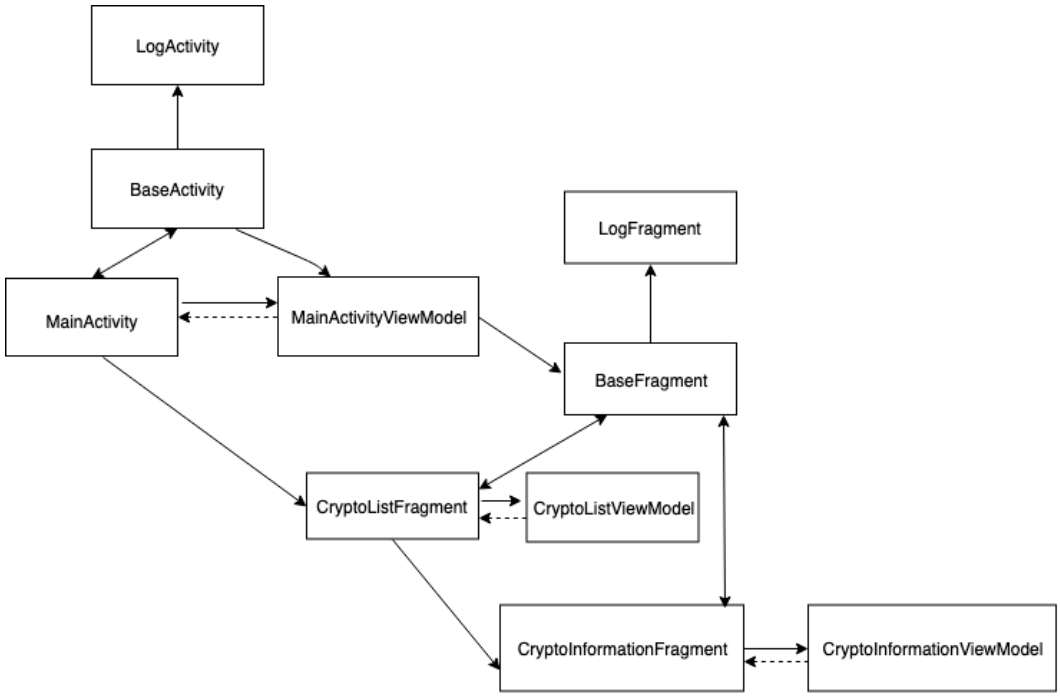
Současná architektura aplikace v rámci této diplomové práce je rozdělena do čtyř hlavních segmentů. Jedná se o logovací vrstvu, předpřipravenou strukturu pro aktivity a fragmenty, obrazovky pro uživatele (aktivity a fragmenty) a jejich jednotlivé ViewModely. Jednotlivé třídy budou popsány v další části diplomové práce.

Jedním z výše uvedených hlavních segmentů je logovací vrstva, která slouží především pro vývoj aplikace nebo pro odstraňování chyb. V této vrstvě je zachycen celý životní cyklus aplikace do logcatu.

Při vytváření nových aktivit a fragmentů v aplikaci dochází k repetitivní činnosti. Neustále se opakující bloky kódu stojí vývojáře spoustu času, který by bylo vhodnější investovat do vývoje funkcionality aplikace nebo opravě chyb. Tento problém byl motivací pro vytvoření další části architektury, která obsahuje abstraktní třídy, které používají generické vytváření aktivit a fragmentů, ViewModelů a prostředí pro data binding připravené pro konstrukci aktivit a fragmentů.

Aktivity v aplikaci této diplomové práce splňují roli hlavního rámce pro skupinu fragmentů, který se nejprve objeví hned po spuštění aplikace a poté po vybrání dané kryptoměny. V rámci budoucího rozšíření mohou být však vytvořeny i nové rámce s novými fragmenty. Každá aktivita má svůj ViewModel, který slouží pro přeposílání dat mezi fragmenty. Pokud je informace hodna přesunu, je nutné si v daném fragmentu zažádat o ViewModel dané aktivity a uložit informaci do proměnné. Na straně druhé, pokud je informaci potřeba využít, stačí opět zažádat o ViewModel a pracovat s konkrétními daty. Mezi opravdu důležité vazby ve schématu architektury patří tedy komunikace mezi ViewModelem dané aktivity a *BaseFragmentem* (obrázek č. 21). Posílání dat mezi různými aktivitami není předmětem této diplomové práce, nicméně je velice snadné tuto funkcionalitu naimplementovat a následně využívat při současném navržení architektury.

Obrázek 21 – Schéma navržené architektury



Zdroj: vlastní zpracování

Na schématu je zobrazena komunikace mezi jednotlivými komponentami. V *CryptoListViewModel* je znázorněna generická komponenta vytvořená pro grafický element *RecyclerView*. Třída *GeneralRecyclerViewAdapter* je vytvořena generickým způsobem, tudíž ji vývojáři mohou použít na více seznamech v aplikaci. V programovacím jazyku Kotlin se generické programování v zápisu od Javy liší. Je nezbytné zvolit generický typ určený pro vnitřní a vnější zpracování (označeno slovy in a out). Důležité metody pro práci s adaptéry a ViewHoldery jsou přepsány z předka a tudíž je lze využít na místě použití.

Celá operace začíná metodou *onCreateViewHolder(),* ve které se definuje vzhled jednotlivých položek ze seznamu. Aby byl vytvořen grafický vzhled dané položky, je nutné definovat nadřazený element společně s informací, zda je potřeba připojit jednotlivé rozložení k sobě. *RecyclerView* v Android API nemá nativní podporu pro zjištění interakce s položkou v seznamu (kliknutí uživatele). Pro tento případ je vytvořeno Rozhraní, které informuje o vzniklé aktivitě. Také je zaslán objekt reprezentující daný řádek, se kterým se nadále pracuje. Dalším rozhraním vytvořeným ve třídě je Binder. Zde je použita genericita (zmíněná výše) pro explicitní označení

objektu používaného přímo ve třídě *GeneralRecyclerViewAdapter*, nebo objektu zasílaného vývojáři pro další operaci. Třída obsahuje abstraktní metody pro inicializaci *ViewHolderu* a referenci na XML soubor reprezentující vzhled položky.

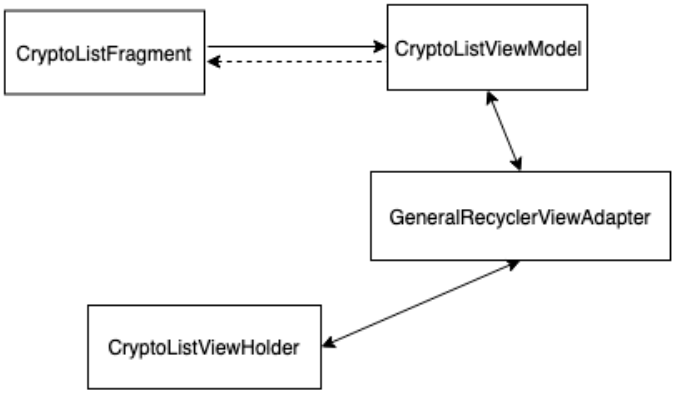
Zdrojový kód daného *GeneralRecyclerViewAdapteru* je zobrazen na obrázku č. 22. Navrhnuté schéma pro demonstrování komunikace mezi komponentami je uvedeno níže na obrázku č. 23.

Obrázek 22 – Zdrojový kód třídy *GeneralRecyclerViewAdapter*

```
override fun onCreateViewHolder
            (parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder {
    return getViewHolder(
        LayoutInflater.from(parent.context)
            .inflate(viewType, parent,  attachToRoot: false), viewType)
}

override fun onBindViewHolder
            (holder: RecyclerView.ViewHolder, position: Int) {
    (holder as Binder<T, T>).bind(listItems[position], clickListener)
}

override fun getItemCount(): Int {
    return listItems.size
}

override fun getItemViewType(position: Int): Int {
    return getLayoutId(position, listItems[position])
}

protected abstract fun getLayoutId(position: Int, item: T): Int

abstract fun getViewHolder
            (view: View, viewType: Int):RecyclerView.ViewHolder

internal interface Binder<in T, out U> {
    fun bind(data: T, clickListener: ItemClickListener<U>)
}
```

Zdroj: vlastní zpracování

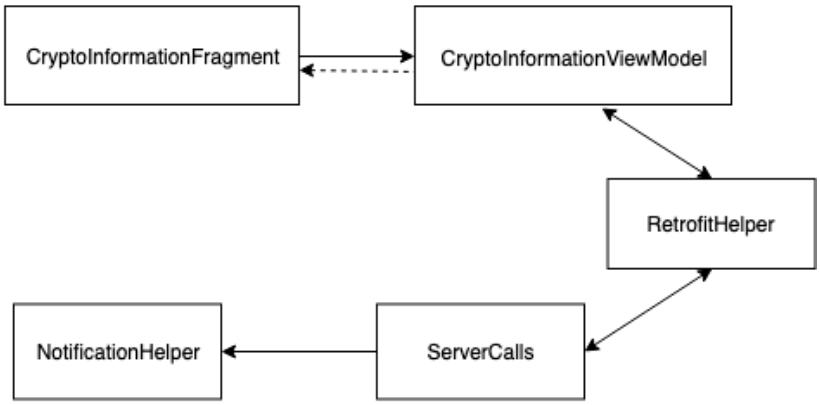
Obrázek 23 – Schéma komunikace mezi komponentami *CryptoListFragment*



Zdroj: vlastní zpracování

Jak již bylo v této diplomové práci zmíněné, je velice důležité komunikavat se serverem třetí strany ve vrstvě Model. V *CryptoInformationViewModel* je veškerá komunikace vytvořená skrze *RetrofitHelper* (blíže popsáno v další části práce). Při výběru volitelného období kurzů kryptoměn je nadále automatizována inicializace notifikací. Vývojář již tedy není povinen při každém použití této funkce kontrolovat, zda je potřeba pracovat s třídou *NotificationHelper*. Komunikace mezi jednotlivými částmi aplikace je znázorněna ve schématu na obrázku č. 24.

Obrázek 24 – Schéma komunikace mezi komponentami *CryptoInformationFragment*



Zdroj: vlastní zpracování

4.5 Návrhový vzor Fasáda pro knihovny třetích stran

Návrhový vzor Fasáda je obvykle používán jako rozhraní pro různé způsoby komunikace.

První typ použití je vhodný pro vývojáře, jejichž aplikace je komplexní a využívá mnoho serverových služeb. V této situaci se návrhový vzor používá následovně. Uživatel komunikuje s uživatelským rozhraním (které má v logice obrazovky právě Fasádu) a rozhodnutí, jaká služba bude použita, je v plné režii zmíněného naimplementovaného návrhového vzoru. (PECINOVSKÝ, 2007)

Dalším využitím, které je naimplementováno i v aplikaci diplomové práce, je pro snadnější komunikaci s knihovnami třetích stran. Některé knihovny (např. RxJava) je složité správně a efektivně používat. Z tohoto důvodu se naimplementuje rozhraní, které usnadní budoucí použití i ostatním členům týmu, kteří s touto vytvořenou třídou nadále pracují. (PECINOVSKÝ, 2007)

Nespornou výhodou při použití návrhového vzoru Fasáda je potencionální výměna používané knihovny nebo refaktorování již existující implementace. V tomto případě by veškeré změny byly provedeny pouze v rámci jedné třídy. (PECINOVSKÝ, 2007)

4.5.1 NotificationHelper

V předchozí kapitole diplomové práce je uvedena knihovna Firebase, díky které jsou vytvářeny lokální notifikace. Companion object *NotificationHelper* (obrázek č. 25) má naimplementovanou metodu *sendNotification()*, která obsahuje veškerou logiku pro vytvoření a zobrazení dané notifikace. Pro správné fungování při vstupu do metody je nutné vyplnit pouze dva parametry (text a context). Pokud je notifikace zobrazována na zařízení, které má verzi systému Android novější než Nougat (verze 7.0), je potřeba naimplementovat notifikační kanál. Poté přichází na řadu vytváření dané notifikace pomocí tečkové anotace návrhovým vzorem Stavitel. Je potřeba nastavit ikonu a barvu reprezentující aplikaci, nadpis, zprávu s informací pro uživatele a nastavení priority, aby byl objekt vytvořen. Finálním krokem je předání příkazu pro zobrazení notifikace uživateli.

Obrázek 25 – *NotificationHelper*

```
object NotificationHelper {
    fun sendNotification(messageBody: String, context: Context) {
        val c = context.applicationContext
        val notificationManager =
            c.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        val defaultSoundUri = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION)

        val CHANNEL_ID = "default"
        val CHANNEL_NAME = "Default"
        if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
            val defaultChannel = NotificationChannel(CHANNEL_ID, CHANNEL_NAME,
                NotificationManager.IMPORTANCE_HIGH)
            notificationManager.createNotificationChannel(defaultChannel)
        }

        val intent = Intent(context, MainActivity::class.java)
        intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK or FLAG_ACTIVITY_CLEAR_TOP
        val pendingIntent = PendingIntent.getActivity(c, requestCode: 0, intent,
            PendingIntent.FLAG_UPDATE_CURRENT)

        val notificationBuilder = NotificationCompat.Builder(c, CHANNEL_ID)
            .setSmallIcon(R.drawable.common_google_signin_btn_icon_dark_focused)
            .setColor(ContextCompat.getColor(c, R.color.colorAccent))
            .setContentTitle(c.getString(R.string.information))
            .setContentText(messageBody)
            .setWhen(System.currentTimeMillis())
            .setSound(defaultSoundUri)
            .setDefaults(Notification.DEFAULT_SOUND or Notification.DEFAULT_VIBRATE)
            .setLights(ContextCompat.getColor(c, R.color.colorPrimaryDark),
                onMs: 5000, offMs: 5000)
            .setAutoCancel(true)
            .setPriority(NotificationCompat.PRIORITY_MAX)
            .setContentIntent(pendingIntent)

        notificationManager.notify( tag: "myapp", id: 0, notificationBuilder.build())
    }
}
```

Zdroj: vlastní zpracování

4.5.2 IntervalThreadExecutor

V aplikaci je možné nastavit push notifikace, které dojdou uživateli při dosažení zvoleného kurzu vybrané kryptoměny. Aplikace se periodicky dotazuje serveru na nejnovější data, aby byl zajištěn aktuální kurz. K tomuto účelu je naimplementovaný *IntervalThreadExecutor*, který tuto funkci umožňuje. Celý *IntervalThreadExecutor* je naimplementovaný v Companion objektu. Používá se v situaci, kdy je potřeba třídu mírně modifikovat, aniž by byla vytvořena nová podtřída. V Javě je tato situace vyřešena anonymními vnitřními třídami. Hlavní motivace pro použití Companion objektu v diplomové práci je spuštění vlákna bez nutnosti vytvoření instance třídy. (KOTLIN, 2019)

Companion objekt lze použít v celé aplikaci, přičemž je možné vytvořit více vláken současně. *IntervalThreadExecutor* vlákno lze vytvořit způsobem, ve kterém je jasně definovaný konec cyklu a není potřeba ukládat referenci pro dodatečné ukončení.

K tomu, aby bylo vlákno uloženo do zásobníku, je nutné vyplnit parametr *name*, který slouží jako jednoznačný identifikátor pro právě vytvářené vlákno (jména vláken se tedy nesmí shodovat).

Pokud se jedná o vícevláknovou aplikaci, je nutné hlídat situace, které mohou během chodu aplikace nastat. První situací, která může nastat při nesprávné manipulaci s objekty je tzv. Deadlock. Naplánovaná akce prvního vlákna je podmíněna dokončení akce ve druhém vlákně, přičemž operace druhého vlákna čeká na ukončení operace vlákna prvního. Z této situace se nelze dostat, tím pádem se aplikace dostane do neřešitelného stavu. Přístup více vláken současně k jedné proměnné při neatomickém přístupu může vyústit v neplatnost přečtené hodnoty. V tu samou chvíli může probíhat operace v jiném vlákně, kde je hodnota právě zapisována a tudíž je dříve přečtená hodnota již neplatná. (POSCH, 2017)

Poslední situací, která může nastat při špatném přístupu k proměnné je tzv. vyhladovění (anglicky starvation). Aby byla vzniklá vlákna dokončena, potřebují informace, které jsou jim však odepírány. Ani v tomto případě není možné nastalou situaci vyřešit, a proto se aplikace dostane do neřešitelného stavu. Z důvodu atomicity je tedy veškerá komunikace se zásobníkem vláken vytvořena v bloku *synchronized()*. (POSCH, 2017)

Při vytváření vlákna je možné naimplementovat logiku do tří separovaných částí (demonstrováno na obrázku č. 26). První část probíhající ve vláknu na pozadí je *repeatBackgroundTask()*. Zde je možné provádět veškeré náročné operace typické pro asynchronní zpracování, jako komunikace se serverem, práce s velkými soubory nebo náročné výpočty. Rozhodování, zda bude vlákno pokračovat další iterací, je prováděno právě v této metodě. Návratová hodnota je typu boolean a vytvořené vlákno má stejné parametry jako cyklus do-while. První iterace cyklem proběhne vždy a pokud je návratová hodnota nastavena na false, vlákno proběhne i v další iteraci.

Následující metoda, která už probíhá na UI vláknu, je metoda *repeatUITask()*. Tato metoda je typicky používána v průběhu operace, kdy je potřeba po každé iteraci

v cyklu upravit vzhled grafického elementu. Pokud je celá operace dokončena, k finálnímu upravení grafických elementů je naimplementována metoda *finishUITask()*.

Obrázek 26 – Metoda pro vytvoření vlákna *IntervalThread*

```
@JvmOverloads
fun executeNewIntervalThread(name: String = "", interval: Long, timeout: Long,
                             repeatBackgroundTask: (tick: Int, timeout: Int) -> Boolean,
                             repeatUITask: ((timeout: Long, tick: Long) -> Unit)?,
                             finishUITask: (() -> Unit)?) {
    synchronized(disposableStack) {
        if (!disposableStack.containsKey(name)) {
            val observable = Observable.interval( initialDelay: 0, interval,
                TimeUnit.MILLISECONDS).map { tick : Long  ->
                IntervalThreadExecutorModel(
                    repeatBackgroundTask(tick.toInt(), timeout.toInt()),
                    timeout,
                    tick
                )
            }
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribeWith(object : DisposableObserver<IntervalThreadExecutorModel>() {
                    override fun onNext(intervalThreadExecutorModel: IntervalThreadExecutorModel) {
                        repeatUITask?.invoke(intervalThreadExecutorModel.timeout,
                            intervalThreadExecutorModel.tick)
                        if (intervalThreadExecutorModel.isFinished) {
                            finishUITask?.invoke()
                            disposeObservable( observable: this)
                        }
                    }
                    override fun onError(e: Throwable) {
                        Timber.e( message: " onError : %s, hashcode: %s", e.message, hashCode())
                        e.printStackTrace()
                        throw e
                    }

                    override fun onComplete() {
                        Timber.i( message: " onComplete, hashcode: %s", hashCode())
                    }
                })

            if (!name.isEmpty()) {
                disposableStack.putIfAbsent(name, observable)
            }
        }
    }
}
```

Zdroj: vlastní zpracování

4.5.3 RetrofitHelper

Komunikace se serverem třetí strany pomocí REST API je v aplikaci velmi důležitá. Třída je naimplementovaná jako návrhový vzor Jedináček, protože je v rámci celé aplikace potřeba mít pouze jednu instanci. Aby mohla být komunikace se serverem úspěšně navázána, je zapotřebí připojit do hlavičky každého REST volání klíč reprezentující autorizaci. Tuto operaci zajišťuje Interceptor.

Dále se nastaví objekt knihovny Retrofit, kde se specifikuje kořenová součást každého REST volání, Moshi převodník pro zpracování JSON souborů, adaptér RxJavy

(k možnosti reaktivního volání) a upravený klient pro možnost posílání autorizace skrz hlavičku volání.

Celý Retrofit objekt je na bázi návrhového vzoru Stavitel, tudíž je veškeré nastavení provedeno skrze tečkovou anotaci. K tomu, aby byla použita Retrofit knihovna, je zapotřebí pouze zavolat metodu *getClient()* a následně vybrat volání reprezentující logiku, která tyto data potřebuje (obrázek č. 27).

Obrázek 27 – Schéma komunikace mezi komponentami *CryptoInformationFragment*

```
class RetrofitHelper {

    private lateinit var retrofit: Retrofit
    private var ourInstance: RetrofitHelper? = null
    private lateinit var mUrlPath: ServerAPI
    private val key = "fb6593baaa90ec871bdad58fab030dd38f4c84089d130ba8541faf26e535d4a9"

    fun getClient(): ServerAPI {
        val interceptor = HttpLoggingInterceptor()
        interceptor.level = HttpLoggingInterceptor.Level.BODY

        val client = OkHttpClient.Builder().addInterceptor(interceptor)
            .addInterceptor { chain : Interceptor.Chain ->
            val newRequest = chain.request().newBuilder()
                .addHeader( name: "authorization"
                    , value: "Apikey $key"
                )
                .build()
            ^addInterceptor chain.proceed(newRequest)
        }.build()

        retrofit = Retrofit.Builder()
            .baseUrl( baseUrl: "https://min-api.cryptocompare.com/data/")
            .addConverterFactory(MoshiConverterFactory.create())
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
            .client(client)
            .build()

        mUrlPath = retrofit.create(ServerAPI::class.java)
        return mUrlPath
    }
}
```

Zdroj: vlastní zpracování

4.5.4 CustomCandleStickChart

Data stažená ze serveru třetí strany jsou zobrazeny ve svíčkovém grafu, který charakterizuje hodnoty kryptoměn za dané období nejlépe. Třída dědí *CandleStickChart*, která je součástí knihovny MPAndroidChart zmíněné výše. V rámci Android aplikace byly předem zvoleny kryptoměny, které jsou zobrazeny uživateli. Pro tento seznam je vytvořen tzv. Enum, kde jsou specifikovány názvy dané pro server a pro grafickou reprezentaci (obrázek č. 28).

Obrázek 28 – Definice názvů kryptoměn

```
enum class CryptoCurrency(val nameForServer: String, val nameForApp: String) {
    BITCOIN( nameForServer: "BTC",  nameForApp: "Bitcoin"),
    LITECOIN( nameForServer: "LTC",  nameForApp: "Litecoin"),
    BITCOIN_CASH( nameForServer: "BCH",  nameForApp: "Bitcoin Cash"),
    MONERO( nameForServer: "XMR",  nameForApp: "Monero"),
    DOGECOIN( nameForServer: "DOGE",  nameForApp: "Dogecoin"),
    ETHEREUM( nameForServer: "ETH",  nameForApp: "Ethereum")
}
```

Zdroj: vlastní zpracování

Při počáteční inicializaci je potřeba nastavit finální vzhled grafu. Jednotlivé svíčky grafu je možné libovolně přibližovat, popřípadě na ně kliknout pro dodatečnou informaci, za jakou cenu a v jaké období byla hodnota dosažena. Data jsou načítána postupně (3 sekundy), aby byl získán čas pro zorientování uživatele na konkrétní obrazovce, pomocí metody *animateX()*. Poté jsou nastaveny možnosti zobrazování hodnot kryptoměn na pravé straně grafu (osa Y). Popis období zobrazených hodnot není vypsán pomocí této knihovny, ale je doimplementován dodatečně v rámci diplomové práce, z důvodu více volitelných režimů. Knihovna taktéž poskytuje zobrazení této informace na ose X, ale při větším počtu dat není informace pro snadnou orientaci v grafu dostatečně deskriptivní. Zdrojový kód metody je vyobrazen na obrázku č. 29.

Obrázek 29 – Inicializace grafu

```
private fun initCustomChart() {
    this.isHighlightPerDragEnabled = true
    this.setDrawBorders(true)
    this.setBorderColor(Color.WHITE)
    this.setBackgroundColor(resources.getColor(R.color.colorPrimary))

    val yAxis = this.axisLeft
    val rightAxis = this.axisRight
    yAxis.setDrawGridLines(false)
    rightAxis.setDrawGridLines(false)
    this.requestDisallowInterceptTouchEvent( disallowIntercept: true)

    val xAxis = this.xAxis
    xAxis.setDrawGridLines(false)// disable x axis grid lines
    xAxis.setDrawLabels(false)
    rightAxis.textColor = Color.WHITE
    yAxis.setDrawLabels(false)
    xAxis.granularity = 1f
    xAxis.isGranularityEnabled = true
    xAxis.setAvoidFirstLastClipping(true)

    val l = this.legend
    l.isEnabled = false
    this.legend.isEnabled = false
    this.animateX( durationMillis: 3000)
}
```

Zdroj: vlastní zpracování

Zodpovědnost za vykreslování dat nese metoda *setDataProcess()*. Poté co jsou data stáhnuta a zpracována z formátu JSON do předem dané struktury, následuje

zpracování informací pro finální vykreslení. Jednotlivé svíčky v grafu se nastavují do objektu z knihovny MPAndroidChart *CandleEntry*.

Důležitými hodnotami jsou tzv. horní knot svíčky (high), dolní knot svíčky (low), cena při otevření trhu (open) a cena při zavření trhu (close) (obrázek č. 30). Při růstu ceny kryptoměny je barva nastavena na zelenou, zatímco při poklesu ceny je svíčka zabarvena do červena. Po nastavení všech hodnot do grafu je potřeba použít metodu *invalidate()*, která grafický element notifikuje o tom, že byl změněn a tudíž je potřeba jej překreslit.

Obrázek 30 – Nastavování hodnot do grafu

```
fun setDataProcess(response: MutableList<DataItem>?) {
    if (response != null) {
        val yValsCandleStick = ArrayList<CandleEntry>()
        for (i in 0 until response.size) {
            yValsCandleStick.add(
                CandleEntry(
                    i.toFloat(), response[i].high!!.toFloat(),
                    response[i].low!!.toFloat(), response[i].open!!.toFloat(),
                    response[i].close!!.toFloat()
                )
            )
        }
        val set1 = CandleDataSet(yValsCandleStick,  label: "DataSet 1")
        set1.color = Color.rgb( red: 80,  green: 80,  blue: 80)
        set1.shadowColor = Color.GRAY
        set1.shadowWidth = 0.8f
        set1.decreasingColor = Color.RED
        set1.decreasingPaintStyle = Paint.Style.FILL
        set1.increasingColor = Color.GREEN
        set1.increasingPaintStyle = Paint.Style.FILL
        set1.neutralColor = Color.LTGRAY
        set1.setDrawValues(false)
        val data = CandleData(set1)
        this.data = data
        this.invalidate()
    }
}
```

Zdroj: vlastní zpracování

4.6 Analýza tříd

Třídy vytvořené v rámci aplikace diplomové práce mají jednoznačnou úlohu pro výsledný chod aplikace. Podrobnější přehled jednotlivých tříd je popsán v dalších kapitolách.

4.6.1 LogActivity a LogFragment

Při vývoji aplikace je dobré si připravit třídy s dostatečně deskriptivním chováním, které usnadní programátorovi práci při vyvíjení nebo při opravě chyb. V projektu je každá aktivita nebo fragment potomkem logovacích abstraktních tříd, které pomáhají monitorovat celý životní cyklus aplikace. V každé metodě je

zaznamenáno jméno a identifikátor, zda log pochází z aktivity nebo fragmentu (==A== pro aktivitu a ==F== pro fragment). Logovací třídy vytvořené pro tuto aplikaci dále dědí ze tříd, které poskytuje Android API (konkrétně *AppCompatActivity* a Fragment), které dávají finální funkce reprezentující aktivity a fragmenty. Způsob použití je ukázán na obrázku č. 31.

Obrázek 31 – Logování životního cyklu

```
abstract class LogActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Timber.tag(this.javaClass.simpleName).d( message: "==A== onCreate: %s", hashCode())
    }


abstract class LogFragment : Fragment() {

    val managerTag: String
        get() = this.javaClass.name

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Timber.tag(this.javaClass.simpleName).d( message: "==F== onCreate: %s", hashCode())
    }
```

Zdroj: vlastní zpracování

4.6.2 BaseActivity

Účel aktivit v aplikaci bývá různý a záleží na vývojáři, jak dané uživatelské obrazovky reprezentuje a jak využívá jejich funkcionalitu. Hlavním cílem této třídy bylo zjednodušení budoucího vytváření a následného minimalizování kódu u nových aktivit. K tomuto účelu byla zvolena abstraktní třída (lze ji poznat dle identifikátoru *abstract* v hlavičce třídy), ze které nelze vytvořit instance objektu (není možno vytvořit obecnou instanci třídy BaseActivity). (PECINOVSKÝ, 2007)

S abstraktní třídou jsou spojeny i abstraktní metody, u kterých lze vynechat jejich tělo (zakončeno středníkem), přičemž tuto metodu musí implementovat všichni potomci, kteří tuto třídu dědí. Ukázka použití této metody bude v následující části diplomové práce. Poskytnuté informace jsou využity v metodě *onCreate()*, zbylé metody životního cyklu lze přepsat, jak v abstraktní třídě, tak ve finálních aktivitách aplikace. (HERODEK, 2014)

Všechny nově založené aktivity mají společné tři informace. První deklarovanou informací je odkaz na XML soubor aktivity pomocí konstrukce abstraktních metod. Vzhledem ke skutečnosti, že aktivita je prostředníkem ke spuštění fragmentů, je další informací první spustitelný fragment v daném rámci. Poslední společnou abstraktní metodou pro všechny nově vytvořené aktivity je reference na grafický element poskytující prostor pro zobrazení fragmentů (obrázek č. 32).

Důležitou konstrukcí pro další fungování během rotace zařízení je podmínka *savedInstanceState == null*. Ta brání opětovnému otevření startujícího fragmentu, protože po dokončení rotace jsou informace uloženy a lze tedy poznat, zda jde o novou obrazovku či znovu vytvořenou. (HERODEK, 2014)

Poslední metodou v této třídě je *goToFragment()*, která obstarává tzv. transakci pro vytvoření nového fragmentu a nahrazení již stávajícího.

Obrázek 32 – Abstraktní třída *BaseActivity*

```
abstract class BaseActivity : LogActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(contentViewResource())
        if (savedInstanceState == null) {
            goToFragment(startingFragment(), fragmentWrapperResource())
        }
    }

    abstract fun contentViewResource(): Int
    abstract fun startingFragment(): LogFragment
    abstract fun fragmentWrapperResource(): Int

    private fun goToFragment(fragment: LogFragment, wrapperResources: Int) {
        val transaction = supportFragmentManager.beginTransaction()
        transaction.replace(wrapperResources, fragment)
        transaction.commit()
    }
}
```

Zdroj: vlastní zpracování

4.6.3 BaseFragment

Třída *BaseFragment* je založena na podobném principu jako *BaseActivity*, až nrozdíly spojené s vytvářením nových fragmentů. V hlavičce třídy lze vidět použití identifikátoru *abstract* (popsáno v předchozí kapitole) a generické proměnné (obrázek č. 33). (SCHILDT, 2017)

Obrázek 33 – Inicializace třídy *BaseFragment*

```
abstract class BaseFragment<in DB : ViewDataBinding, VM : ViewModel>
    : LogFragment() {

    private lateinit var binding: DB

    protected lateinit var activityViewModel: MainActivityViewModel
    protected lateinit var fragmentViewModel: VM
```

Zdroj: vlastní zpracování

Informace spojené s vytvářením nových fragmentů jsou naimplementovány pomocí abstraktních metod. Při průchodu mobilní aplikací o více obrazovkách je pro uživatele velice důležité vědět, na jaké obrazovce se právě nachází. Z tohoto důvodu je vytvořena metoda *getTitle()* (obrázek č. 34), do které je přidán řetězec s potřebnou informací, která je přidána do záhlaví aplikace. Pro správné fungování fragmentu (viditelného pro uživatele) je potřeba nastavit referenci na XML soubor, kde je vzhled definovaný. K tomu slouží další abstraktní metoda *layoutResource()*.

Při běžném užití je pro fragment stěžejní metoda životního cyklu *onCreateView()*, protože se zde nastavuje daný vzhled a ostatní potřebné údaje ke správnému fungování vývojářem naimplementované logiky. Zároveň je zde prostor pro přípravu dat pomocí hlavičky již definované generiky a typu souboru určeného pro práci s data bindingem. K tomuto účelu slouží poslední abstraktní metoda *onCreateViewCustom(),* která dokáže vygenerovat potřebnou instanci objektu a lze s ní nastavit ostatní atributy. (LACKO, 2015)

Obrázek 34 – Abstraktní metody třídy *BaseFragment*

```
abstract fun getTitle(): String
abstract fun layoutResource(): Int
abstract fun onCreateViewCustom(binding: DB)
```

Zdroj: vlastní zpracování

Jak již bylo zmíněno výše, jedna z nejdůležitějších vazeb v architektuře aplikace této diplomové práce je propojení ViewModelu aktivity s *BaseFragmentem*. Při inicializaci fragmentu je požádáno o propojení v metodě *onCreate()*, aby došlo k automatizaci procesu (obrázek č. 35). Tento postup je výhodný při dalším použití dané

proměnné, protože není nutné tuto logiku znovu implementovat s každým novým fragmentem.

Obrázek 35 – Získání instance objektu *MainActivityViewModel*

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    activityViewModel = ViewModelProviders.of(activity!!).
        get(MainActivityViewModel::class.java)
}
```

Zdroj: vlastní zpracování

Další metodou životního cyklu, která připravuje výše zmíněná data abstraktních metod je *onCreateView()*. Zde je poprvé použita technologie data bindingu. Pro svoji inicializaci potřebuje referenci na XML soubor reprezentující vzhled. Dále přeposílá argument container a poslední parametr je nastaven na hodnotu false. Zadání opačné hodnoty (true) je v tomto rozvržení nežádoucí, protože by poté došlo k přidání grafických elementů z fragmentů do kořenového prvku (obrázek č. 36).

Obrázek 36 – Inicializace vzhledu fragmentu pomocí data bindingu

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {

    binding = DataBindingUtil.inflate(inflater,
        layoutResource(),
        container,
        attachToParent: false)
    onCreateViewCustom(binding)
    activity?.title = getTitle()
    return binding.root
}
```

Zdroj: vlastní zpracování

4.6.4 MainActivity

Výsledkem snahy minimalizovat kód pouze na logiku dané obrazovky, pomocí třídy *BaseActivity* vytvořené v rámci této diplomové práce, je např. *MainActivity* (obrázek č. 37). Tělo třídy je pouze na třech řádcích kódu a zároveň zajištěna celá funkcionalita týkající se propojení aktivity a fragmentů.

Jak již bylo uvedeno výše, nejprve se metoda jen definuje a posléze je na cílovém místě metoda rozšířena i o tělo, se kterým se nadále pracuje. V tomto případě nebylo

zapotřebí užití rozsáhlé logiky a vyhodnocování, postačilo pouze přidání reference na žádaný objekt.

Obrázek 37 – Použití abstraktních metod pro vytvoření aktivity

```
class MainActivity : BaseActivity() {
    override fun contentViewResource(): Int = R.layout.activity_main

    override fun startingFragment(): LogFragment = CryptoListFragment()

    override fun fragmentWrapperResource(): Int = R.id.fragmentWrapperMain
}
```

Zdroj: vlastní zpracování

4.6.5 MainActivityViewModel

Mezi fragmenty je v rámci vytvořené aplikace nutné posílat právě jednu informaci, a to informaci o uživatelem zvolené kryptoměně. Vytvořená konstrukce předávání dat mezi uživatelskými obrazovkami je navrhnuta i pro předávání většího počtu proměnných. Z tohoto důvodu je velice jednoduché rozšířit současnou funkcionalitu o další parametry. *MainActivityViewModel* (obrázek č. 38) dědí třídu ViewModel, která je součástí výše zmíněného Android Jetpack balíku pro vývojáře.

Obrázek 38 – ViewModel pro přenášení informací mezi fragmenty

```
class MainActivityViewModel : ViewModel() {
    lateinit var chosenCrypto: CustomCandleStickChart.CryptoCurrency
}
```

Zdroj: vlastní zpracování

4.6.6 CryptoListFragment

Prvním viditelným fragmentem pro uživatele je *CryptoListFragment*, který obsahuje list vybraných kryptoměn. Na této uživatelské obrazovce si může uživatel zvolit jednu z šesti vybraných kryptoměn, jejíž kurzový vývoj za vybrané období chce znát. Obrazovka obsahuje genericky upravený *RecyclerView* (generický adaptér bude popsán v další podkapitole diplomové práce), kde každá položka obsahuje *ImageView* (v aplikaci pouze obrázky typu SVG) a *TextView*. Jedná se o třídu s prvním použitím generiky a data bindingu. Generika je deklarována přímo v hlavičce třídy, s tím, že následné metody mají již požadovaný objekt. Zdrojový kód je na obrázku č. 39.

Obrázek 39 – Hlavička třídy *CryptoListFragmentu*

```
class CryptoListFragment : BaseFragment<FragmentCryptoListBinding, CryptoListViewModel>() {
```

Zdroj: vlastní zpracování

V cílovém fragmentu jsou používány abstraktní metody ke stejnému účelu, jako při implementaci aktivit. Díky proměnné typu Disposable (obrázek č. 40) lze korigovat registrace na notifikační kanál, případně i odregistrování. (REACTIVEX, 2019)

Obrázek 40 – Nastavené informace pro vytvoření fragmentu

```
lateinit var subject: Disposable

override fun getTitle(): String = "Crypto List"

override fun layoutResource(): Int = R.layout.fragment_crypto_list
```

Zdroj: vlastní zpracování

Tvorbě vzhledu uživatelské obrazovky předchází práce s ViewModelem dané obrazovky. Nejprve je nutné zjistit, zda jsou data již uložena z předchozí relace (např. rotace mobilního zařízení, uchování aplikace v zásobníku a mnoho dalších situací), případně je zahrnout do vykreslování obrazovky (obrázek č. 41). Tento proces by měl, z důvodu zahrnutí výsledku do pozdějšího vyhodnocování, probíhat v první metodě životního cyklu *onCreate()*. (VÁVRŮ, UJBÁNYAI, 2013)

Obrázek 41 – Inicializace ViewModelu

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    fragmentViewModel =
        ViewModelProviders.of( fragment: this).get(CryptoListViewModel::class.java)
}
```

Zdroj: vlastní zpracování

Při vzniku uživatelského rozhraní je nutno se registrovat k *PublishSubjektu*, který je vytvořen v *CryptoListViewModel* (bude popsáno v další části diplomové práce). Při získání notifikace o uskutečnění vyběru dané kryptoměny uživatelem, je pomocí lambda funkce poslán i *CryptoListModel* (datová třída), který v sobě obsahuje všechny potřebné informace. Údaj o vybrané kryptoměně je uložen do *MainActivityViewModelu* pro následné využití v dalším fragmentu. Po zapsání informace je připravena tzv. transakce, sloužící k nahrazení stávajícího fragmentu. Zdrojový kód reprezentující výše uvedenou problematiku je k dispozici na obrázku č. 42.

Obrázek 42 – Veškeré funkční nastavení třídy *CryptoListFragment*

```
override fun onCreateViewCustom(binding: FragmentCryptoListBinding) {
    binding.viewModel = fragmentViewModel

    subject = fragmentViewModel.publishSubject.subscribe { cryptoModel: CryptoListModel ->
        val activityViewModel = ViewModelProviders.of(activity!!).
            get(MainActivityViewModel::class.java)
        activityViewModel.chosenCrypto = cryptoModel.crypto

        val transaction = activity?.supportFragmentManager?.beginTransaction()
        transaction?.replace(R.id.fragmentWrapperMain,
            CryptoInformationFragment())?.addToBackStack( name: null)
        transaction?.commit()
    }
}
```

Zdroj: vlastní zpracování

Poslední volanou metodou v rámci životního cyklu při zničení fragmentu je *onDestroyView()* (obrázek č. 43). Zpravidla je tato metoda volána pokud bylo grafické rozhraní při vzniku fragmentu vytvořeno pomocí *onCreateView()*. Při opětovném vzniku fragmentu se uživatelské zobrazení opět vytvoří. V rámci aplikace byla metoda použita pro odregistrování z objektu Disposable, jehož fragment notifikuje o provedené akci vybrané kryptoměny s příslušnými daty. (VÁVRŮ, UJBÁNYAI, 2013)

Obrázek 43 – Odregistrování objektu *PublishSubject*

```
override fun onDestroyView() {
    super.onDestroyView()
    subject.dispose()
}
```

Zdroj: vlastní zpracování

4.6.7 CryptoListViewModel

Dle MVVM architektury, která je popsána v předchozí kapitole diplomové práce, je důležitým faktorem každého fragmentu ViewModel, který zastává funkci zprostředkovatele mezi uživatelskou interakcí a logikou dané obrazovky. Třída *CryptoListViewModel* dědí objekt ViewModel (obrázek č. 44), dostupný z Android API (součást knihovny Jetpack).

Obrázek 44 – Inicializace *CryptoListViewModel*

```
class CryptoListViewModel : ViewModel() {
```

Zdroj: vlastní zpracování

V aplikaci diplomové práce je seznam kryptoměn určen předem. Pro naplnění seznamu jsou potřeba dva parametry. Prvním zmíněným parametrem je Enum položka, která navíc obsahuje informaci o zkratce použitelnou pro server třetí strany a plný název zobrazený uživateli (obrázek č. 45). Dalším parametrem je reference na lokálně uložené logo dané kryptoměny ve formátu SVG.

Obrázek 45 – Seznam kryptoměn

```
val list = listOf(
    CryptoListModel(CustomCandleStickChart.CryptoCurrency.BITCOIN,
        R.drawable.ic_bitcoin),
    CryptoListModel(CustomCandleStickChart.CryptoCurrency.LITECOIN,
        R.drawable.ic_litecoin),
    CryptoListModel(CustomCandleStickChart.CryptoCurrency.BITCOIN_CASH,
        R.drawable.ic_bch),
    CryptoListModel(CustomCandleStickChart.CryptoCurrency.ETHEREUM,
        R.drawable.ic_ethereum),
    CryptoListModel(CustomCandleStickChart.CryptoCurrency.MONERO,
        R.drawable.ic_monero),
    CryptoListModel(CustomCandleStickChart.CryptoCurrency.DOGECOIN,
        R.drawable.ic_dogecoin)
)
```

Zdroj: vlastní zpracování

Hlavní podmínkou při konstrukci MVVM architektury je nepřítomnost reference ve ViewModulu na grafické elementy. Existují však situace, při kterých je potřeba komunikovat pomocí notifikací z ViewModelu s grafickými elementy (v této diplomové práci je příkladem výběr dané kryptoměny uživatelem). Z tohoto důvodu je vytvořen PublishSubject (obrázek č. 46), který tuto komunikaci umožnuje. Při vzniku instance objektu je nezbytné určit, jaký typ objektu bude notifikací posílán. (DEVELOPER.ANDROID/VIEWMODEL, 2019)

Obrázek 46 – Inicializace objektu *PublishSubject*

```
val publishSubject = PublishSubject.create<CryptoListModel>()
```

Zdroj: vlastní zpracování

Počáteční definice generického adaptéru pro *RecyclerView* je ve všech možných způsobech použití identický (to je také hlavní motivací pro vytvoření generického

adaptéru a jeho následné používání). Odlišnostmi rozdělující funkčnost pro různé seznamy jsou zvolený návratový typ při uživatelské interakci, reference na XML soubor reprezentující vzhled položky, vytvořený ViewHolder (každý seznam reprezentuje daný ViewHolder) a následná reakce po kliknutí na položku v seznamu. Zde je použita interakce mezi ViewModelem a daným fragmentem pomocí notifikace, ve které je i veškerá informace o objektu, který je uložený v dané položce (obrázek č. 47).

Obrázek 47 – Inicializace *GenericRecyclerViewAdapteru*

```
val cryptoListAdapter =
    object : GenericRecyclerViewAdapter<CryptoListModel>
        (list, object : ItemClickListener<CryptoListModel> {
        override fun onItemClicked(clickedItem: CryptoListModel) {
            Timber.i( message: "I just click: %s", clickedItem.crypto.nameForServer)
            publishSubject.onNext(clickedItem)
        }
    }) {
        override fun getLayoutId(position: Int, item: CryptoListModel): Int = R.layout.item_crypto_list

        override fun getViewHolder(view: View, viewType: Int): RecyclerView.ViewHolder {
            return CryptoListViewHolder(view)
        }
    }
```

Zdroj: vlastní zpracování

Na konci třídy *CryptoListViewModel* je přepsána metoda z Android API *onCleared()*, která je volána, pokud je daný ViewModel již nepoužívaný a bude zničen. V aplikaci diplomové práce je odregistrování a finální ukončení *PublishSubjektu* (obrázek č. 48), který už nebude notifikovat žádné události. Při opakovaném vytváření ViewModelu a potencionálním neukončování *PublishSubjektu* by mohlo dojít k úniku paměti, což může způsobit nekorektní chování nebo dokonce pád celé aplikace. (REACTIVEX, 2019)

Obrázek 48 – Odregistrování objektu *PublishSubject*

```
override fun onCleared() {
    super.onCleared()
    publishSubject.onComplete()
}
```

Zdroj: vlastní zpracování

4.6.8 CryptoInformationFragment

Jak již bylo deklarováno v předchozí kapitole diplomové práce, hlavním rozdílem mezi fragmenty je specifická logika, která je vykonávaná v těle metody životního cyklu, v tomto případě metody *onCreateView()*. Při zobrazení tohoto fragmentu se již uživatel

rozhodl, jakou kryptoměnu chce mít zobrazenou. Je tedy velice důležité před prvním načtením vzhledu aplikace zjistit o jakou kryptoměnu se jedná a posléze získat ze serveru třetí strany relevantní data. V předchozí kapitole je již poukázáno na fakt, že informace o zvolené kryptoměně je posílána přes *MainActivityViewModel*, která je přístupná ze všech fragmentů spadajících do téže aktivity. Je potřeba si informaci o zvolené kryptoměně nejdříve uložit do proměnné *chosenCrypto* ve třídě *CryptoInformationViewModel*, aby bylo možné ji nadále použít.

Poté je inicializována komponenta zobrazující kalendář a implementováno následné zpětné volání při zvolení časového období. Zvolené období musí být delší než je sedm dní. Pokud zvolené období neodpovídá naimplementovanému rozmezí, uživatel je o této skutečnosti informován skrz tzv. *Toast*. Pokud je zvolené období dostatečně dlouhé, uživateli se opět zobrazí svíčkový graf s odpovídajícími daty (obrázek č. 49).

Obrázek 49 – Inicializace *CryptoInformationFragment*

```
override fun onCreateViewCustom(binding: FragmentCryptoInformationBinding) {
    binding.viewModel = fragmentViewModel
    fragmentViewModel.chosenCrypto = activityViewModel.chosenCrypto.nameForServer
    fragmentViewModel.appContext = context!!.applicationContext
    fragmentViewModel.prepare()
    fragmentViewModel.callServer()

    val nextYear = Calendar.getInstance()
    nextYear.add(Calendar.MONTH, -18)

    val today = Date()
    binding.root.calendarPicker.init(nextYear.time, today)
        .inMode(CalendarPickerView.SelectionMode.RANGE)
    binding.root.calendarPicker.scrollToDate(today)
    binding.root.calendarPicker.setOnDateSelectedListener(object:
        CalendarPickerView.OnDateSelectedListener {
        override fun onDateSelected(date: Date?) {
            if (binding.root.calendarPicker.selectedDates.size > 1) {
                if (binding.root.calendarPicker.selectedDates.size >= 7 ) {
                    val calendar = Calendar.getInstance()
                    calendar.time = binding.root.calendarPicker
                        .selectedDates[binding.root.calendarPicker.selectedDates.size -1]
                    calendar.add(Calendar.DAY_OF_MONTH, 1)
                    val calendarDates = binding.root.calendarPicker.selectedDates
                    calendarDates.add(calendar.time)
                    fragmentViewModel.showCustomData(calendarDates, today)
                } else {
                    Toast.makeText(activity,  text: "More than 7 days", Toast.LENGTH_LONG).show()
                }
            }
        }
        override fun onDateUnselected(date: Date?) {
        }
    })
}
```

Zdroj: vlastní zpracování

4.6.9 CryptoInformationViewModel

Vzhledem k absenci propojení serverové a klientské části je potřeba si uložit informaci o tom, zda byla zadána hodnota k periodické kontrole. V metodě *prepare()* je k tomuto účelu použit *SharedPreferences,* do kterého je hodnota uložena a je stále přístupná i po zničení ViewModelu. Pokud je hodnota uložena z předchozí relace, tak je potřeba nejdříve původně nastavenou hodnotu zrušit a případně nastavit novou. (LACKO, 2015)

Při vstupu uživatele do *CryptoInformationFragmentu* je důležité mít nahraná data z *CryptoInformationViewModelu*, která budou zobrazena uživateli. Pro defaultní zobrazení dat je připravena metoda *callServer()*. Jak již bylo zmíněno výše, maximální časové období, které dokáže zobrazit aplikace diplomové práce, je rok a půl. Finální metoda pro komunikaci se serverem třetí strany pomocí REST API je naimplementována v *CryptoInformationModel*. Pro úspěšné získání dat je nutné zaslat informaci o názvu (zkratka) zvolené kryptoměny, počet dní (hodin) a lambda funkci, ve které se provede operace po úspěšném doručení dat ze serveru. Všechny data s časovými údaji přichází ve formátu Long, proto je důležité si tento formát konvertovat na objekt Date() s formátem času dd.MM.yyyy H:mm (obrázek č. 50). (ALLEN, 2013)

Obrázek 50 – Metoda pro výpočet dnů ve zvoleném časovém rozmezí

```
fun countHowManyDaysToPast(dates: MutableList<Date>, todayDate: Date): Int {
    val diff = todayDate.time - dates[0].time
    return TimeUnit.DAYS.convert(diff, TimeUnit.MILLISECONDS).toInt()
}

fun filterSpecificDatesFromData(
    data: MutableList<DataItem>,
    dates: MutableList<Date>
): MutableList<DataItem> {
    return data.filter { it: DataItem
        (Date( date: it.time!! * 1000).after(dates[0]) && Date( date: it.time!! * 1000)
            .before(dates[dates.size - 1])) || Date(
            date: it.time!! * 1000
        ) == dates[dates.size - 1]
    }.toMutableList()
}

@SuppressLint( ...value: "SimpleDateFormat")
fun convertLongDateToHumanReadableDate(date: Long): String =
    SimpleDateFormat( pattern: "dd.MM.yyyy H:mm").format(Date( date: date * 1000))
```

Zdroj: vlastní zpracování

Vybraný poskytovatel nepodporuje možnost zaslání informace o začátku a konci sledovaného období. Při označení časového období musí být proveden vypočet

k získání přesného počtu dnů od stávajícího data do začátku zvoleného období uskutečněn na klientské straně (aplikace diplomové práce). V aplikaci je tedy vykonáván výpočet, který by v případě přítomnosti serverové části byl prováděn právě tam. Po výpočtu je výsledné číslo zasláno serveru. Po obdržení dat je opět nutné filtrovat, které informace spadají do zvoleného časové období. Vybraná data jsou zobrazena a zbytek dat je zahozen.

4.7 Testování

Naprogramovat všechny plánované funkce a vykreslit grafické komponenty podle drátových modelů, nejsou posledními kroky v procesu při vývoji aplikace. Aby bylo možné určit, zda je aplikace připravena ke zveřejnění a zda ji může potencionální uživatel plnohodnotně používat, je zapotřebí provést řádné testování.

4.7.1 Unit testy

Pro kontrolu správnosti fungování a korektnosti dílčí logiky aplikace slouží tzv. unit testy. Aby se dalo posoudit, zda pracuje logika podle očekávání, je nutné vytvořit každý testovací případ nezávislý na zbylých testech. Dalším důležitým faktorem je izolace testované logiky od zbylé části programu (logika by měla být ovlivněna pouze vstupními daty, nikoliv okolními jevy).

V aplikaci diplomové práce je unit test použit pro ověření správnosti výpočtu při výběru volitelných dat. U testů vytvořených v jazyku Kotlin je odlišnost od testů naprogramovaných v Javě zřejmá již na první pohled. Liší se názvem nesoucí mezery a jsou ohraničeny gravisy. Tato vlastnost umožňuje vývojáři lépe demonstrovat a deskriptivně popsat danou funkcionalitu, který test zastává (tato vlastnost platí pouze pro metody vytvořené pro testování). Metoda s názvem *countHowManyDaysToPast* (obrázek č. 51) obsahuje logiku, kterou je nutné testovat. Pro vyhodnocení výsledku unit testu je použita metoda *assertEquals*, která obsahuje dva argumenty (předpokládaný výsledek a skutečný výsledek), které navzájem porovná a vyhodnotí, zda jsou výsledky ve shodě či nikoli.

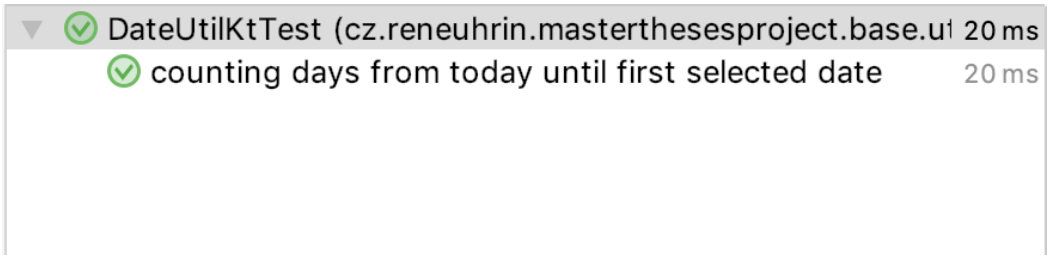
Obrázek 51 – Unit test pro kontrolu výpočtu dnů v intervalu

```
@Test
fun `counting days from today until first selected date`() {
    val dateList = arrayListOf<Date>()
    dateList.add(Date( year: 2019,  month: 1,  date: 24))
    dateList.add(Date( year: 2019,  month: 1,  date: 27))
    Assert.assertEquals(
         expected: 4,
        countHowManyDaysToPast(dateList,
            Date( year: 2019,  month: 1,  date: 28))
    )
}
```

Zdroj: vlastní zpracování

Po proběhnutí testu lze ve vývojovém prostředí Android Studia lehce vyhodnotit, zda test proběhl úspěšně nebo je daná logika špatně naimplementovaná. Pokud se zobrazí zelené potvrzení (demonstrováno na obrázku níže), lze předpokládat správnost unit testu (obrázek č. 52).

Obrázek 52 – Výsledky unit testování



Zdroj: vlastní zpracování

4.7.2 Testy na reálných zařízeních a emulátorech

Mnoho výrobců používá operační systém Android pro své mobilní telefony a další zařízení. V současnosti se rozměry obrazovek mobilních telefonů pohybují nejčastěji od 4 do 6 palců a rozměry obrazovek tabletů od 7 do 10 palců. Navíc mají odlišné poměry stran displeje (např. 4:3, 16:9, 16:10 a další). Z tohoto důvodu jsou uživatelé často svědkem neuspořádaného zobrazení jednotlivých grafických komponent na obrazovce. Před zveřejněním aplikace je tedy velice důležité aplikaci testovat na co možná nejvíce zařízeních.

Testování celé škály rozměrů na reálných zařízeních bývá pro firmy nákladnou záležitostí. Často je k pokrytí testování veškerých dostupných variant nutná koupě vybraných zařízení. Z tohoto důvodu se tento druh testování kombinuje s testováním na

virtuálních strojích (emulátory vytvořené pomocí vývojového prostředí), které zastupují zbývající vzorek zařízení.

Pro testování aplikace diplomové práce byly zvoleny následující mobilní zařízení:

- Samsung S9 (Android Pie)
- Xiaomi Mi A1 (Android Oreo)
- Nexus 5 (Android Oreo, vytvořený emulátor)
- Pixel XL (Android Pie, vytvořený emulátor)

Na výše zmíněných zařízeních bylo provedeno testování, které ověřilo, zda bylo rozvržení grafických komponent na všech obrazovkách korektní. Další iterace testování ověřila správnost chování při rotaci zařízení (horizontální i vertikální poloze).

5 Výsledky a diskuse

Výsledná mobilní aplikace pro operační systém Android byla naimplementována podle cílů stanovených na začátku diplomové práce. Nicméně jak je v rámci programování aplikací běžné, je i v této aplikaci ještě stále prostor pro rozvíjení stávajícího řešení.

5.1 Zasílání notifikací

Jednou z funkcí aplikace jsou příchozí push notifikace, které informují uživatele o dosažení předem stanoveného kurzu. Tyto notifikace se tvoří v aplikaci lokálně. Aby však mohl být uživatel upozorněn na vzniklou situaci, musí být aplikace otevřena na popředí, nebo být alespoň v zásobníku otevřených aplikací na pozadí. V případě propojení se serverovou částí (není předmětem této diplomové práce), kde by tato logika vyhodnocení proběhla, by aplikace mohla být zavřená a potencionálnímu uživateli by se push notifikace i přesto objevila.

Vzhledem k nepřítomnosti serveru, jsou v aplikaci navíc i další implementace logiky. Klientská část by měla, v ideálním případě, pouze zastávat funkci přístroje komunikujícího s koncovým uživatelem. Tím by se centralizovala co největší část společné logiky na straně serveru. Při multiplatformním vývoji aplikací by tak nedocházelo k replikování chyb na všech platformách a následné opravy chyb by byly uskutečňovány na jednom místě.

5.2 Návrhový vzor Fasáda

Další možné rozšíření stávajícího řešení je zdokonalení současných tříd, které splňují funkci prostředníka mezi programátorem a reálnou knihovnou. V současné chvíli je návrhový vzor Fasáda použit na hlavní funkcionalitu, která obsahuje i větší část kódu. Pro jednotlivé součásti tato Fasáda však naimplementovaná není. Pro zlepšení současné implementace by bylo nutné vytvořit třídu, která by navíc implementovala rozhraní pro *PublishSubject* a Disposable. Vývojář by pak nemusel vzniklou komunikaci mezi objekty ručně odregistrovávat.

5.3 Celkové zhodnocení aplikace a diplomové práce

Diplomová práce se zaměřuje na proces vývoje mobilní aplikace pro platformu Android. Uvedené postupy a zdrojový kód mohou být inspirací pro další Android programátory, kteří by chtěli naprogramovat stabilní mobilní aplikaci s moderními prvky vývoje za použití osvědčené technologie. Dále může aplikace sloužit uživatelům sledujícím aktuální vývoj kurzů kryptoměn, kteří ji mohou použít jako informační zdroj při rozhodování pro jejich nákup či prodej. Vývoj kurzu je také možné hlídat pomocí push notifikací, které upozorní uživatele při dosažení zvoleného kurzu.

Diplomová práce splňuje předem stanovené cíle i dílčí cíle a vývoj aplikace probíhal dle dané metodiky. Architektura aplikace je rozdělena do jednotlivých vrstev, její zdrojový kód je přehledný (pro ostatní vývojáře snadno čitelný) a dobře testovatelný pomocí unit testů. Aplikace je díky zvolené architektuře a způsobu implementace škálovatelná, tudíž je v případě budoucího rozšíření možné aplikaci rozšířit o libovolný počet uživatelských obrazovek nebo funkcí. V rámci budoucího rozšíření je také možné přidat UI a instrumentální testy.

6 Závěr

V této diplomové práci byl představen vývoj mobilní aplikace pro monitorování změn kurzů vybraných kryptoměn pro platformu operačního systému Android, naimplementované použitím moderních postupů, technologií a nejnovějších programovacích jazyků. Při vývoji byly prostudovány aktuální literární díla s touto problematikou a uplatněny znalosti získané během dosavadní praxe. Při výběru architektury a moderních technologií byly brány v potaz především předchozí zkušenosti s danou knihovnou nebo technikou vývoje, jejich použitelnost a dostupnost informačních zdrojů o dané problematice.

V diplomové práci je zdokumentován celý vývojový cyklus. Samotnému vývoji mobilní aplikace v této diplomové práci předcházel výběr vhodných knihoven a komunikačních serverů třetích stran. Dále byla představena implementace architektury a generických tříd a na závěr proběhlo finální testování vybrané logiky pomoci unit testů, zatímco celkový vzhled aplikace byl otestován na reálných a virtuálních zařízeních.

Pro vývoj klientské části byly zvoleny a použity programovací jazyky Kotlin a Java ve verzi 8. V souvislosti s již zmíněnou Javou bylo přidáno reaktivní rozšíření ve formě knihovny RxJava. Zvolená architektura MVVM pro projekt diplomové práce byla použita i s doplňující knihovnou data binding. Pro snadnější rozšiřitelnost budoucích aktivit a fragmentů byla použita genericita. Další komponenty použité z Android Jetpack byly ViewModely, které se používají v již zmíněné MVVM architektuře. Byl nakonfigurován Gradle soubor pro řádnou kompilaci aplikace pro vývoj nebo případné zveřejnění APK souboru do obchodu Google Play.

Pro lepší orientaci uživatele byly použity nativní prvky tzv. Material Designu, který je použit ve všech Googlem vytvořených aplikacích. Tyto grafické komponenty jsou již přítomny ve vývojovém prostředí, tudíž není potřeba stahovat žádnou dodatečnou knihovnu, jelikož jsou součástí Android API (součást Android SDK). Aplikace komunikuje pomocí REST API se serverem třetí strany a získaná data jsou následně vykreslena ve svíčkovém grafu.

Pro vytvoření lokálních push notifikací byla použita knihovna Firebase. Celý proces vývoje byl lokálně ukládaný do verzovacího systému Git, který zaručoval zálohování předchozích relací souborů během implementace aplikace.

Výsledkem této diplomové práce je tedy vytvoření stabilní mobilní aplikace s moderními prvky vývoje, při kterém byly použity osvědčené technologie. Pomocí této Android aplikace si uživatel může zvolit konkrétní kryptoměnu, sledovat aktuální vývoj zvoleného měnového kurzu v různých časových intervalech a rozhodovat se na základě těchto údajů pro nákup či prodej dané kryptoměny. Uživatel si taktéž může nastavit upozornění ve formě push notifikace, která se objeví, jakmile bude dosažena uživatelem zvolená výše kurzu.

Závěrem lze konstatovat, že vzniklá mobilní aplikace může posloužit nadále i ostatním Android vývojářům pro svou snadnou čitelnost, nejen díky přehlednosti zdrojového kódu dosažené rozdělením architektury do jednotlivých vrstev, ale i velice jednoduché testovatelnosti pomocí unit testů. Vzhledem k možné škálovatelnosti aplikace je v případě budoucího rozšíření možné aplikaci testovat i pomoci UI a instrumentálních testů.

7 Seznam použitých zdrojů

7.1 Tištěné zdroje

ALLEN, Grant. Android 4: průvodce programováním mobilních aplikací. Brno: Computer Press, 2013. ISBN 978-80-251-3782-6.

DRESCHER, Daniel. Blockchain Basics: A Non-Technical Introduction in 25 Steps. APress, 2017. ISBN 1484226038.

EDDISON, Leonart. Ethereum: A Deep Dive Into Ethereum. CreateSpace Independent Publishing Platform, 2017. ISBN 978-80-271-0742-1.

HERODEK, Martin. Android: jednoduše. Brno: Computer Press, 2014. ISBN 978-80-251-4298-1.

KALISKÝ, Boris. Bitcoin a ti druzí: Nepostradatelný průvodce světem kryptoměn. IFP Publishing, 2018. ISBN 978-80-87383-71-1.

LACKO, Ľuboslav. Vývoj aplikací pro Android. Brno: Computer Press, 2015. ISBN 978-80-251-4347-6.

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

POSCH, Maya. Mastering C++ Multithreading. Packt Publishing Limited, 2017. ISBN 9781787121706.

SCHILDT, Herbert. Mistrovství - Java: Kompletní průvodce vývojáře. Computer Press, 2017. ISBN 978-80-251-4145-8.

STROUKAL, Dominik, SKALICKÝ, Jan. Bitcoin a jiné kryptopeníze budoucnosti. Grada, 2018. ISBN 978-80-271-0742-1.

THOMPSON, Christopher P. Dogecoin - History of the First Year. CreateSpace Independent Publishing Platform, 2015. ISBN 1519667353.

VÁVRŮ, Jiří, UJBÁNYAI, Miroslav. Programujeme pro Android. Praha: Grada, 2013. ISBN 978-80-247-4863-4.

7.2 Online zdroje

ANDROID/HISTORY. Platforma Android [online]. [cit. 2018-11-28].
Dostupné z WWW: https://www.android.com/history

BITCOINCASH. Bitcoin Cash oficiální stránky [online]. [cit. 2018-11-28].
Dostupné z WWW: https://www.bitcoincash.org/

DEVELOPER.ANDROID/DATA-BINDING. Android komponenty [online].
[cit. 2019-01-23]. Dostupné z WWW:
https://developer.android.com/topic/libraries/data-binding

DEVELOPER.ANDROID/GRADLE-PLUGIN [online]. [cit. 2018-12-15].
Dostupné z WWW: https://developer.android.com/studio/releases/gradle-plugin

DEVELOPER.ANDROID/GRADLE-TIPS. Oficiální dokumentace Google [online].
[cit. 2018-12-15]. Dostupné z WWW:
https://developer.android.com/studio/build/gradle-tips

DEVELOPER.ANDROID/JETPACK. Jetpack oficiální stránky [online].
[cit. 2019-05-01]. Dostupné z WWW: https://developer.android.com/jetpack

DEVELOPER.ANDROID/VIEWMODEL. Android DataBinding [online].
[cit. 2019-02-28]. Dostupné z WWW:
https://developer.android.com/topic/libraries/architecture/viewmodel

FIREBASE.GOOGLE. Firebase oficiální stránky [online]. [cit. 2019-02-10].
Dostupné z WWW: https://firebase.google.com

GETMONERO. Monero Oficiální stránky [online]. [cit. 2018-10-10].
Dostupné z WWW: https://www.getmonero.org/

KOTLIN. Kotlin (programovací jazyk) [online]. [cit. 2019-01-08].
Dostupné z WWW: https://kotlinlang.org

LITECOIN. Litecoin oficiální stránky [online]. [cit. 2018-12-12].
Dostupné z WWW: https://litecoin.org

REACTIVEX. Oficiální stránky RxJavy [online]. [cit. 2019-02-10].
Dostupné z WWW: http://reactivex.io

VSKP.VSE. Popište a porovnejte nástroje Maven a Gradle [online].
[cit. 2013-02-05]. Dostupné z WWW:
https://vskp.vse.cz/36027_popiste_aporovnejte_nastroje_maven_agradle

8 Přílohy

Aplikace se zdrojovými kódy je dostupná v přiloženém zip souboru.