FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Platforma pro algoritmické obchodování
Student:	Bc. Jan Kirchner
Vedoucí:	Ing. Michal Šoch, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

Seznamte se s problematikou algoritmického obchodování na světových burzách a s možnostmi, které na trhu existují.
Následně navrhněte, naimplementujte a otestujte vlastní webovou platformu pro algoritmické obchodování na burzách. Aplikace bude rozhraním mezi burzami a dalšími zdroji dat relevantními pro rozhodnutí o nákupu/prodeji a samotnými programy zodpovědnými za vydání takových příkazů (tzv. strategie).

Aplikace nechť má následující funkcionality:
1: umožnit uživatelům snadno sledovat portfolio rozmístěné na několika burzách,
2: zajišťovat exkluzivní přidělení prostředků jednotlivým strategiím,
3: vykonávat příkazy k nákupu/prodeji s reálnými prostředky,
4: umožnit testování na reálných současných i minulých datech,
5: vhodně vizualizovat vývoj portfolia v čase,
6: poskytovat možnost strategie spravovat - především zastavit například při neočekávaném vývoji.

Návrh a implementace nechť jsou provedeny tak, aby umožňovaly snadnou implementaci dalších datových zdrojů.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 11. června 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Aplikace pro algoritmické obchodování

Bc. Jan Kirchner

Katedra softwarového inženýrství
Vedoucí práce: Ing. Michal Šoch, Ph.D.

7. května 2019

Poděkování

Rád bych poděkoval panu Ing. Michalu Šochovi, Ph.D. za cenné rady v průběhu vedení práce. Díky ale patří všem, kteří mě během studia podporovali a dostali tak daleko.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. května 2019 .....

Odkaz na tuto práci

Kirchner, Jan Kirchner. *Aplikace pro algoritmické obchodování.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a implementací platformy sloužící pro správu strategií operujících na kryptoměnových trzích a hodnocením jejich vývoje. Kromě poskytnutí sjednoceného rozhraní pro obchodování na několika burzách nabídne aplikace různorodé datové zdroje a také možnost testovat programy na historických datech, nebo za pomoci simulovaných prostředků. Pro vývoj byly použity především technologie .NET, React a Docker pro následné nasazení v kontejnerech.

Klíčová slova Algoritmické obchodování, .NET, React, Docker, kryptoměny, SPA

Abstract

This diploma thesis deals with analysis, design and implementation of platform for administration of strategies operating on cryptocurrency markets and evaluation of their results. In addition to providing a unified multi-exchange trading interface, the application will offer diverse information sources as well as the ability to test programs on historical data, or with simulated assets. For development technologies such as .NET, React and Docker for following deployment in containers were used.

Keywords Algorithmic trading, .NET, React, Docker, cryptocurrencies, SPA

Obsah

Úvod **1**

1 Teorie algoritmického obchodování a cíle práce **3**
1.1 Obchodování obecně 3
1.2 Algoritmické obchodování 4
1.3 Cíle práce 5

2 Analýza **7**
2.1 Konkurenční řešení 7
2.2 Analýza požadavků 8
2.3 Model případů užití 11
2.4 Analýza API rozhraní kryptoměnových burz 17

3 Návrh aplikace **21**
3.1 Architektura systému a popis komponent 21
3.2 Datový model 25
3.3 Návrh uživatelského rozhraní 27

4 Implementace a nasazení **31**
4.1 Implementace serverové části 31
4.2 Implementace klientské části 37
4.3 Dockerizace a nasazení aplikace 42

5 Testování **47**
5.1 Unit testy 47
5.2 Heuristická analýza 49
5.3 Uživatelské testování 50

Závěr **55**
Další rozvoj 55

Ukázka aplikace 56

Literatura **59**

A Seznam použitých zkratek **63**

B Dockerfile **65**

C Obsah přiloženého CD **67**

Seznam obrázků

11 Ukázka grafu hloubky trhu 4

21 Diagram případů užití – strategie 12
22 Diagram případů užití – uživatel 13

31 Architektura systému 22
32 Autentizace pomocí JWT tokenu 25
33 Datový model – Market Data Provider 26
34 Datový model – Master Data Manager 27
35 Wireframe – Detail portfolia 30
36 Wireframe – Detail strategie 30

41 Utilizace RAM 43
42 Docker versus Virtualizace 44

51 Snímek obrazovky aplikace – detail portfolia 57
52 Snímek obrazovky aplikace – detail strategie 57

Seznam tabulek

21 Pokrytí případů užití - uživatel 16
22 Pokrytí případů užití - strategie 16

51 Nielsenova heuristická analýza – hodnocení 51

Úvod

Automatizace se dnes týká každého myslitelného odvětví. Stroje se umí rychleji a přesněji rozhodovat, nedělají chyby, nepodléhají tlaku okolí ani stresu a mohou pracovat neustále bez toho, aniž by se kdy unavily. Ne jinak je tomu na poli obchodování. Už dávno se netýká pouze lidí snažících se zanalyzovat všechny dostupné relevantní informace a na jejich základě provést rozhodnutí o nákupu nebo prodeji. I zde platí, že správně využitá výpočetní síla dokáže ušetřit spoustu námahy a pomoci dobrat se přesnějších výsledků za zlomek času. Nejedná se ovšem o univerzální a všemocné řešení, jde pouze o nástroj. A ten se, jak už je pro ně běžné, dá použít dobrým i špatným způsobem.

V případě kryptoměn a pro ně uzpůsobených burz platí poznámka o rychlosti a neúnavnosti dvojnásobně. Obchoduje se neustále a volatilita je v porovnání s tradičnějšími aktivy mnohem větší – výjimkami nejsou ani pohyby cen v řádech stovek procent denně. I z těchto důvodů je speciálně v tomto prostředí algoritmické obchodování, jak se využití aplikací pro tyto účely říká, stále více a více populární.

I přes lehkost s jakou programy dokáží zprocesovat množství dat, které by člověk jenom četl dlouhé měsíce, se nemusí rozhodovat správně. Vždy totiž platilo a platit bude, že trh je do jisté míry nepředvídatelný a i malá chyba může majitele přijít draho. V takovém případě na minimalizaci rizik spojených s algoritmickým obchodováním nejvíc platí jediný osvědčený lék – důkladné testování a analýza výsledků. A platforma, jež by tyto úkoly měla usnadnit a přidat i řadu funkcionalit navíc, je i cílem této diplomové práce.

Kapitola 1

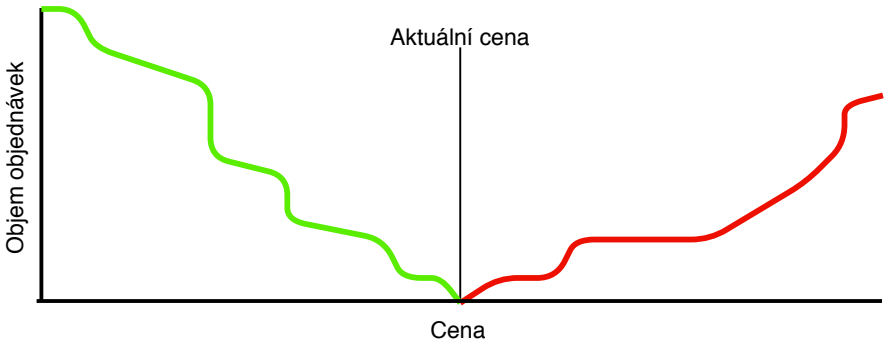
Teorie algoritmického obchodování a cíle práce

Dříve, než bude plně objasněn cíl této práce, je třeba seznámit se se zásadami algoritmického obchodování. Nejenom s obecně platnými principy, ale také s terminologií typickou pro toto odvětví. Některá vysvětlení budou oproti realitě částečně zjednodušená, pro účely výsledné aplikace však plně dostačující.

1.1 Obchodování obecně

Obchod na burze není nic jiného než obyčejná směna tak,
jak ji známe třeba z kamenné prodejny. Obchodují spolu vždy dvě strany (nakupující a prodávající) a to za účelem dosažení zisku. Hlavním rozdílem oproti zmiňovanému příkladu je ten, že cena není nikdy pevně daná. Neustále se hýbe, a proto rozkazy, které uživatel posílá, odpovídají spíše objednávkám na nákup/prodej určitého množství měny. Bod, kde se setkává nejvyšší nabídka na nákup spolu s nejnižší nabídkou na prodej, udává momentální cenu. Všechny objednávky buď na nákup s nižší než aktuální cenou, nebo na prodej s cenou vyšší, jsou označovány jako otevřené pozice. Pokud by se jejich kumulovaný objem vynesl na osu Y, dostali bychom graf hloubky trhu, který dokáže o současné situaci prozradit více, než jen pouhá informace o ceně. Ukázka, jak může graf hloubky trhu vypadat, je znázorněná na obrázku 11

V místě aktuální ceny dochází k samotnému uzavírání obchodů. To je také hlavní funkce burzy. Jelikož všechny měny má v držení právě ona, představuje takový moment jenom přepsání daných částek z jednoho účtu na druhý – stejně jako je tomu například při posílání peněz v rámci jedné banky. Jedná se také o chvíli, kdy je inkasována provize. Ty jsou pro burzy primárním zdrojem příjmů.



Obrázek 11: Ukázka grafu hloubky trhu

1.2 Algoritmické obchodování

Algoritmické obchodování označuje způsob, kdy je za rozhodování o provedení nákupu nebo prodeje zodpovědný počítačový program. Výhody takového přístupu jsou zjevné – minimalizuje se riziko, snižují se náklady a zvyšují šance na zisk. Ani tak však úspěch není zdaleka zaručen – proměnných je příliš mnoho a vývoj trhu se nikdy nedá s jistotou předpovídat.

Strategie, jak se těmto specializovaným aplikacím říká, mohou být založeny na mnoha principech. Zároveň mohou spadat do různých kategorií například podle toho, jak často obchodují. Zde je důležitý pojem vysokofrekvenční obchodování, který, jak je již podle názvu patrné, zahrnuje strategie provádějící velké množství obchodů během krátkého času. Běžně se označuje zkratkou HFT.

Již zmiňovanými principy, na kterých může být fungování strategie založeno, jsou například [1]:

- **Sledování trendu** – jedním ze základních přístupů při vytváření strategií. Program se na základě současných i historických cen snaží odhadnout budoucí trend a podle toho provádět nákupy/prodeje. Velkou roli při rozhodování hrají různé technické indikátory, na jejichž hodnotách je většinou celá logika založená.
- **Arbitráž** – arbitráží se rozumí situace, kdy strategie využije různé ceny na několika trzích. Tím, kromě profitování přímo úměrnému rozdílu cen, dochází k jejímu vyrovnání.
- **Analýza sentimentu** – je přístup, kdy strategie sbírají a analyzují data o náladě investorů daného trhu. Jedná se totiž o silný ukazatel toho, jakým směrem se cena bude vyvíjet.
- **Sledování novinek** – obchodování je založeno na informacích. A nepočítají se jen ty technické, ale i ty z prostředí vedení společností.

Jediná zpráva tak do velké míry může ovlivnit cenu. Tento fakt je ještě umocněn ve světě kryptoměn známém svou volatilitou a také tím, že se na něm pohybuje řada lidí majících v komunitě velkou pozornost. S tímto faktem můžou pracovat i strategie a využít jej ke svému zisku.

- **Návrat k průměru** – strategie založené na tomto principu pracují s předpokladem, že trhy se většinu času pohybují v určitém rozmezí. Kdykoliv se z něj vychýlí pak může strategie počítat s tím, že se cena dřív nebo později opět vrátí.

Dalším způsobem, jak strategie rozdělit, je podle toho, s jakými daty a prostředky obchodují. Možné jsou tři varianty

- **Trading** – v práci dále označované jako real trading je klasické obchodování s reálnými prostředky nad aktuálními daty.

- **Backtesting** – jako backtesting je označovaný mód, ve kterém strategie dostává historická data o vývoji na trhu. Díky tomu je možné během zlomku času otestovat, jak by si vedla.

- **Paper trading** – často také označovaný jako Forward Testing, je způsob obchodování, kdy strategie pracuje s reálnými údaji o trhu, ale obchody, které provádí, jsou pouze fiktivní stejně jako prostředky, k jejichž směně dochází. Výhodou je, že simulace je opravdu přesná a teoreticky by se její výsledky neměly lišit od obyčejného obchodování – ovšem s nulovým rizikem. Hlavní nevýhodou oproti backtestingu je nemožnost strategii otestovat v jediném momentu.

1.3 Cíle práce

Cílem práce je vytvořit aplikaci, která bude mezivrstvou mezi programy zodpovědnými za rozhodování o nákupu nebo prodeji a samotnými burzami. Díky tomu bude možné dosáhnout řady efektů, jež umožní snadnější testování strategií a analýzu jejich výsledků. Těmito efekty se rozumí především:

- **Jednotné rozhraní** – strategie nemusí zohledňovat na jaké burze realizuje obchod. Rozhraní je pro všechny podporované burzy sjednocené, takže se uživatel může soustředit jen na to skutečně podstatné – kdy, za jakou cenu a jaký objem měny nakoupit nebo prodat.

- **Exkluzivní přiřazení prostředků** – aplikace ve své podstatě vytváří virtuální účty v rámci jednoho skutečného, který na burze existuje. Umí tak zaručit exklusivní přiřazení prostředků přesně jedné strategii. Ty tak nemají možnost se navzájem jakkoliv ovlivňovat.

- **Testování s fiktivními prostředky** – krom výše zmíněných výhod přidává aplikace možnosti testování strategií s fiktivními prostředky. A to jak na současných datech, tak na těch historických. Uživatel tak může strategie vyzkoušet aniž by riskoval finanční ztrátu. Snahou je pak simulovat práci s fiktivními prostředky co možná nejvěrněji, aby šance, že takto vyzkoušené strategie uspějí i na opravdové burze, byly co možná největší.

- **Vizualizace vývoje strategií a portfolia** – problémem při obchodování na burze je fakt, že seznam provedených obchodů nedá uživateli moc informací o tom, jak hodnota v historii rostla a klesala. Tato aplikace bude umět vývoj vizualizovat nejen v rámci celého portfolia, ale i v rámci jednotlivých strategií. Díky této zpětné vazbě je strategie snadnější analyzovat a v případě potřeby vylepšovat.

- **Zdroje dat a indikátory na jednom místě** – stejně jako je sjednocené rozhraní burz benefitem pro uživatele, i všechny zdroje dat dostupné na jednom místě dokáže ušetřit čas a energii. I tuto schopnost tak aplikace bude mít.

Rovněž je třeba podotknout, pro jaké účely aplikace není určená a jaké funkce nebude poskytovat.

- **Vysokofrekvenční obchodování** – vysokofrekvenční obchodování je nad rámec této práce. Vyžaduje jiný přístup k řešení problému a má mnohem vyšší systémové nároky.

- **Tvorba strategií** – cílem aplikace je především testování hotových strategií a poskytnutí jednotného rozhraní pro ostrý běh. Nemá za cíl fungující strategie nabízet ani participovat na jejich vývoji jinak, než poskytováním informací o jimi dosažených výsledcích.

Kapitola 2

Analýza

2.1 Konkurenční řešení

Na poli algoritmického obchodování panuje poměrně silná konkurence. Existuje sice celá řada služeb nabízející různé možnosti algoritmického obchodování, ale většinou se jedná buď o komplexní platformy poskytující mnohem více možností, než je pro splnění cílů této práce nutné, nebo se jedná o až příliš automatizovaný software mířící i na nezkušené uživatele. Ten pak neposkytuje dostatečný prostor pro tvorbu vlastních postupů a strategií. Zde je popis několika služeb vybraných ke srovnání:

- **TradingView[2]** – silný nástroj pro testování strategií představuje společnost TradingView. Ta se primárně zaměřuje na poskytování grafů (všech myslitelných trhů – nejen kryptoměnových) a s tím spojených služeb. Velkou publicitu pak získává i díky možnosti prezentovat představy uživatelů o budoucím vývoji trhu. Ti nejúspěšnější se tak můžou těšit velké prestiži uvnitř komunity obchodníků a s tím spojených benefitů. Hlavní nevýhody jsou dvě. První z nich je fakt, že služba nenabízí prostředí pro strategie obchodující s reálnými prostředky – slouží čistě na testování. Druhým bodem je nutnost strategie psát ve speciálním jazyce, kterým je Pine Script. Hlavní výhodou této služby je potom silná komunita a také nepřeberné množství veřejně přístupných strategií a indikátorů, které se dají dále rozvíjet a používat.
- **Signals[3]** – Signals je původem český startup nabízející kompletní zázemí pro tvorbu a zpeněžení obchodovacích algoritmů. Svou komplexností ovšem dalece přesahuje cíle i možnosti této diplomové práce. Podstatným rozdílem je fakt, že Signals má ambice stát se tržištěm a umožnit obchodování i se samotnými strategiemi, indikátory, zdroji dat atd. To vše navíc na základech blockchainové technologie, která sama o sobě dokáže nalákat davy lidí. I přesto, že se za pomoci ICO podařilo vybrat značnou sumu, vývoj stále není u konce a nabídka datových zdrojů nijak

neohromí. Co naopak zaujme je skvěle zpracované uživatelské prostředí, které je moderní, bohaté, ale zároveň intuitivní a nerušivé. Obecně vzato se jedná o zajímavou alternativu, ze které se dají čerpat nápady, ale nejedná se o přímou konkurenci.

- **Cryptohopper[4]** – Cryptohopper je prvním představitelem konkurenčních řešení, které uspokojí i uživatele bez programovacích schopností. Nabízí totiž jednoduché GUI umožňující zadat příkazy k nákupu/prodeji na základě změny hodnoty jednoho z nabízených indikátorů. I zde existuje možnost hotové strategie nabízet na tržišti. Oproti Signals.Network je však tato služba značně zjednodušená a také placená. V závislosti na množství strategií nebo pozic se výše měsíčního předplatného pohybuje v rozmezí 19 až 99 dolarů.

- **Gekko[5]** – Gekko je velice populární open-source nástroj, který se svými funkcemi částečně přibližuje službě, jež by měla být vyvinuta na základě této práce. Kolem Gekka se postupem času vytvořila početná komunita, která nejenže neustále vylepšuje program samotný, ale rovněž vyvíjí různé plug-iny dále rozšiřující jeho možnosti. I přesto, že se jedná o kvalitní software s řadou užitečných funkcí by se ovšem daly najít výhrady. Tou hlavní je fakt, že Gekko podporuje jen obchody s celým objemem držených prostředků a navíc jen na jednom měnovém páru. Tento přístup sice značně zjednodušuje implementaci, ale závazkem této aplikace je dát uživatelům volné ruce v tom, jak strategie postupují. Druhá výtka pak směřuje na uživatelské rozhraní, které působí zastarale a ne příliš intuitivně. To podstatně kazí dojem z jinak propracované služby, která je navíc zcela zdarma.

2.2 Analýza požadavků

Důkladná a systematická analýza požadavků je jedním ze základních předpokladů pro úspěšné dokončení jakéhokoliv projektu. Slouží pro jasné vydefinování toho, jaké funkce má systém mít a jak se má chovat.

Sesbírané požadavky jsou především výsledkem zkoumání preferencí cílové skupiny. Do značné míry jsou pak ovlivněny dokončeným rozborem konkurenčních řešení. Všechny takto nashromážděné a interpretované informace jsou rozděleny do dvou kategorií, kterými jsou funkční a nefunkční požadavky.

2.2.1 Funkční požadavky

Funkční požadavky slouží pro jasné definování veškerých funkcí, kterými by finální aplikace měla disponovat.

- **FP1 – Založení uživatelského profilu** – systém umožní zakládání nových uživatelských účtů. Registrací je podmíněné použití drtivé většiny funkcí aplikace.
- **FP2 – Správa uživatelského účtu** – aplikace umožní správu uživatelského účtu. kromě změny kontaktního emailu umožní i správu klíčů k aplikačním rozhraním podporovaných burz.
- **FP3 – Získání stavu účtů reálných burz** – aplikace umožní získání informací o účtech na podporovaných burzách, ke kterým uživatel poskytl potřebné klíče.
- **FP4 – Vytvoření a správa simulovaných prostředků** – aplikace umožní vytvoření a správu falešných prostředků, které mohou být použity pro trénink strategií nanečisto.
- **FP5 – Zobrazení stavu všech prostředků** – aplikace přehledně zobrazí seznam veškerých prostředků.
- **FP6 – Zobrazení strategií** – aplikace dokáže zobrazit seznam strategií spolu se základními informacemi o jejich průběhu.
- **FP7 – Vizualizace vývoje portfolia** – aplikace uživateli zvládne poskytnout vizualizaci vývoje celého portfolia.
- **FP8 – Založení strategie** – uživatel bude mít možnost založit novou strategii v jednom z možných obchodovacích režimů. Dále bude možné strategii přidělit určitou část volných prostředků, které budou strategii výhradně alokovány.
- **FP9 – Zobrazení detailu strategie** – uživatel bude mít možnost zobrazit detail strategie. Není podstatné, jestli strategie stále běží, nebo byl její běh už ukončen. V obou případech by aplikace měla být schopná zobrazit informace o průběhu a základní informace týkající se úspěšnosti.
- **FP10 – Zastavení strategie** – v případě nepříznivého nebo neočekávaného vývoje může uživatel strategii zastavit a uvolnit všechny prostředky, které měla strategie alokované, k dalšímu použití.
- **FP11 – Spuštění strategie s historickými daty** – Uživatel bude mít možnost spustit strategii nad historickými daty. Bude mít možnost specifikovat zdroj a dobu běhu této strategie.
- **FP12 – Získání informací o aktuálním stavu trhu** – strategie bude mít možnost získat informace o aktuálním stavu trhu.
- **FP13 – Získání informací o přidělených prostředcích strategie** – strategie bude mít možnost získat informace o tom, jaké prostředky má v držení.

- **FP14 – Zadání požadavku na nákup/prodej** – strategie bude mít možnost zadat požadavek na nákup nebo prodej určitého prostředku. Bude mít možnost stanovit cenu, na které by se obchod měl uskutečnit.
- **FP15 – Získání informací o obchodech strategie** – vzhledem k tomu, že vykonání obchodu není synchronní operace, bude mít strategie možnost získat informace o provedených obchodech a jejich stavu.
- **FP16 – Zrušení požadavku na nákup/prodej** – strategie bude mít možnost nevykonaný obchod zrušit a uvolnit tak zablokované prostředky.
- **FP17 – Získání informací o historickém stavu trhu** – strategie bude mít možnost získat informace o historickém stavu trhu.

2.2.2 Nefunkční požadavky

Nefunkční požadavky mají za úkol popsat kvalitativní stránku systému. Nesoustředí se tedy na to, jaké funkce má aplikace vykonávat, ale jak. Při jejich definování je třeba počítat s tím, že se mohou navzájem do určité míry vylučovat. Příkladem takového sporu může být požadavek na nízkou cenu řešení spolu s požadavkem na robustní zabezpečení. Je tak třeba brát v potaz i váhu konkrétních bodů a ve výsledku se snažit hledat takový kompromis, který co nejvíce vyhovuje všem uživatelům i provozovatelům systému.

- **NFP1 – Dostupnost, spolehlivost a rychlost** – aplikace musí být snadno dostupná z celého světa, musí mít pro uživatele přijatelnou dobu odezvy a být spolehlivá.
- **NFP2 – Bezpečnost** – aplikace by výhledem ke svému charakteru měla být zabezpečena tak, aby odpovídala obecně platným standardům.
- **NFP3 – Design** – aplikace musí být přehledná, snadno použitelná a rozhraní musí být přizpůsobeno i pro zobrazení na mobilních telefonech a tabletech.
- **NFP4 – Jazyk aplikace** – aplikace musí být v anglickém jazyce. Případná lokalizace v budoucnu by však neměla představovat závažnější problém
- **NFP5 – Rozšiřitelnost** – aplikace musí být navržená tak, aby se dala snadno rozšiřovat. To platí především o dalších datových zdrojích a podporovaných burzách.
- **NFP6 – Testovatelnost** – aplikace by měla být navržena tak, aby se dala snadno otestovat.

2.3 Model případů užití

Dalším krokem důležitým pro detailní pochopení aplikace je vytvoření modelu případů užití. Jedná se o UML diagram složený z následujících částí:

- **Aktér** – člověk nebo systém, který z venčí interaguje s aplikací. Aktéři jsou také poskytovatelem nebo příjemcem dat.
- **Případ užití** – každý případ užití reprezentuje proces, na jehož konci je nějaký pozorovatelný výsledek.
- **Vztah** – spojnice, která může být buď mezi dvěma aktéry, aktérem a případem užití, popřípadě mezi dvěma případy užití.
- **Hranice systému** – hranice systému, jak je z názvu patrné, ohraničuje určitou podmnožinu případů užití a jasně tak vymezuje zodpovědnosti jednotlivých subsystémů.

2.3.1 Aktéři systému

2.3.1.1 Nepřihlášený uživatel

Vzhledem k tomu, že smysluplné používání aplikace je přímo podmíněné registrací, má nepřihlášený uživatel velmi omezené možnosti. Může se jen registrovat nebo přihlásit.

2.3.1.2 Přihlášený uživatel

Přihlášený uživatel má naopak řadu možností.

2.3.1.3 Strategie

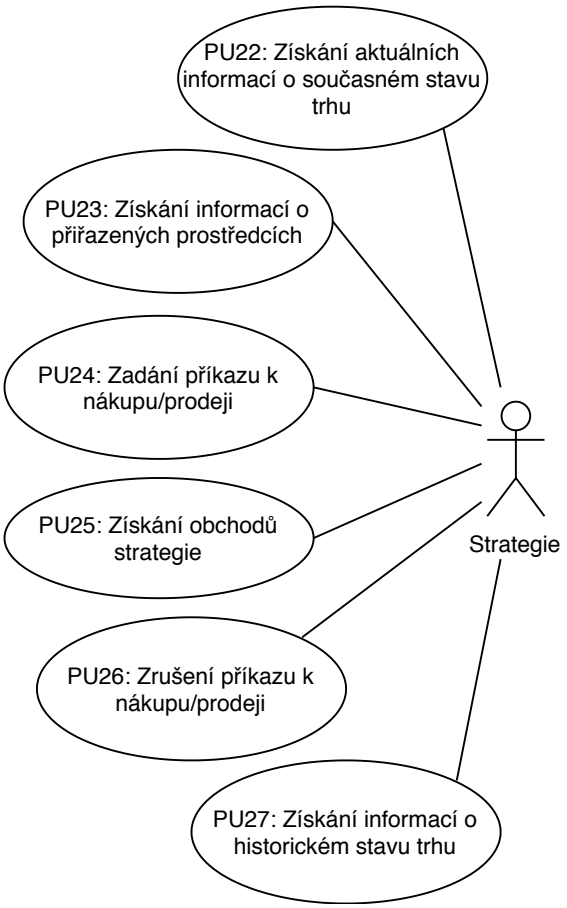
V modelech bude vystupovat i samotná strategie. Ta interaguje se systémem a jde tak o aktéra neuživatelského typu.

2.3.2 Diagram případů užití

V případě této aplikace byl pro větší přehlednost diagram rozdělen na dvě části. Na obrázku 21 jsou tak vidět případy užití spojené s aktérem strategie. Další obrázek 22 pak reprezentuje případy užití týkající se přihlášeného a nepřihlášeného uživatele. Takto vytvořené diagramy pomáhají především pochopit vztahy mezi aktéry a systémem jako takovým.

Následuje seznam samotných případů užití.

- **PU1 – Registrace uživatele** – aplikace umožní uživateli založit si s pomocí emailové adresy uživatelský účet.



Obrázek 21: Diagram případů užití – strategie

- **PU2 – Přihlášení uživatele** – registrovaný uživatel bude mít možnost se přihlásit.
- **PU3 – Odhlášení uživatele** – přihlášený uživatel bude mít možnost se odhlásit.
- **PU4 – Změna kontaktní emailové adresy** – uživatel bude mít možnost změnit svou kontaktní emailovou adresu.
- **PU5 – Změna přihlašovacího hesla** – uživatel bude mít možnost změnit své přihlašovací heslo.
- **PU6 – Vytvoření API klíče** – ke každé z podporovaných burz bude mít uživatel možnost uložit svůj osobní API klíč a tajemství, díky nimž bude aplikace autorizovaná zjistit stav účtu a vykonávat obchody.

Obrázek 22: Diagram případů užití – uživatel

- **PU7 – Odstranění API klíče** – již uložené API klíče a tajemství bude mít uživatel možnost kdykoliv odstranit. V případě, že na dané burze existují prostředky alokované běžící strategií bude uživatel varován.
- **PU8 – Změna API klíče** – uložené burzovní API klíče bude možné kdykoliv nahradit novými.
- **PU9 – Stažení stavu účtu z reálné burzy** – aplikace na příkaz uživatele získá informace o stavu jeho účtu na burze. Takto získané informace o prostředcích aplikace uloží a umožní s nimi dále nakládat.
- **PU10 – Přidání volných simulovaných prostředků** – aplikace nabídne možnost vytvoření simulovaných prostředků, se kterými

následně mohou obchodovat strategie v obchodovacím módu Paper Trading. Až na možnost s těmito prostředky libovolně nakládat by uživatel neměl poznat rozdíl oproti prostředkům reálným.

- **PU11 – Změna stavu volných simulovaných prostředků** – stav již uložených volných prostředků může uživatel libovolně měnit. Ať již přidávat nebo naopak ubírat.

- **PU12 – Odstranění volných simulovaných prostředků** – uložené prostředky, které nejsou alokované žádné běžící strategii bude možné ze systému smazat.

- **PU13 – Zobrazení všech prostředků** – všechny prostředky, které aplikace eviduje, budou přehledně zobrazené.

- **PU14 – Zobrazení detailu portfolia** – uživatel bude mít možnost zobrazit detail portfolia, díky čemuž získá základní informace o současné hodnotě a také vývoje hodnoty v historii.

- **PU15 – Zobrazení seznamu strategií** – aplikace uživateli umožní zobrazit seznam strategií.

- **PU16 – Seřazení seznamu strategií podle atributu** – strategie, ať už se jedná o ty běžící, nebo již zastavené, bude možné filtrovat podle řady atributů. Jsou jimi:

  – datum zahájení (základní nastavení).

  – profit (počítaný buď jako rozdíl počáteční a konečné hodnoty v případě již zastavených strategií, nebo jako rozdíl počáteční a současné hodnoty v případě běžících strategií).

  – počet obchodů.

  Dále bude možné třídit výpis primárně podle toho, jestli chce uživatel vidět pouze buď běžící, nebo zastavené strategie. Na takto vyfiltrované záznamy pak bude možné vztáhnout výše zmíněná kritéria řazení.

- **PU17 – Založení nové strategie** – aplikace umožní uživateli založit novou strategii buď s reálnými, nebo simulovanými prostředky. Uživatel po vyvolání akce musí mít možnost strategii alokovat libovolné množství volných prostředků. S těmi pak strategie bude libovolně obchodovat za účelem dosažení zisku.

- **PU18 – Zobrazení detailu strategie** – aplikace umožní uživateli zobrazit detail strategie. Kromě základních údajů o době běhu bude rovněž zobrazen historický vývoj hodnoty.

- **PU19 – Zobrazení obchodů strategie** – uživateli bude mít možnost si v aplikace zobrazit seznam obchodů dané strategie. Ty budou řazené podle data uzavření a nabídnou základní informace o obchodovaném páru, objemu, ceně, stavu a dat otevření a uzavření.

- **PU20 – Zastavení strategie** – uživatel bude mít možnost strategii kdykoliv a z jakéhokoliv důvodu zastavit. V tu chvíli by aplikace měla zrušit neuzavřené obchody (pokud takové jsou) a uvolnit strategií alokované prostředky k dalšímu použití.

- **PU21 – Spuštění strategie nad historickými daty** – uživatel bude mít možnost spustit strategii v módu, kdy bude strategie získávat historická data z burzy.

- **PU22 – Získání aktuálních informací o současném stavu trhu** – aplikace umožní strategiím získávat aktuální data o současné hodnotě trhu. Bude se jednat především o ceny a objemu kryptoměn, ale dále i statistiky o burzách, upozornění na blížící se události spojené s danou měnou, atd. Jakákoliv data využitelná pro rozhodnutí o provedení obchodu jsou žádoucí.

- **PU23 – Získání informací o přiřazených prostředcích** – aplikace strategiím poskytne seznam jim přiřazených prostředků pro obchodování.

- **PU24 – Zadání příkazu k nákupu/prodeji** – aplikace umožní strategii zaslat požadavek na nákup/prodej měny. Bude umožněn i požadavek bez uvedené směnné ceny. V takovém případě by obchod měl proběhnout za cenu, na které se trh aktuálně nachází. Ve chvíli, kdy je požadavek zadán, musí dojít k zablokování prostředků nutných pro nákup/prodej o daném objemu. V opačném případě by strategie mohly zadat nákupy za prostředky, kterými by později v době vykonání již nemusely disponovat.

- **PU25 – Získání obchodů strategie** – aplikace strategiím poskytne seznam jimi vykonaných obchodů včetně stavu, ve kterém se nachází.

- **PU26 – Zrušení příkazu k nákupu/prodeji** – aplikace umožní, aby odeslaná objednávka na nákup/prodej měny mohla být v případě, že samotný obchod ještě neproběhl, zrušena.

- **PU27 – Získání informací o historickém stavu trhu** – aplikace umožní strategiím spuštěným nad historickými daty dostávat informace o minulém stavu trhu. Počet datových zdrojů bude ovšem omezen.

	FP1	FP2	FP3	FP4	FP5	FP6	FP7	FP8	FP9	FP10	FP11
PU1	✓										
PU2	✓	✓									
PU3	✓	✓									
PU4		✓									
PU5		✓									
PU6		✓									
PU7		✓									
PU8		✓									
PU9			✓								
PU10				✓							
PU11				✓							
PU12				✓							
PU13					✓						
PU14							✓				
PU15						✓	✓				
PU16						✓	✓				
PU17								✓			
PU18									✓		
PU19									✓		
PU20										✓	
PU21											✓

Tabulka 21: Pokrytí případů užití - uživatel

	FP13	FP14	FP15	FP16	FP17
PU22	✓				
PU23		✓			
PU24			✓		
PU25				✓	
PU26					✓

Tabulka 22: Pokrytí případů užití - strategie

2.3.3 Pokrytí případů užití

Pro potřeby analýzy je ideální dostat se do stavu, kdy každý funkční požadavek je pokryt alespoň jedním případem užití. Stejně jako v případě diagramu jsou i tabulky pokrytí rozděleny pro přehlednost na dva případy. Tabulka 21 je určena pro případy užití navázané pro strategii a tabulka 22 pro případy užití navázané na uživatele.

2.4 Analýza API rozhraní kryptoměnových burz

Z analýzy požadavků a modelu případů užití vzešel set funkcí, které musejí být implementovány v aplikačním rozhraní každé burzy (jeho existence a možnost jej využívat aplikacemi třetích stran, jako je právě tato, je automaticky předpokládána), která může být do systému zapojena.
Jedná se o následující funkce:

- Získání informací o obchodovaných měnových párech.
- Získání aktuálních cen a objemů obchodovaných párů.
- Získání stavu účtu.
- Zadání objednávky na nákup/prodej.
- Zrušení objednávky na nákup/prodej.
- Získání stavu objednávek uživatele.

Pokud by burza funkce neimplementovala, nebylo by možné získat potřebná data a tím pádem naplnit požadavky.

Další funkce aplikačních rozhraní burz, kterou systém bude využívat, je endpoint poskytující historická data. Ta budou využita pro možnost Backtestingu strategií, jež je jedním z cílů. Nutno ovšem podotknout dvě věci. První z nich je fakt, že aplikace nutně nemusí nabízet možnost tohoto obchodovacího módu pro všechny podporované burzy. Nasazení této funkcionality pro alespoň jednu s dostatečně silným jménem (mezi těmi, které byly vybrány k analýze a které budou dále zmíněny, by tomuto měly odpovídat všechny), by pro potřeby testování mělo stačit většině uživatelů. Druhým faktem je, že historická data se dají získat i z jiných zdrojů, než přímo z dané burzy. Ani tak se ale nedá vyhnout problémům, které by z takového řešení plynuly. Nabízí se totiž buď objednání služby třetí strany, které většinou nebývá (alespoň u důvěryhodných poskytovatelů) zdarma, nebo jednorázové stažení historických dat v souboru. V druhém případě by to znamenalo nutnost živé ceny ukládat do databáze. I když se tak v případě dat přímo z burzy nejedná o jediné řešení, je určitě tím nejsnadnějším. To bude zohledněno při určení těch, které budou podporovány mezi prvními.

V prvním kole bylo vybráno pět populárních burz, jejichž zapojení do systému by vzhledem k široké uživatelské základně přicházelo v potaz.
V následujícím výčtu je kromě základního popisu i informace o tom, jestli implementují všechny z potřebných funkcí a jestli (popřípadě s jakými omezeními) poskytují historická data.

- **Binance[6]** – podle objemů se jedná o největší světovou burzu založenou v červenci roku 2017. Podle dostupných informací se na ní v tuto chvíli obchoduje 164 měn na 496 trzích. Krom toho, že podporuje všechny

potřebné funkce, poskytuje i veškerá historická data všech obchodovaných párů. Maximum svíček získaných v jednom dotazu je 1000 a kromě nezbytných hodnot obsahují i informace o počtu obchodů a objemu přepočteném na obě měny z páru.

- **Bittrex**[7] – další populární burzou, na které se obchoduje 262 měn na 342 trzích, je Bittrex. Založena byla v roce 2014 a těší se dobré reputaci především díky bezpečnosti. Z požadovaného setu funkcí burza podporuje všechny. Není ovšem vhodným poskytovatelem historických dat. Nějaká sice poskytuje, ale čím kratší je interval svíčky, tím méně do minulosti se lze dostat. Pro pětiminutové svíčky pak API poskytuje data jen zhruba dva týdny nazpět, což pro účely aplikace není dostatečné. Testovaný endpoint navíc není zahrnut v oficiální dokumentaci API, používají ho však grafy webového klienta Bittrex.

- **Kraken**[8] – jednou ze starších burz, které se stále těší velké popularitě, je Kraken. Byla založena ve Spojených státech amerických v roce 2011. Z potřebných funkcí jsou sice dostupné všechny, avšak, stejně jako v případě Bittrexu, v dokumentaci není zmínka o endpointu poskytujícím historická data. Rovněž množství obchodovaných měn není příliš velké – jedná se o 20 měn na 73 trzích.

- **Poloniex**[9] – Poloniex je jednou ze starších amerických burz. Založena byla v roce 2014 a v roce 2018 byla koupena společností Circle, čímž vstoupila do povědomí mnoha lidí. Circle je totiž startup vlastněný bankou Goldman Sachs a toto spojení tak značí určitou stahu klasických institucí adaptovat se na nové technologie, popřípadě jich využít ke splnění svých obchodních cílů. Na burze se obchoduje s 65 měnami na 121 trzích. Povinné funkce jsou v API implementovány a dokonce se jedná o ideálního poskytovatele historických cen. I když je počet svíček, které se vrací v odpovědi na jeden dotaz, je určitým způsobem limitován (v API dokumentaci tento limit není uveden), při testování nebyl problém získat i více než 20000 záznamů.

Na základě požadavků a shromážděným informacím bude aplikace v první fázi vývoje podporovat burzy dvě – Binance a Poloniex. První z nich je zvolena především kvůli své popularitě, počtu měnových párů a obchodovaným objemům, Poloniex pak především kvůli možnosti snadno získat historická data.

2.4.1 Data o stavu trhu

Počítá se, že primárním zdrojem dat pro běžící strategie budou data získaná z burzy. V tomto směru lze čerpat tři typy informací.

- **Tiky** – současná cena, za kterou proběhl poslední obchod na daném měnovém páru.
- **Svíčky** – (anglicky candles/klines) je typ dat vyjadřující pohyb ceny během určitého časového období. Nejkratší rámec je většinou 1 minuta. Každý záznam identifikovaný pomocí časové značky, kdy daná svíčka začala, musí obsahovat minimálně 4 čísla reprezentující:
  1. cenu při otevření,
  2. cenu při zavření,
  3. nejvyšší cenu za dané období,
  4. nejnižší cenu za dané období,
- **Hloubka trhu** – informace o tom, jaký je objem otevřených pozic na různých cenových hladinách. Jejich relevance je ovšem na kryptoměnových trzích menší, než na klasických burzách. Především díky neomezené možnosti uživatelů otevírat falešné pozice a obratem je rušit v případě, že by hrozilo jejich naplnění. Tímto způsobem ovlivní tento zdroj dat, jehož výpovědní hodnota zákonitě klesá.

2.4.1.1 Zabezpečené dotazy

Zabezpečení je doménou, kterou kryptoměnové burzy rozhodně nepodceňují. Vzhledem k povaze aktiv může i malá chyba skončit nenávratnou ztrátou – a to jak samotných prostředků, tak reputace zasažené společnosti. I proto se endpointy, které burzy vystavují, dělí na ty zabezpečené a nezabezpečené. Podobně se sice chová většina internetových aplikací, ale v tomto případě je vše o něco důmyslnější. Dobrou zprávou ovšem je, že všechny výše zmíněné burzy používají stejný model, díky čemuž bude implementace snadnější.

Pro získání dat ze zabezpečených endpointů je třeba postupovat podle následujících bodů. Jedná se o zjednodušený popis, který ignoruje případné rozdíly mezi jednotlivými burzami. Ty, i když implementují stejný model, se mohou v určitých parametrech nepatrně lišit.

1. Přihlášení uživatele na burze a vygenerování API klíče a API tajemství.
2. Vytvoření URL reprezentující daný HTTP dotaz.
3. Přiložení API klíče k dotazu. Jedná se tak o jeden z parametrů volání.
4. Vytvoření podpisu výše zmíněného řetězce pomocí HMAC-SHA512 algoritmu. Klíčem je API tajemství.
5. Přiložení podpisu k již vytvořené URL stejně jako tomu bylo u API klíče.

Kombinace API klíče a tajemství má v přeneseném významu podobné vlastnosti jako uživatelské jméno a heslo. Takto podepsané dotazy se dají ověřit vzhledem k tomu, že pouze burza a uživatel sám znají danou kombinaci.

Kapitola 3

Návrh aplikace

3.1 Architektura systému a popis komponent

Navržená architektura systému je zachycená na diagramu 31. Jak je z obrázku zřejmé, jedná se o klasický model klient-server, kde backendová část je rozdělena na dvě samostatné služby.

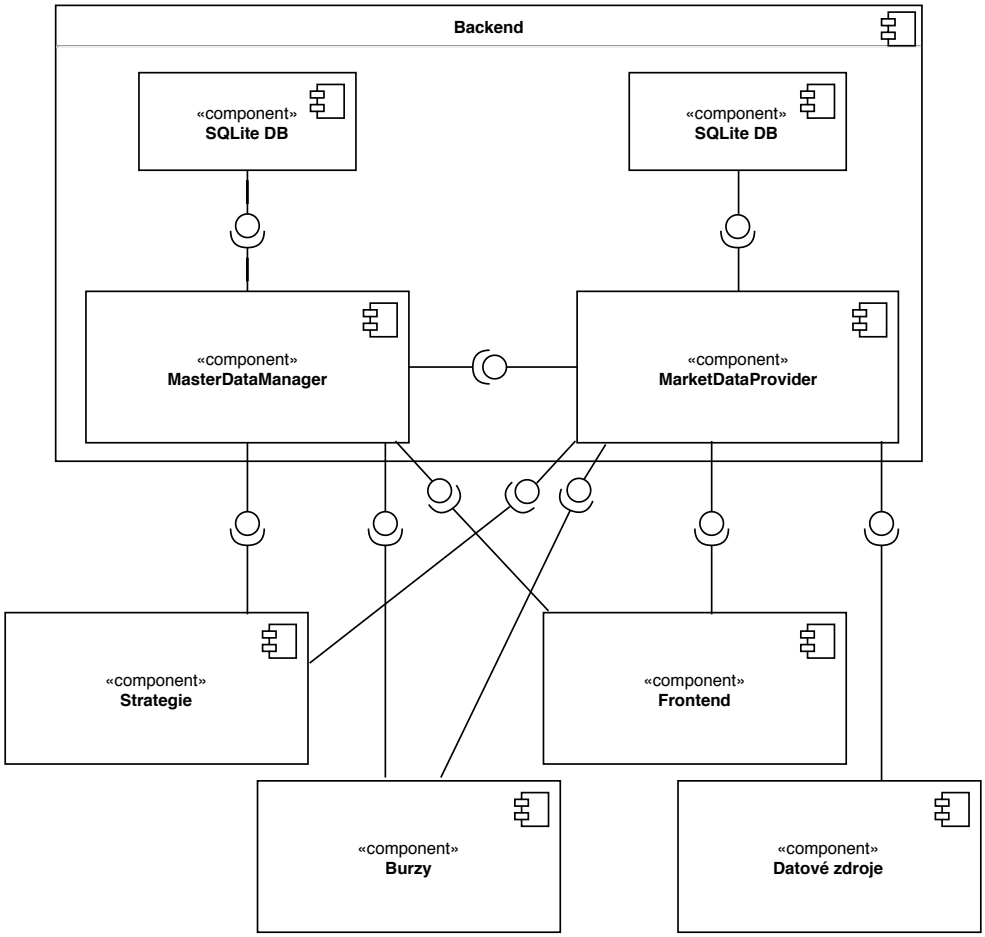
Jedním z původních plánů bylo aplikaci implementovat na základech micro-service architektury. Tento záměr byl sice opuštěn, ale snaha separovat od sebe logiku, která může fungovat samostatně, zůstala. Tímto přístupem se dá získat řada benefitů. Kromě lepší udržitelnosti kódu také možnost implementovat každou část v jiném jazyce. To ale týká především větších týmů, kde tato cesta může usnadnit nábor potřebných lidí. Nutno je ovšem zmínit i nevýhody. Mezi ty se řadí například náročnější nasazení nebo zvýšení systémových nároků, protože dvě současně běžící aplikace mají vyšší nároky než jedna – i kdyby množství funkcionalit bylo zachováno[10]. V tomto případě však tento fakt nebyl považován za hrozbu. Jaké jsou funkce jednotlivých subsystémů, jakým způsobem komunikují a jaký mají vztah k ostatním částem je rozvedeno v následujících sekcích.

3.1.1 Komponenty systému

3.1.1.1 Market Data Provider

Market Data Provider je komponenta úzce specializovaná na poskytování informací o trhu – a to jak aktuálních, tak těch historických. Konzumentem těchto dat jsou v první řadě běžící strategie, které na jejich základě provádějí nákupy nebo prodeje, ale i klientská část aplikace.

Hlavním cílem bylo vytvořit službu poskytující relevantní informace, která by ale současně byla výpočetně nenáročná a umírněná v množství dotazů poslaných na vybrané datové zdroje. Ty totiž často limitují jejich počet z jedné IP adresy. Překročení této hranice by tak vedlo k dočasné ztrátě těchto dat, což by pro mnohé strategie mohlo být kritické.



Obrázek 31: Architektura systému

Jelikož se jedná o samostatnou službu, má i svou vlastní SQLite databázi. Ta obsahuje informace o podporovaných burzách, na nich obchodovaných měnách a další záznamy spíše organizačního rázu. Jedním z charakterů těchto dat je, že se v průběhu času moc nemění.

Komunikace se službou bude umožněna pomocí aplikačního rozhraní postaveném na HTTP protokolu.

3.1.1.2 Master Data Manager

Master Data Manager představuje službu zodpovědnou za většinu samotné aplikační logiky. Komunikovat s ní bude klient a všechny běžící strategie. Toto spojení bude zajištěno veřejným aplikačním rozhraním postaveném opět na HTTP protokolu. Persistence dat pak bude zajištěna za pomoci vlastní relační databáze. Hlavní funkce, které služba musí zajistit, jsou především:

- Registrace a správa uživatelů.
- Uchování uživatelských dat – především kontaktních informací a API klíčů.
- Vytváření, správa a periodické ohodnocování běžících strategií.
- Správa prostředků a zajištění jejich exkluzivního přiřazení strategiím.
- Rozhraní pro strategie – exekuce obchodů a jejich záznam.
- Komunikace s reálnými burzami ohledně dat uživatele a jejich synchronizace – především stavu prostředků a obchodů.

I když s burzami komunikuje i výše zmíněná služba Market Data Provider, v tomto případě se bude jednat především o zabezpečené endpointy týkající se uživatelských dat.

3.1.1.3 Klient

Klient představuje prezentační vrstvu aplikace běžící v internetovém prohlížeči uživatele. Komponenta bude naprogramovaná v JavaScriptu za pomoci knihovny React a s backendovou částí aplikace (oběma službami) bude komunikovat přes jimi vystavené aplikační rozhraní.

3.1.1.4 Strategie

Strategie jsou další komponentou, kterou je v rámci architektury třeba zohlednit. Za jejich vytvoření i běh jsou zodpovědní sami uživatelé a na provoz této aplikace nemá vliv ani to, jak jsou naprogramované, ani na jakém serveru jsou fyzicky spuštěné. Komunikaci s oběma backendovými službami je zajištěná pomocí HTTP rozhraní.

3.1.1.5 Burzy a datová zdroje

Burzy a datová zdroje jsou dalšími komponentami, se kterými systém bude komunikovat. Rozdílem mezi nimi je fakt, že burzy budou data jak poskytovat tak přijímat – budou na ně například mířit žádosti o otevření obchodu s reálnými prostředky. Oproti tomu datové zdroje budou figurovat jen jako poskytovatelé informací. Není tak nutné, aby s nimi komunikovala i služba Master Data Manager.

Počítá se s tím, že počet i tak datových zdrojů bude postupem narůstat. I proto není dopředu známý způsob, jak bude probíhat komunikace. Z doposud prověřovaných služeb, které by mohly být do systému zapojeny, ale všechny poskytovaly veřejné aplikační rozhraní postavené na REST architektuře. Jelikož se jedná o v současné době obecně uznávaný webový standard, nepředpokládá se, že by komunikace i v budoucnosti probíhala jinak.

3.1.2 Zabezpečení komunikace

3.1.2.1 Šifrování komunikace

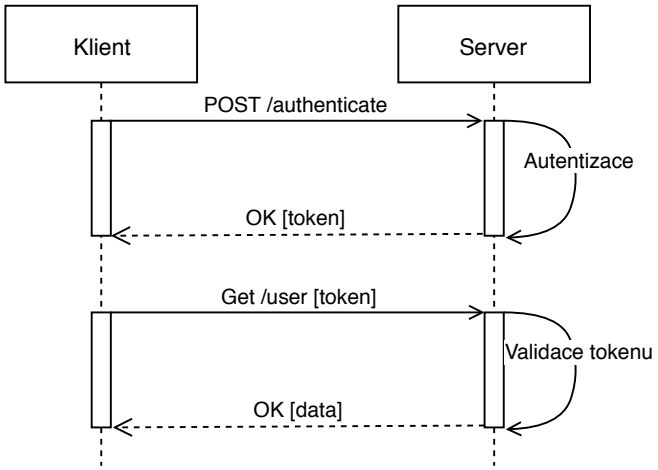
Vzhledem k tomu, že v případě protokolu HTTP nedochází k šifrování komunikace, existuje nebezpečí, že informace putující mezi jednotlivými službami, může útočník přečíst. To platí především ve vztahu backendových služeb a zbytku systému, protože v ten moment dotazy putují přes veřejnou síť, kde případní útočníci mohou operovat. Tento fakt je kritický především u login metody, kde je nutné zajistit, aby přihlašovací jméno a heslo nešlo snadno zjistit. Možností, jak těmto hrozbám zamezit, je šifrování komunikace. K tomuto účelu byl protokol HTTP rozšířen a jeho zabezpečená verze – HTTPS – bude použita na všech kanálech. Konkrétní zajištění využívání tohoto protokolu bude vysvětleno v kapitole 4.3.2.

3.1.2.2 Autentizace a autorizace

Druhým bod zabezpečení spočívá v autentizaci a autorizaci uživatelů a jejich následných dotazů přicházejích na server. Pro tyto účely bude využito JWT tokenů, které se především u moderních SPA staly standardem nahrazujícím vývojově starší metody založené na cookies. Princip fungování je jednoduchý. Uživatel, který se úspěšně přihlásí a dokáže tak, že je skutečným vlastníkům účtu, obdrží v odpovědi ze serveru samotný JWT token. Ten se následně přidá do speciální hlavičky ke každému dotazu, kdy je třeba prokázat identitu nebo právo vyvolat určitou akci. Příkladem může být snaha získat seznam běžících strategií a informacích o nich. V ten moment token slouží pro autentizaci – prokázání identity. Server na jeho základě vybere strategie vlastněné daným uživatelem a vrátí je v odpovědi. Druhým příkladem, kdy je token použit pro autorizaci dotazu, je moment, kdy danou akci může vykonat jen uživatel s administrátorskými právy. I tento druh informací (uživatelovy role) totiž token klasicky obsahuje.

Pro výše popsaný přístup je nutné zajistit, že se data obsažená v tokenu nedají měnit. V opačném případě by bylo snadné se vydávat za jiného uživatele nebo administrátora systému a buď neprávem získávat data, která by měla být soukromá, nebo ještě v horším případě provádět akce, které mohou mít vliv na celý systém.

Aby to nebylo možné, vytvoří server z informací obsažených v tokenu hash. Vstupem algoritmu je kromě nich ještě tajný textový řetězec, ke kterému by nikdo neměl mít přístup. Výsledek se následně stane součástí tokenu a je díky němu snadné ověřit to, že data odpovídají těm původním. Pokud by totiž k jejich změně došlo, nová hash by neodpovídala té obsažené v tokenu. Zároveň, vzhledem k tomu, že server jako jediný zná tajemství, s jehož pomocí je hash vytvořená, nemá útočník možnost ji podvrhnout. Hlavní výhodou oproti cookies je fakt, že ověření podepsaných informací může proběhnout rychle a bez nutnosti volat databázi.



Obrázek 32: Autentizace pomocí JWT tokenu

Podobného způsobu podepisování dotazů využívají i burzy.
Celý proces je vyobrazen na diagramu 32

3.2 Datový model

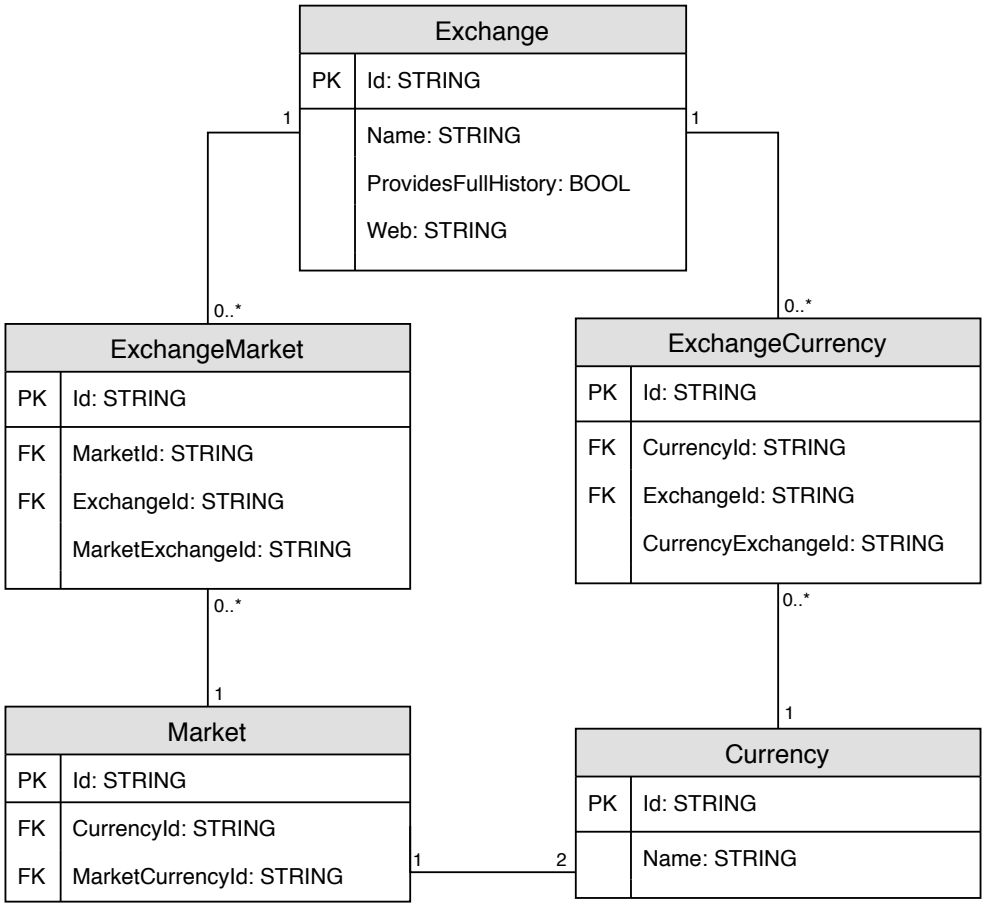
Datové modely jsou uvedeny rovnou dva. Jedná se o výsledek navržené architektury blíže popsané v předchozí kapitole.

3.2.1 Market Data Provider

Datový model pro službu Market Data Provider je vidět na obrázku 33 Jelikož je služba poměrně úzce specializovaná a z pohledu databáze slouží především pro sjednocení informací o burzách a na nich obchodovaných měnových párech, není ani model zvlášť složitý. Důležité jsou především dvě vazební tabulky

- ExchangeMarket
- ExchangeCurrency

Ty kromě informace o tom, jestli daný měnový pár/měna existuje na dané burze, poskytují i překlad. Díky tomu bude možné uvnitř systému používat jednotné značení. Příkladem problému, který tento přístup řeší, je situace, kdy směnit Bitcoin za Litecoin znamená na burze Binance poslat objednávku na trh identifikovaný kódem BTCLTC, zatímco na burze Kraken LTCXBT.



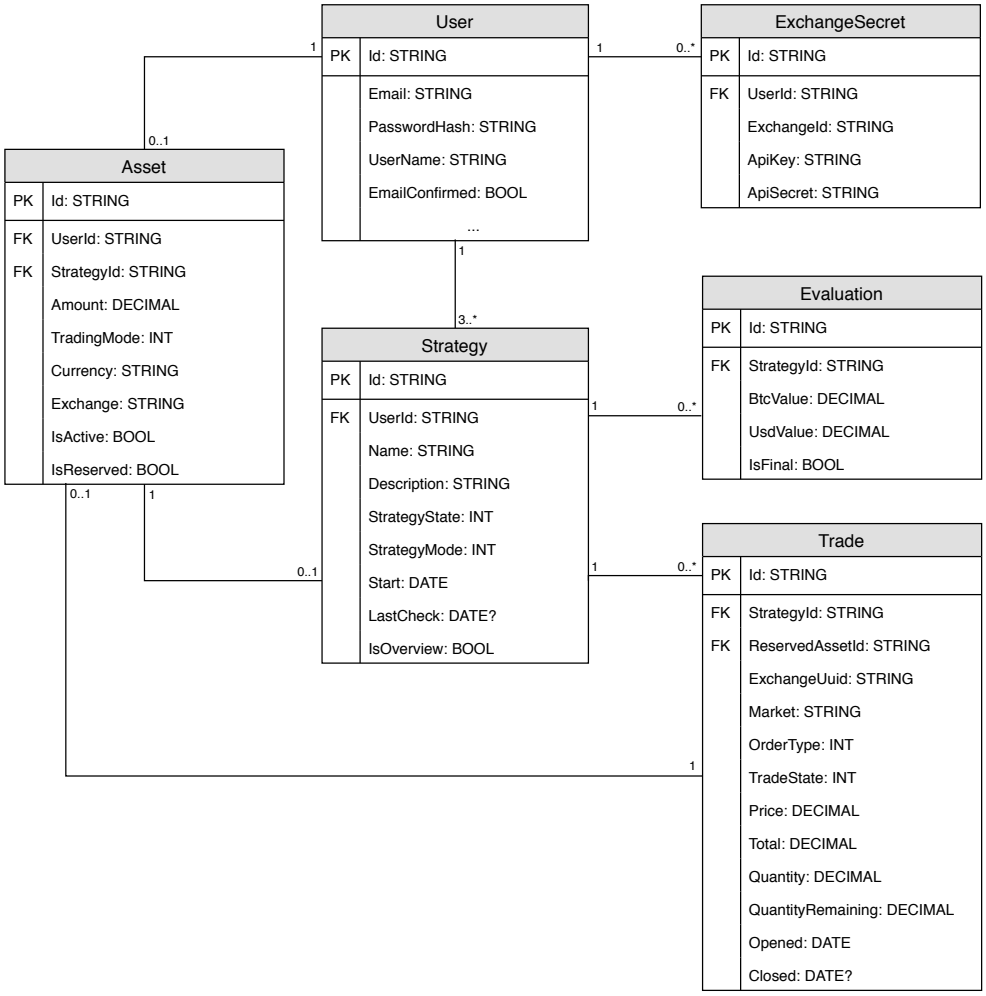
Obrázek 33: Datový model – Market Data Provider

3.2.2 Master Data Manager

Datový model této backendové služby je vidět na obrázku 34. Jak ja zřejmé, oproti prvnímu případu je o něco složitější. Vzhledem k povaze služby, která má za úkol většinu aplikační logiku, to ovšem není nijak překvapivé.

Za detailnější pohled stojí především objekt Asset. Těmi jsou reprezentovány všechny prostředky, které mají uživatelé v držení. V moment, kdy při založení strategie dojde k přiřazení Assetu, je objekt zkopírován. Z původního se odečte suma, o kterou uživatel zažádal. Novému objektu je pak stejná částka nastavena a k tomu ještě přidělena vazba na zakládanou strategii. Tímto mechanismem je zajištěno, že součet hodnot zůstává stejný jako před touto akcí a v systému tak správně zůstává suma nezměněná. Dalším důsledkem je, že všechny objekty této třídy bez přiřazené strategie ie jsou volné a mohou být přiděleny.

Podobným způsobem se řeší moment, kdy dojde k otevření pozice. Aby



Obrázek 34: Datový model – Master Data Manager

totiž strategie nemohla prostředky použít dvakrát, je daná částka okamžitě zablokována a následně s ní nejde obchodovat (pokud nedojde ke zrušení obchodu). Stejným způsobem tento problém řewší samotné burzy

U entity Asset dále stojí za zmínku bool hodnota IsActive. Ve chvíli, kdy dojde k ukončení strategie, se všechny přiřazené Assety zkopírují a nastaví se jim atribut IsActiv na false. Tím dojde k uchování informace o tom, s jakými prostředky strategie disponovala na konci svého fungování.

3.3 Návrh uživatelského rozhraní

Hezké a snadno použitelné uživatelské rozhraní je zásadní pro případný úspěch aplikace. Především v dnešní době, kdy jsou uživatelé doslova zavaleni

nabídkami všemožných služeb a můžou si tak vybírat, je intuitivní a funkční design nutností.

Při návrhu je dobré držet se zavedených a prověřených principů, které minimalizují šanci na vznik chyb. Podle jedné z mnoha dohledatelných metodik, zní pravidla následovně [11]:

- **Odražení mentálního modelu uživatele, ne designéra** – Návrh by měl být v první řadě zaměřen na uživatele. Ti sice budou mít jiné druhy očekávání a zkušeností než designér, ale aplikace má sloužit právě jim.
- **Použití reálných analogií a metafor** – Design by měl používat takové procesy a jejich analogie, které jsou uživatelům dobře známé – například z desktopové platformy.
- **Dodržování očekávání, zvyků, rutin a stereotypů** – Systém by měl používat rozšířené postupy a nevytvářet zbytečně nic nového, co by uživatelům nemuselo být známé. Myšleny jsou například barvy, kdy zelená je typicky použitá v případě, kdy je vše v pořádku.
- **Kompatibilita akce a odezvy** – Odezvy by měly být kompatibilní s vyvolanými akcemi a reflektovat tím, že spolu souvisejí.
- **Neviditelné části systému by měly být viditelné** – I když mnohé procesy v systému nejsou vidět, je dobré je uživatelům graficky znázornit – díky tomu se s aplikací naučí rychleji pracovat.
- **Poskytnutí vhodné zpětné vazby** – Uživatel by měl být neustále informován o tom, co se v systému právě odehrává.
- **Vyhýbání se zbytečnému a irelevantnímu** – Věci, které uživatel v danou chvíli nemusí vidět, by ani neměly být zobrazovány.
- **Konzistence návrhu** – Design by měl být v rámci celého systému konzistentní. Notně to zvyšuje jeho čitelnost.
- **Poskytnutí nápovědy** – Uživatelům by měl být k dispozici dostatečně návodný manuál poskytující základní informace pro používání systému.
- **Podpora pro nováčky i zkušené uživatele** – Design by měl reflektovat různou úroveň uživatelů. Nováčci by neměli být omezování nedostatečnou znalostí systému, zatímco zkušení uživatelé by měli mít možnost systém používat maximálně efektivně.

Rovněž byly při návrhu zohledněny znalosti z magisterského předmětu Návrh uživatelského rozhraní [12] přednášeném na Fakultě informačních technologií Českého vysokého učení technického v Praze.

3.3.1 Wireframes

Prvním krokem při návrhu uživatelského rozhraní je vytvoření wireframes. Jedná se o jednoduché vyobrazení základních prvků stránky a jejich rozložení. Na základě takto vytvořených modelů je pak mnohem snazší hledat případné nedostatky a kontrolovat, jestli jsou splněny všechny požadavky.

Velkou výhodou je rychlost a jednoduchost, s jakou je lze vytvořit. Bohatě totiž postačí tužka a papír. Alternativou jsou pak různé online nástroje, kterých existuje nespočet – v tomto případě bylo využito služby MockFlow. Wireframes by měly být primárně zaměřeny na kritické sekce a nemusejí pokrývat celou aplikaci. V rámci této práce budou zobrazovat dvě podstránky – detail portfolia a detail zvolené strategie.

3.3.2 Detail portfolia

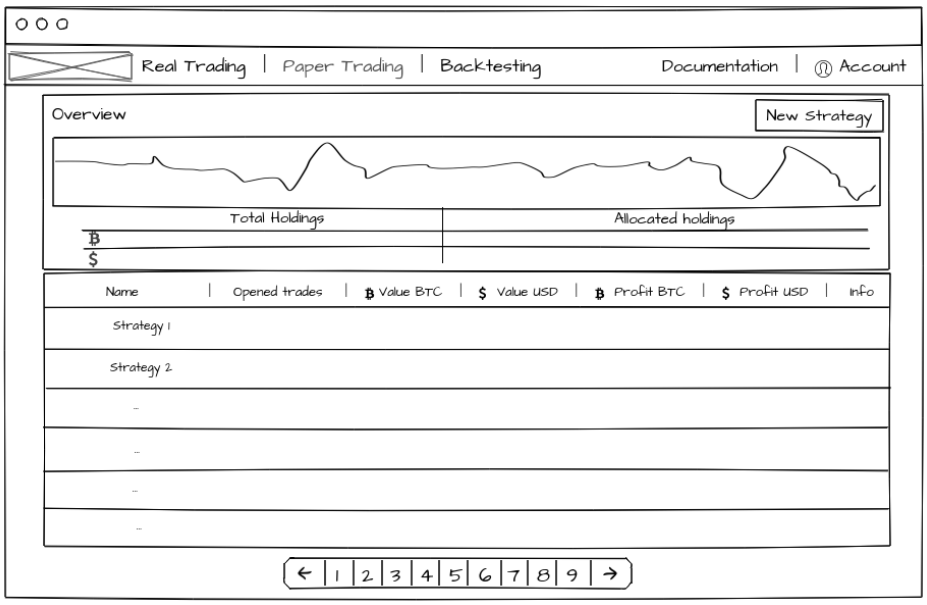
Portfoliem se rozumí všechny prostředky daného obchodovacího módu nehledě na to, jaké strategii jsou přiřazeny. Podstránky tedy budou tři pro každý z nabízených obchodovacích módů:

- Real Trading
- Paper trading
- Backtesting

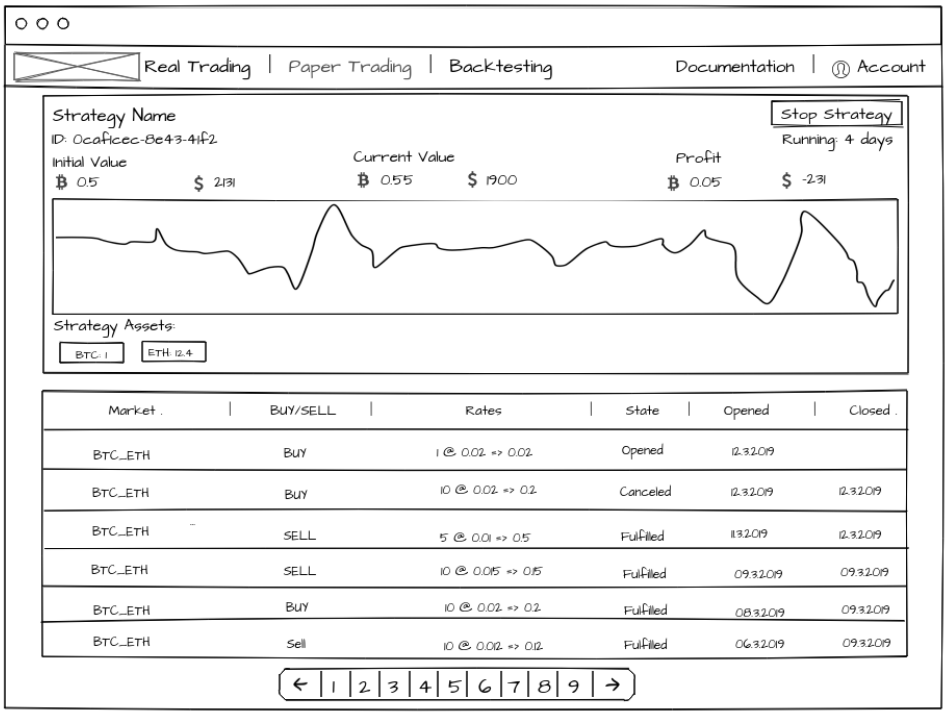
Přecházet se mezi nimi bude dát pomocí odkazů v hlavní navigaci a samy od sebe by se měly lišit nepatrně. Kromě vizualizace vývoje portfolia je funkcí stránky také zobrazit seznam strategií a možnost založit novou. Wireframe je vidět na obrázku 36.

3.3.3 Detail strategie

Při kliknutí na jednu z vyobrazených strategií (popřípadě při příchodu na příslušnou URL) je zobrazen její detail. Ten, kromě vizualizace vývoje hodnoty v čase, má za funkci zobrazit seznam obchodů a možnost strategii zastavit.



Obrázek 35: Wireframe – Detail portfolia



Obrázek 36: Wireframe – Detail strategie

Kapitola 4

Implementace a nasazení

Tato kapitola bude věnována převážně informacím týkajících se samotného vývoje aplikace a jejího následného nasazení na testovací server.
Budou popsány použité technologie spolu s jejich případnými výhodami a nevýhodami. Rovněž budou zmíněny specifické problémy, které v průběhu implementace nastaly, a jejich možné řešení. Cílem není rozebrat každý detail technického řešení, jako spíš upozornění na konkrétní aspekty, které nemusejí být na první pohled zřejmé.

4.1 Implementace serverové části

4.1.1 Použité Technologie

Jako primární technologie schopná naplnit veškeré požadavky vzešlé z analytické a návrhové části byl zvolen framework .NET Core, konkrétně jeho součást uzpůsobená pro tvorbu webových aplikací – ASP.NET Core. První verze byla veřejnosti představena v roce 2016 a hlavním rozdílem oproti svému předchůdci, frameworku .NET, je způsob vývoje, který je nově open-source, a možnost spuštění aplikací na všech hlavních operačních systémech[13]. Technologie je tedy nově multiplatformní. Konkrétním programovacím jazykem z rodiny .NET Core byl pro obě serverové části aplikace zvolen jazyk C#.

Kvůli požadavku na persistenci dat bylo pro potřeby vývoje rozhodnuto o použití technologie SQLite[14]. Jde o relační databázi s velmi malými nároky na systémové prostředky. Zprovoznění je prakticky okamžité a nevyžaduje žádnou konfiguraci. Navíc vzhledem k tomu, že jde o technologii postavenou na souborech, je snadné s databází manipulovat. Jedná se o minimalistické řešení, které u větších aplikací nemusí být dostatečné. Není ovšem třeba se podobného scénáře obávat. Pokud by k němu došlo, díky objektově relačnímu mapovači rovněž použitém v rámci této aplikace, nepředstavuje výměna databázové technologie větší problém. Tento ORM s názvem Entity Framework, který odděluje aplikační logiku od databáze a je zodpovědný za tvorbu sa-

motných SQL dotazů, pochází stejně jako celý .NET Core z dílen Microsoftu a těší se mezi vývojáři velké popularitě. Spolu s využitím dalšího jazyka .NET Core frameworku, kterým je LINQ, je programátor dokonale odstíněn od konkrétního databázového řešení, výsledkem čehož k datům může přistupovat stejně, jako kdyby se jednalo o kolekce v paměti programu. Tento přístup bude dále rozebrán a názorně předveden v jedné z ukázek.

4.1.2 Použité Nástroje

Pro vývoj aplikace byly použity následující nástroje:

- **Visual Studio[15]** – pro implementaci backendové části aplikace bylo použito IDE nabízené přímo společností Microsoft. Pro jazyky rodiny .NET jde o jedno z nejpoužívanějších prostředí pro vývoj.
- **Visual Studio Code[16]** – odlehčenou verzí výše zmíněného IDE, vyznačující se jednoduchostí, přehledností a stabilitou, je Visual Studio Code. To bylo použito pro implementaci klientské části aplikace.
- **DB Browser for SQLite[17]** – jako průzkumník databáze byl zvolen oficiálně podporovaný program DB Browser for SQLite.
- **Insomnia[18]** – pro snadnější testování HTTP dotazů často přišel vhod program Insomnia, který slouží jako jednoduchý REST klient.
- **GitHub[19]** – pro správu verzí.

Všechny zmíněné programy byly instalovány na operačním systému macOS Mojave verze 10.14.3.

4.1.3 Problémy a jejich řešení

4.1.3.1 Datové repozitáře

Jednou ze zásad psaní čistého kódu je princip DRY – don't repeat yourself spočívající v tom, že aplikační logika by se neměla nikde opakovat[20]. Datové repozitáře jsou ideálním místem, kde jej uvést do praxe. Existuje totiž set operací, které je třeba provádět nad všemi druhy entit. Jsou čtyři:

- Create
- Read
- Update
- Delete

Zkráceně CRUD. Tohoto faktu využívá návrhový vzor Repository, který je v aplikaci použit. Základní rozhraní každého repozitáře je popsáno třídou IRepository. Jak může vypadat ukazuje následující ukázka.

```
public interface IRepository<T> where T : IdEntity
{
  T GetById(string id);
  IEnumerable<T> List();
  IEnumerable<T> List(Expression<Func<T, bool>> predicate);
  void Add(T entity);
  void Delete(T entity);
  void Edit(T entity);
  void Save();
}
```

Jak je z ukázky zřejmé, rozhraní je generické – lze použít pro jakoukoliv entitu, která dědí od jednoduché abstraktní třídy IdEntity. Tím je zajištěno, že objekty mají jednoznačný identifikátor. Třída IdEntity je vidět na následující ukázce.

```
public abstract class IdEntity
{
  public string Id { get; set; }
}
```

Samotná implementace třídy Repository je příliš obsáhlá na to, aby zde byla uvedena celá. Pro představu ovšem plně poslouží ukázka rozhraní spolu s jednou implementovanou metodou.

```
public class Repository<T> :
  IRepository<T> where T : IdEntity
{
  protected readonly MasterDataContext _dbContext;
  public void Add(T entity)
  {
    _dbContext.Set<T>().Add(entity);
    _dbContext.SaveChanges();
  }
}
```

Poslední ukázkou je samotný datový repozitář. V IStrategyRepository se dají přidat další metody specifické pro entitu Strategy, které nejsou zahrnuty v základním IRepository rozhraní.

```
public class StrategyRepository :
  Repository<Strategy>, IStrategyRepository
{
  public StrategyRepository(MasterDataContext dbContext) :
    base(dbContext){}
}
```

Jak je vidět, samotný databázový kontext je repositáři předán v konstruktoru. Díky tomu je možná inicializace za pomoci vkládání závislostí. Tento fakt ve velké míře usnadňuje testovatelnost aplikace, která je jedním z požadavků.

4.1.3.2 Repetitivní úkoly

Obě backendové aplikace potřebují repetitivně provádět určitý druh činností. Zatímco v případě Market Data Provideru je to kvůli nutnosti neustále získávat aktuální data, u Master Data Manageru je to například kvůli ohodnocování strategií nebo kvůli kontrole stavu otevřených obchodů. Jedním způsobem, jak podobného výsledku dosáhnout, by bylo využít unixové technologie Cron, která dokáže plánovat úkoly a v definovaný čas spustit akci. Touto akcí by mohlo být vytvoření HTTP dotazu a jeho odeslání na připravený endpoint, který by již zařídil zbytek. Jako elegantnější řešení se ukázal způsob, kdy jsou po startu automaticky zaregistrované úkoly, které se asynchronně provádějí na pozadí. Ty jsou navíc plně koordinované s životním cyklem aplikace, takže jsou přirozeně spuštěny ve chvíli, kdy je program připraven, a ukončeny ve chvíli, kdy program končí. Pro potřeby těchto opakovaně vykonávaných služeb byla vytvořena abstraktní třída HostedService.cs[21]. Její implementace je vidět na následující ukázce.

```
public abstract class HostedService : IHostedService
{
  private Task _executingTask;
  private CancellationTokenSource _cts;

  public Task StartAsync(CancellationToken cancellationToken)
  {
    _cts = CancellationTokenSource
      .CreateLinkedTokenSource(cancellationToken);
    _executingTask = ExecuteAsync(_cts.Token);

    return _executingTask.IsCompleted ?
      _executingTask : Task.CompletedTask;
  }

  public async Task StopAsync
    (CancellationToken cancellationToken)
  {
    if (_executingTask == null) return;
    _cts.Cancel();
    await Task.WhenAny
      (_executingTask, Task.Delay(-1, cancellationToken));

    cancellationToken.ThrowIfCancellationRequested();
  }

  protected abstract Task ExecuteAsync
```

```
    (CancellationToken cancellationToken);
}
```

Výše zmíněným službám pak stačí jen podědit z HostedService.cs a přepsat metodu ExecuteAsync. Princip je naznačený v následující ukázce.

```
public class StrategyEvaluationService : HostedService
{
  protected override async Task ExecuteAsync
    (CancellationToken cancellationToken)
  {
    while (!cancellationToken.IsCancellationRequested)
    {
      ... //servisni logika
      await Task.Delay
        (TimeSpan.FromMinutes(5), cancellationToken);
    }
  }
}
```

4.1.3.3 Cachování dat

V komunikaci mezi serverem a klientem je celkem běžné, že prohlížeče ukládají mezivýsledky určitých HTTP dotazů. Poté, když se stejný dotaz opakuje, prohlížeč předpokládá, že se výsledek nezměnil, a použije data, která se vrátila v reakci na poslední volání. Je zjevné, že takto se nedá postupovat vždy. Pokud by server implementoval funkci /gettime, je jasné, že klienta bude zajímat vždy aktuální čas a cachování, jak se této metodě říká, by kvalitu služby jen zhoršovalo.

V uvedeném případě byla zodpovědnost za ukládání mezivýsledku na straně klienta. Server jenom pomocí HTTP hlaviček informoval, jak by mělo ideálně vypadat. Existují ale i případy, kdy je třeba podobného chování dosáhnout na straně backendu. Třeba v momentě, kdy se velký počet klientů (pro příklad 1000 během jedné sekundy) snaží získat stejnou informaci, u které navíc není striktně vyžadována její absolutní aktuálnost (jako například právě čas). V takový moment by bylo krajně neefektivní vyvolat při každém příchozím dotazu jakoukoliv časově nebo výkonnostně operaci, když ji stačí provést jen jednou za sekundu a zbytek času vracet tuto zapamatovanou hodnotu.

Dalším případem, kdy se podobná implementace hodí, jsou data, která se prakticky nemění – třeba výčet měnových párů obchodovaných na vybrané burze. V takovém případě může být časové okno, kdy se data načtená do paměti berou jako aktuální a nekontroluje se pravý zdroj (databáze), mnohem delší. Vždy je ovšem třeba rozumně vážit platnost cache.

Pro implementaci tohoto řešení byla použita metoda podporovaná přímo v dokumentaci ASP.NET Core [22]. Prvním krokem je registrace speciální

třídy v souboru Startup.cs, kde se normálně registrují všechny služby a repozitáře předávané skrz dependency injection kontrolerům zodpovědným za zpracování příchozích dotazů.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMemoryCache();
}
```

V objektech, které následně takto vytvořenou MemoryCache využívají, je použití následovné.

```
public class MarketDataMemCacheService
  : IMarketDataMemCacheService
{
  private readonly IServiceScopeFactory _scopeFactory;
  private IMemoryCache _memoryCache;

  public MarketDataMemCacheService(
    IServiceScopeFactory scopeFactory,
    IMemoryCache memoryCache)
  {
    _scopeFactory = scopeFactory;
    _memoryCache = memoryCache;
  }

  public ExchangeMemCache GetExchange(string exchangeId)
  {
    return _memoryCache.GetOrCreate(exchangeId, entry =>
    {
      using (var scope = _scopeFactory.CreateScope())
      {
        var exchangeRepository = scope.ServiceProvider
          .GetRequiredService<IExchangeRepository>();
        entry.SlidingExpiration = TimeSpan.FromDays(1);
        var exchange = exchangeRepository
          .GetForMemCache(exchangeId);
        return new ExchangeMemCache()
      }
    });
  }
}
```

Jak je vidět, pomocí DI jsou při inicializaci převzaty dva objekty – IMemoryCache, kterou v předchozím kole bylo nutné zaregistrovat, a IServiceScopeFactory, jež je do konstruktorů předávána defaultně bez nutnosti registrace. V metodě GetExchange pak dochází k samotnému cachovaní dat. Jejich původní zdroj je v repozitáři, tedy v databázi. Zde je nutné využít IServiceScopeFactory a vytvořit nový scope, v kterém lze s pomocí metody GetRequiredService<T> získat požadovanou službu nebo repozitář. Důležitá

je rovněž stanovení doby, během které je cache platná a na konci které opět dojde k dotazu do databáze. V tomto případě je to jeden den, ale vždy záleží na okolnostech a charakteru takto ukládaných informací.

4.2 Implementace klientské části

Jak již bylo řečeno v předchozích sekcích, pro implementaci klientské části bude použita knihovna React[23]. Výsledkem je takzvaná single page aplikace. Ta se od klasické webové služby liší tím, že při prvním navštívené URL je klientovi odeslán javascriptový program, který od té doby žije v prohlížeči vlastním životem. Při změně URL nedochází k opětovnému volání serveru. Aplikace pouze změní svůj stav tak, aby vyobrazovala přesně to, co uživatel chce. Výhodou je především značné zlepšení interakce, jelikož stránka se mění okamžitě a ne až po několika sekundách, jako tomu bylo dříve. Dalším benefitem je také snížení objemu datové komunikace, což ještě více snižuje dobu odezvy. Mezi backendem a frontendem se totiž posílají jenom data.

Jedním z problémů, které javascriptová komunita v minulosti řešila a stále řeší, je poměrně komplikované prvotní nastavení projektu.
To je způsobené velkým množstvím závislostí a pro začínající vývojáře poměrně složitou konfigurací modulu, který z jednotlivých souborů vytváří výsledný program. Existuje sice celá řada startovacích projektů, které tuto konfiguraci řeší za vývojáře, ale obecně chybí společný postup a standard. Krom toho se i tak velmi často najde něco, kvůli čemu je vývojář nucen do konfigurace zasáhnout.

V návaznosti na tento problém společnost Facebook (zodpovědná rovněž za vytvoření Reactu) představila nástroj create-react-app. Ten se těší velké popularitě především díky opravdu rychlému zahájení vývoje. Krom toho není nutná žádná konfigurace. Pro menší a středně velké projekty, které nevyžadují nějaké specifické chování, se jedná o jednoduchý a zároveň silný nástroj. V následující kapitole budou popsány především zajímavé knihovny, které zefektivňují, nebo jiným způsobem zpříjemňují vývoj. Dále budou zmíněny neobvyklé problémy spolu s jejich řešením a ukázkami kódu.

4.2.1 Použité knihovny

V sekci budou popsány pouze knihovny, které nemusejí být široce známy a nepatří do základního setu, jimiž je projekt create-react-app vybaven. Ve všech příkladech se jedná o moduly psané v javascriptu

4.2.1.1 Axios[24]

Knihovna axios je poměrně úzce zaměřená a používá se pro vytváření HTTP (respektive XMLHttp) dotazů na server. Rovněž podporuje v moderním JS široce používané sliby (anglicky promises), které zjednodušují práci

s asynchronní logikou a svým zápisem (použitím klíčového slova await před asynchronní metodou) připomínají spíše vyšší programovací jazyky. Může se používat buď přímo v prohlížeči, nebo v node.js prostředí. V prohlížečích je ovšem primárně podporovaná konkurenční knihovna fetch. Oproti té má ale axios intuitivnější aplikační rozhraní a také nabízí řadu funkcí navíc. Jako příklad se dá uvést:

- **Automatická transformace JSON dat** – v případě použití fetch API je nejdříve nutné vyčkat na vykonání asynchronního HTTP požadavku a v případě úspěchu pak opět asynchronně vyčíst data pomocí metody json(). V případě použití axios ovšem tento krok navíc odpadá. Data, o která při HTTP požadavcích primárně jde, jsou dostupná už po jednom volání, což značně zjednodušuje práci.

- **Možnost globálního nastavení** – Vzhledem k tomu, že nastavení většiny požadavků je stejné (myšlena je především
základní URL, hlavičky,..), je dobré mít příležitost nastavit tyto parametry globálně (samozřejmě s možností je přepsat v případě, že by některému z plánovaných dotazů nevyhovovaly). Axios tuto možnost nabízí, fetch ne.

- **Globální odchytávání odpovědí** – V této aplikaci se drtivá většina chybových odpovědí ze serveru řeší stejným způsobem, kterým je zobrazení chybové hlášky informující klienta o tom, co se stalo. Axios tyto situace umožňuje řešit globálně pomocí takzvaných zachytávačů (anglicky interceptors). Jedná se ve své podstatě o middleware, který (třeba i v závislosti na HTTP statusu odpovědi, nebo přítomnosti chybové hlášky v odpovědi) může reagovat a dokonce i vyvolat akce měnící Redux store. Dalším příkladem, kdy se toto řešení nabízí, je odhlášení uživatele v případě, že server vrátí odpověď se stavem 401 – Unauthorized. Jedná se o elegantnější řešení než v případě ruční implementace, které se programátor používající knihovnu fetch nevyhne.

  Další možností je obnovování autorizačního tokenu, který server může předat v hlavičce odpovědi ve chvíli, kdy starý již vypršel.

  Ukázka je uvedena v kapitole 4.2.2.2.

4.2.1.2 Reselect[25]

Jednou z hlavních zásad Reduxu je to, že interní zdroj dat aplikace by měl být vždy minimální. To znamená, že všechny přidružené informace, které je nutné uživateli zobrazit, mají být průběžně dopočítávány. Jako příklad může posloužit seznam úkolů, který kromě jednotlivých úkolů samotných bude zobrazovat i jejich počet. Pro ušetření nějakého výpočetního času by pouze stačilo inkrementovat nebo dekrementovat konstantu ve chvíli, kdy se seznam změní.

Je to ovšem proti zásadám reduxu a eventuálně by mohlo dojít k nekonzistencím. U takto jednoduchého příkladu by neměl být problém si vše pohlídat, ale to by se zhoršovalo spolu s rozšiřováním aplikace o další funkcionality. Problém je, že každá komponenta může být v průběhu své existence několikrát překreslena a to i ve chvíli, kdy se ve výsledku nijak nezmění. To u složitějších aplikací může vést k velkým nárokům na výkon, kterým se ale dá vyhnout – právě použitím knihovny Reselect. Ta derivovaná data přepočítává pouze ve chvíli, kdy se původní zdroj dat opravdu změnil (kontroluje se podle reference stejně jako v samotném Reactu). Další výhodou je dekompozice komponent a interního zdroje dat. Na případné změny aplikačního rozhraní během vývoje se tak dá mnohem snadněji a rychleji reagovat.

4.2.1.3 React Semantic UI[26]

Semantic UI je poměrně známá grafická knihovna, která je uzpůsobená i pro React. Jedná se o sadu minimalisticky pojatých komponent, které se dají snadno upravit podle vlastního designového návrhu. Hlavní výhodou je jednoduchost oproti například Material Designu, který (z důvodu dodržení standardů) vytváří mnohem složitější komponenty. Jejich úprava v případě potřeby je tím pádem o dost složitější.

4.2.1.4 VX[27]

Jedním z požadavků na aplikaci je schopnost vizualizovat vývoj portfolia (popřípadě strategie) v čase. Na podobný úkol existuje celkem široké spektrum více i méně propracovaných knihoven z nichž mnohé jsou nadstavbou nad populární D3.js sloužící právě pro vizualizaci dat. Problém této knihovny je, že je v přímém rozporu s filosofií Reactu. Je tomu tak proto, že D3.js přímo manipuluje s DOM, zatímco React DOM simuluje a až ve chvíli, kdy dojde ke změně propíše všechny potřebné změny. Oba principy se nicméně dají sloučit a výsledkem je třeba i knihovna vx.

4.2.2 Problémy a jejich řešení

4.2.2.1 Globální CSS

Ve chvíli, kdy velikost projektu překročí určitou hranici, se práce
s kaskádovými styly může stát opravdu nepříjemná. Důvod je jednoduchý. CSS styly se uplatňují v globálním rozsahu celé aplikace. Kvůli tomu může odhalení toho, co ovlivňuje vzhled určité komponenty, hraničit s detektivní prací. I když se vývojáři stačí sebevíc, tento způsob je dlouhodobě jenom těžce udržitelný. Několik neduhů, kterými design CSS trpí, může být odstraněno s pomocí preprocesorům. Jmenovat se dají například populární SASS nebo LESS. Ani ty však nejsou všemocné a neřeší ten vůbec největší problém, jímž je globální rozsah. Existuje ovšem řešení, které je navíc v nové verzi výše

zmiňovaného nástroje create-react-app přímo podporované. Je jím použití CSS Modules, s kterými se dá díky přístupu CSS-in-JS importovat soubor obsahující kaskádové styly stejně, jako by se jednalo o javascriptový modul. Všechny názvy tříd z tohoto souboru jsou pomocí unikátního prefixu mapovány lokálně. Definované třídy tak zasahují pouze tam, kde jsou výslovně vyžádány a neovlivňují tak uživatelské rozhraní nikde jinde. To umožňuje řádově snažší vývoj a testování, než v případě použití klasických CSS. Jak se s CSS moduly pracuje dokresluje následující ukázka kódu.

React komponenta

```
import React from 'react';
import styles from './styles.module.scss';

export default ({heading}) => (
  <div className={styles.heading_wrapper}>
    <h1>{heading}</h1>
  </div
)
```

CSS soubor

```
.heading_wrapper{
  text-align: center;
  & > h1{
    color: red;
  }
}
```

4.2.2.2 Globální odchytávání odpovědí ze serveru

Jak již bylo zmíněno, chybové odpovědi ze serveru se s pomocí knihovny axios dají odchytávat globálně na jednom místě. Aby ovšem bylo možné na takové události reagovat změnou stavu Redux storu, je nutné interceptorům, které jsou v zásadě middlewarem HTTP dotazů, store předat. Jak taková konfigurace vypadá v případě tohoto projektu je vidět v následující ukázce kódu.

configureStore.js

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/rootReducer';
import { createLogger } from 'redux-logger';
import promiseMiddleware from 'redux-promise-middleware';
import { setupInterceptors } from './interceptors';


export default function configureStore(initialState={}) {
  const store = createStore(
    rootReducer,
    initialState,
```

```
    applyMiddleware(
      thunk,
      promiseMiddleware,
      createLogger()
    )
  );
  setupInterceptors(store); //Axios middleware
  return store;
}
```

interceptors.js

```
import axios from 'axios';
import {
  LOGOUT,
  ALERT_ERROR
} from '../actions/types';

export const setupInterceptors =  (store) => {
  axios.interceptors.response.use(function (response) {
    return response;
  }, function (error) {
    if (!error.response || error.respose.status === 500) {
      store.dispatch({
        type: ALERT_ERROR,
        payload: ''Server error - contact support''
      });
      return Promise.reject();
    }
    if ( error.response.status === 401) {
      localStorage.clear();
      store.dispatch({ type: LOGOUT });
      store.dispatch({
        type: ALERT_ERROR,
        payload: 'Session expired'
      });
      return Promise.reject();
    }
    console.error(error.response.data);
    if(error.response.data.message){
      store.dispatch({
        type: ALERT_ERROR,
        payload: error.response.data
      });
    }
    return Promise.reject(error.response.data);
  });
};
```

4.2.2.3 Konfigurace proxy pro SPA aplikaci

Klientská část aplikace je obsluhována pomocí Nginx serveru. Jelikož webové služby postavené na Reactu jsou takzvané single page aplikace, kdy se při změně URL nevolá server (jestliže není nutné získat potřebná data), dochází k jednomu zásadnímu problému. Pokud totiž klient zadá cokoliv jiného než root adresu služby, například: www.sluzba.cz/users, Nginx v defaultním nastavení nenajde požadovaný zdroj dat a dotaz zamítne. Je proto třeba, aby na všechny dotazy, které:

- jsou určené pro klientskou aplikaci
- nejsou statický zdroj dat (typicky obrázky)

bylo odpovězeno stejně, jako by směřovaly do kořenového adresáře,
tedy www.sluzba.cz. Vzhledem k tomu, že aplikace je spouštěna v Docker kontejneru postaveném nad šablonou Nginx, jediné, co je třeba, je vytvořit konfigurační soubor, který server použije. Jak může vypadat důležitá část konfigurace je vidět na následující ukázce.

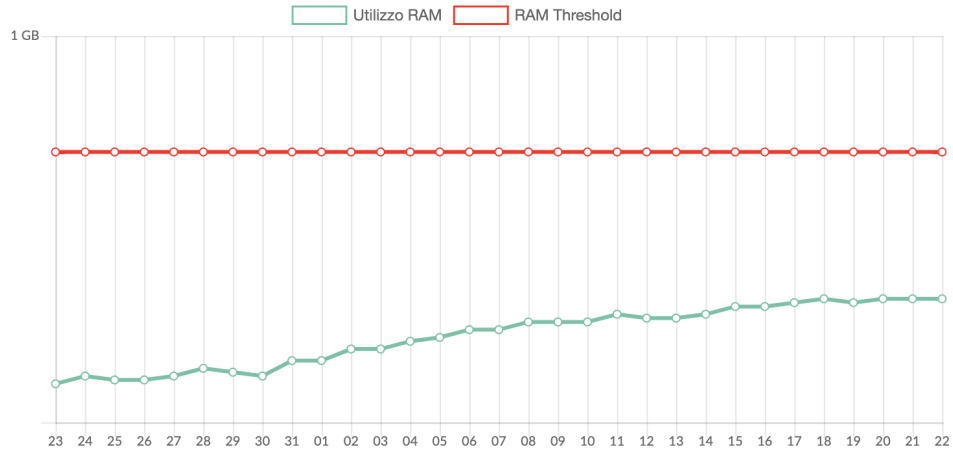
```
server {
        listen       80;
        server_name  localhost;

        location / {
                root   /usr/share/nginx/html;
                index  index.html index.htm;
                try_files $uri $uri/ /index.html;
        }
}
```

Pro fungování je nejpodstatnější direktiva try_files. Logika je taková, že pro každý požadavek, který je na klienta směřován, Nginxs nejdříve zkusí najít soubor, který adrese odpovídá. To je důležité pro statický kontent, jelikož URL www.sluzba.cz/static/images/cat.jpg by bez tohoto testu vracela kód aplikace. Pokud takový soubor nenajde, zkusí najít odpovídající složku. Jestliže ani v tomto případě neuspěje, vrátí index.html.

4.3 Dockerizace a nasazení aplikace

Pro účely testování aplikace sloužil předkonfigurovaný virtuální soukromý server od společnosti Forpsi s následujícími parametry:

- **Operační systém** : Debian 8 64bit
- **Počet procesorů** : 1 Virtual CPU
- **Operační paměť** : 1GB



Obrázek 41: Utilizace RAM

- **Místo na disku** : 20GB

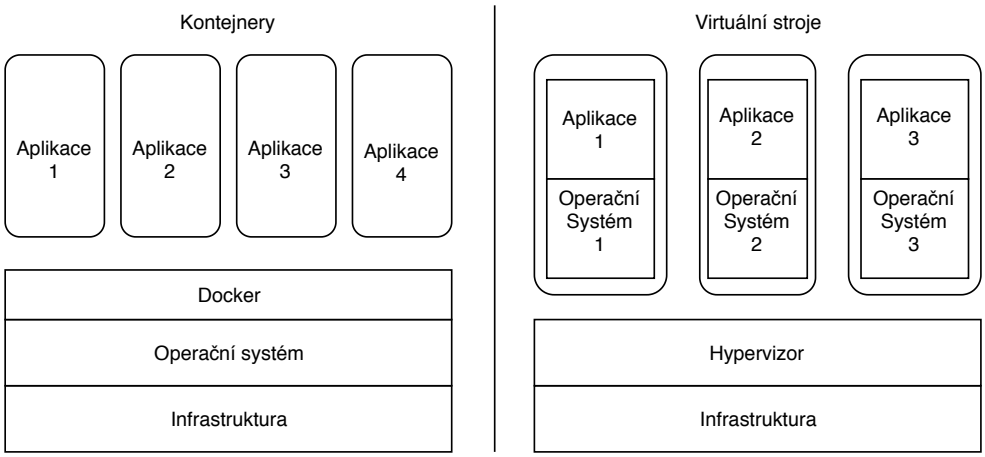
I když nebylo předem jasné, jestli daná konfigurace bude výkonnostně stačit, ukázala se sestava jako plně dostačující. Hraniční byla především velikost operační paměti. Ani u té však průměrné hodnoty vytížení málokdy překonaly hranici třiceti procent. Na obrázku 41 je vidět vytížení RAM při současném běhu dvaceti testovacích strategiích provádějících průměrně jeden obchod za hodinu.

4.3.1 Docker

Pro usnadnění vývoje a nasazení byl použit nástroj Docker. Jedná se o technologii, která dokáže ve velké míře zjednodušit celý proces vytváření aplikace a těší se značné a neustále rostoucí oblibě ve vývojářské komunitě.

Stojí na jednoduché myšlence, kdy každá aplikace (popřípadě část aplikace) je uzavřena do standardizovaného kontejneru, který kromě kódu samotného nese i informace o systému, použitých knihovnách a veškerých závislostech. Takto zabalený program pak může být spuštěn na jakémkoliv stroji, na kterém je nainstalovaný Docker bez jakékoliv další konfigurace. Vývojáři se tak mohou plně věnovat psaní výpočetní logiky a nemusí trávit čas přípravou aplikačního serveru, kde by v opačném případě museli zajistit stažení všech (v případě více programů mnohdy navzájem se ovlivňujících) závislostí. Tento benefit je pak ještě více znát ve chvíli, kdy se musí nasadit starší aplikace závislá na technologiích, jejichž verze nejsou s těmi novějšími kompatibilní. Oproti klasické virtualizaci má Docker zásadní výhodu v tom, že všechny kontejnery sdílí jedno jádro systému [28].

Nedochází tak ke zbytečnému využívání výpočetní síly několika paralelně běžícími operačními systémy, což vede k zásadnímu snížení nároků na výkon



Obrázek 42: Docker versus Virtualizace

takového řešení. I přes sdílené jádro navíc jednotlivé procesy zůstávají plně izolované. Porovnání architektury klasické virtualizace a dockerizace můžete vidět na diagramu 42.

4.3.2 Docker Compose

Ve chvílích, kdy je aplikace tvořená více službami, může být nutná určitá orchestrace. K tomuto úkolu vznikla technologie Docker Compose, která jedním příkazem zařídí vše nezbytné a sestaví dohromady kompletní systém. Konfiguruje se pomocí YAML souboru obsahující potřebné direktivy jak pro jednotlivé podslužby, tak pro další potřebné elementy. Těmi může být například síť propojující jednotlivé kontejnery, nebo sdílená úložiště. Mezi nejčastěji používané direktivy pro Docker Compose konfigurační soubor patří například:

- version – verze docker-compose
- services – výčet jednotlivých služeb tvořících dohromady celou aplikaci
  - image/build – definuje, kde se nachází šablona pro vytvoření služby
  - ports – definuje a mapuje na sebe porty hostujícího systému a samotného kontejneru.
  - depends_on – stanovení závislostí. Typickým příkladem je aplikace využívající databázový kontejner. Ten se musí vytvořit jako první. V opačném případě by závislá služba hlásila chybu.
  - environment – nastavení systémových proměnných.
  - volumes – umožňují definovat, kde a jaký způsobem kontejner ukládá svá data. Ty tak mohou být dostupná buď z hostujícího systému, nebo z jiného běžícího kontejneru.

- networks – umožňuje definovat síť skrz kterou spolu kontejnery komunikují. Ty jsou defaultně všechny propojené, ale snadno se dá nastavit jiné chování.

- volumes – definuje úložiště pro kontejnery.

Tato aplikace je složená z pěti samostatných kontejnerů. Kromě již výše zmíněných pro klientskou aplikaci a obě backendové služby, zde figurují ještě kontejnery nginx-proxy a letsencrypt-nginx-proxy-companion. Jejich funkce jsou popsány v následujících sekcích.

4.3.2.1 nginx-proxy[29]

Jedná se o kontejner zodpovědný za rozdělování požadavků přicházejících na server. Jak je z názvu patrné, je založen na populárním open-source nástroji Nginx. Velkou výhodou je, že konfigurační soubor je vytvářen automaticky. Toho je docíleno tak, že složka hostujícího systému obsahující metadata o běžících kontejnerech je skrz volume nasdílena do nginx-proxy. Kontejner rovněž sleduje změny a v případě nutnosti konfiguraci dynamicky změní. Pro správné fungování je třeba v Docker Compose souboru kontejnerům, kterým mají být požadavky směrovány, nastavit systémovou proměnnou podle které proxy rozpozná, jaké službě požadavek přeposlat. Název této proměnné je VIRTUAL_HOST.

4.3.2.2 letsencrypt-nginx-proxy-companion[30]

Zabezpečení aplikace je zásadním požadavkem vzešlým z analýzy. Jedním z pilířů je zajištění zabezpečené komunikace mezi jednotlivými komponentami. Stručně řešeno je třeba zabránit tomu, aby data, která se posílají a která mohou přijít do rukou případným útočníkům, nebylo možno přečíst – je třeba je zašifrovat. To je třeba zajistit především mezi klientskou aplikací a backendem. Je to tak proto, že zatímco backendové služby by měly být spuštěné v rámci jednoho serveru, kam by neměl mít kromě správců nikdo jiný přístup, tak komunikace mezi zmíněnými komponentami probíhá na internetu, kde mohou útočníci operovat.

Způsob, jakým se komunikace dá zabezpečit, je použití HTTPS protokolu, který v síti šifruje komunikaci dvou stran a znemožňuje tak čtení dat kýmkoliv jiným. Základním kamenem potřebným pro šifrování komunikace je takzvaný certifikát. Ten je nutné získat, ověřit, nainstalovat a poté periodicky obnovovat. Tento poměrně složitý proces se podařilo znatelně snížit certifikační autoritě Let's Encrypt, která navíc své TLS certifikáty nabízí zcela zdarma. Vzhledem k popularitě tohoto řešení vznikl i kontejner letsencrypt-nginx-proxy-companion, který, jak je z názvu patrné, rozšiřuje možnosti proxy o zabezpečenou komunikaci. Pro úspěšné spuštění aplikace je třeba pouze nastavit

kontejnerům, jež se to týká, systémové proměnné LETSENCRYPT_HOST a LETSENCRYPT_EMAIL. Vše ostatní je plně automatizované.

Výsledný Docker Compose soubor můžete vidět v příloze B

4.3.3 Spuštění aplikace

Aplikaci v lokálním prostředí, na kterém je nainstalovaný Docker, je díky výše zmíněné deklaraci možné spustit jediným příkazem.

```
$ docker-compose up
```

Ten se postará o vystavení všech služeb a ostatních nezbytností.

Kapitola 5

Testování

Testování softwaru představuje proces, kdy se za pomoci standardizovaných postupů zkoumá výsledná aplikace za účelem ověření její kvality a odhalení případných nedostatků nebo chyb oproti dokumentaci. Dalším benefitem je pak to, že s pomocí již vytvořených testů lze ověřit, že v případě rozšíření nebyla narušena již existující a správně pracující funkcionalita.

Pro testování existuje celá řada přístupů. Zkoumat se dá například to, že určitá procedura vrací zamýšlený výsledek, že uživatelské rozhraní je přesné a správně zachycuje aktuální stav programu, nebo že spolu jednotlivé komponenty aplikace dokáží komunikovat.

5.1 Unit testy

Unit testy mají za úkol ověřit správnost implementace konkrétních jednotek kódu. Těmi jsou jakékoliv samostatně testovatelné části aplikace – většinou funkce, ale není to pravidlem. Obecný postup je takový, že je vytvořen vedlejší projekt, do kterého se následně přidávají samotné testy. Ty by v ideálním případě měly po spuštění pokrýt sto procent zdrojového kódu včetně všech větví v konkrétních metodách. Není to ovšem pravidlem – důležité je především ověření kritických sekcí. Už proto, že plné pokrytí je jen velmi obtížně dosažitelné a vzhledem k pracnosti i neekonomické. Navíc ani takový stav by nedokázal zaručit dokonalou funkčnost programu. Testovány jsou totiž jen malé samostatně pracující části. To, jak jsou navzájem propojené, nebo jestli jsou jejich výsledky interpretovány a využívány tak, jak by měly být, již unit testy neodhalí.

Unit testy neslouží jen k odhalení chyb na konci vývoje, ale i ke snadnému ověření toho, že aplikační logika nebude ani v případě budoucího rozšíření nijak narušena a bude stále fungovat tak, jak má. Vzhledem k tomu, že je tento druh testování plně automatizovaný dostane programátor odpověď na tuto otázku během několika málo okamžiků.

Každý test zjednodušeně pracuje tak, že dané jednotce kódu jsou předány předpřipravené parametry a zkoumá se, jestli se předpokládaný výsledek shoduje s tím, který se opravdu vrátí. Tento proces se notně komplikuje v případě složitějších částí programu, které mají více závislostí. I z toho důvodu je nutné na testování myslet už během implementace. Existují totiž postupy a návrhové vzory, které jej buď zjednoduší, nebo dokonce umožní v případě, kdy by to jinak bylo prakticky nemožné. Myšleno je především:

- **Vkládání závislostí** – (anglicky Dependency Injection) je jedním ze základních návrhových vzorů, kdy objekt není zodpovědný za závislosti, které pro svoje fungování potřebuje – jsou mu totiž předány již při jeho vytváření. Díky tomu se snižuje provázanost mezi jednotlivými moduly a aplikace se tím pádem dá snadněji testovat a udržovat[31].
- **Mockování** – výše zmíněného návrhového vzoru je při testování často využíváno pomocí takzvaného mockování. Testované objekty totiž mohou využívat reference, jejichž návratové hodnoty implementovaných funkcí se nedají snadno ovlivnit. Vzhledem k tomu, že jsou tyto reference předávány v konstruktoru při vytváření daného objektu, je ale možné je podvrhnout a vložit takové, které by vracely předem definované výsledky. A právě tomuto procesu se říká mockování. Nejsnadnější je využít některé z rozšířených knihoven. V tomto případě je to Moq, jejíž rozhraní obsahuje vše potřebné pro základní otestování obou backendových služeb. Následuje jednoduchá ukázka demonstrující jak práci s knihovnou, tak způsob, jakým jsou mockované objekty předávané třídám pracujícím s vkládáním závislostí.

Ukázkové rozhraní, které rozšiřuje třída, která bude mockována.

```
public interface IUserRepository{
  User GetUser(string id);
}
```

Třída využívající vkládání závislostí, jejíž funkčnost bude testována.

```
public class UserService{

  private IUserRepository _userRepository

  public UserService(IUserRepository userRepository){
    _userRepository = userRepository;
  }

  public User GetUserName(string id){
    var user = _userRepository.GetUser(id);
    if(user != null) return user.name;
    return null;
  }
}
```

Samotné mockování a testování, které je v tomto případě triviální.

```
var mockRepository = Mock<IUserRepository>();
var name = "John Doe";
mockRepository.Setup(foo => foo.GetUser(It.IsAny<string>())
  .Returns(new User{Name = name});
var service = new UserService(mockRepository);
Assert.Equals(name, service.GetUserName(""));
```

Knihovna Moq samozřejmě nabízí mnohem více, než bylo naznačeno v ukázce. Celé API je popsáno v oficiální dokumentaci [32].

V případě tohoto projektu bylo implementováno několik desítek unit testů pro obě backendové služby. Na jejich základě pak bylo odhalena celá řada chyb. Ve většině případů se sice nejednalo o příliš kritické problémy, ale i tak se jedná o důkaz, že testování by se nemělo podceňovat.

5.2 Heuristická analýza

Uživatelské rozhraní je z pohledu testování specifickou disciplínou. Oproti obecným funkcionalitám, u kterých je poměrně jednoduché odhalit moment, kdy aplikace nefunguje správně, je grafická reprezentace mnohem pružnější. Dovoluje jistou míru kreativity, s čímž je spojen fakt, že testování nelze provádět exaktně.

Obecně platí, že i když design různých aplikací a služeb může být (a většinou je) specifický, vždy by se měl pohybovat ve vymezených kolejích. Je tomu tak proto, že přílišná odlišnost může uživatele mást. Jsou totiž zvyklí na určité ovládání a rozhraní, které by se chovalo jinak, by nepřijali. Rozhodně se tak nevyplatí jít proti proudu, ale naopak podporovat stejný standard, jako ostatní úspěšné služby. Ten popisuje právě Nielsenova heuristická analýza a její body jsou následující [33]:

- **NHA1 – Viditelnost stavu systému** – uživatel by měl být neustále informován o současném stavu aplikace pomocí vhodné zpětné vazby poskytnuté během snesitelné doby odezvy. Typickým příkladem jsou různé prvky informující o tom, že se čeká na odpověď serveru. Uživatel tak nemá tendenci na tlačítko klikat opakovaně aniž by věděl, jestli je akce nějakým způsobem zpracovávána.

- **NHA2 – Konzistence mezi systémem a realitou** – aplikace by měla být konzistentní s reálným světem. Měla by odrážet to, čemu uživatel rozumí. Tím se rozumí například standardizované ikony, smysluplné hlášky, kterým se dá porozumět atd.

- **NHA3 – Uživatelská kontrola a svoboda** – systém by měl umožnit uživateli, který omylem vyvolá nějakou akci, nebo si ji chce vyzkoušet, aby se mohl bez větších problémů vrátit do původního stavu. Ne každá

akce lze ovšem vrátit. V tom případě by měl být uživatel jasně informován.

- **NHA4 – Shoda s platformou a standardy** – systém by měl být konzistentní s platformou a obecnými standardy. Stejné akce by měly být i stejně pojmenovány, aby se zabránilo případným nedorozuměním.
- **NHA5 – Prevence chyb** – pokud to je možné, měla by se aplikace snažit zabránit uživateli ve vyvolání chyby. Příkladem může být tlačítko pro odeslání emailu, které se aktivuje až ve chvíli, kdy uživatel vyplní příjemce ve správném formátu.
- **NHA6 – Rozpoznání namísto vzpomínání** – aplikace by měla na aktuální stránce poskytovat veškeré informace potřebné pro rozhodnutí o další akci. Uživatel tak má jasný přehled a opět se minimalizuje riziko vyvolání nechtěné akce.
- **NHA7 – Flexibilita a efektivita** – aplikace by měly být uzpůsobeny pro uživatele s různými potřebami. Měly by být snadno pochopitelné i novým uživatelům, ale zároveň by měly poskytnout efektivní rozhraní i pro uživatele zkušené.
- **NHA8 – Estetické a minimalistický design** – aplikace by měla být jednoduchá a čistá. Uživatel by neměl být zahlcen zbytečnými informacemi ale jen tím, co skutečně potřebuje.
- **NHA9 – Pomoc uživatel poznat, pochopit a vzpamatovat se z chyb** – aplikace by měla vést uživatele tak, aby pochopil v jakém stavu se aplikace nachází a dokázal se z něj dostat. Základním kamenem jsou smysluplné chybové hlášky pochopitelné i bez technických znalostí systému.
- **NHA10 – Nápověda a dokumentace** – systém by měl poskytovat veškeré podklady pro plné pochopení aplikace i pro nezkušené uživatele. I když je dobré, když uživatelé dokáží systém používat rovnou bez jakékoliv předchozí přípravy, dokumentace by měla být vždy po ruce.

Jednotlivé body byly použity pro ohodnocení uživatelského rozhraní. Bodový rozsah byl 0–5 bodů, kde 5 je nejlepší hodnocení. Výsledek analýzy je vidět v tabulce 51. Jak je podle výsledku zřejmé, nebyly shledány zásadní rozpory oproti obecnému standardu.

5.3 Uživatelské testování

Pro programátora je prakticky nemožné simulovat chování aplikace v reálném provozu. Uživatelé jsou příliš různí a možností, jak službu používat příliš

	NHA1	NHA2	NHA3	NHA4	NHA5	NHA6	NHA7	NHA8	NHA9	NHA10
Výsledek	5	5	4	5	4	4	3.5	4.5	4.5	4

Tabulka 51: Nielsenova heuristická analýza – hodnocení

mnoho. I proto byli vybráni čtyři testeři, jejichž úkolem bylo zkoušet klientskou část a jakékoliv problémy popřípadě návrhy na vylepšení reportovat.

5.3.1 Klientská část

Jak již bylo zmíněno, úkolem vybraných uživatelů bylo aplikaci procházet a snažit se s ní reálně pracovat. Jako nejdůležitější body a otázky, na které by se měli primárně zaměřit, byly stanoveny následující:

- **Správa uživatelského účtu**
  - Probíhá registrace tak, jak je běžné u jiných služeb?
  - Je správa API klíčů dostatečně intuitivní tak, aby ji pochopil i začínající uživatel?
- **Správa prostředků**
  - Jak intuitivní je správa prostředků, kterými uživatel disponuje?
  - Je dostatečně zřejmé, které prostředky jsou volné a tím pádem alokovatelné pro nové strategie?
- **Založení strategie**
  - Je možnost založení strategie dostatečně jasná a celý proces sám o sobě návodný?
  - Je nabízená dokumentace dostatečně popisná pro naprogramování implementaci vlastní strategie?
- **Sledování vývoje strategie a portfolia**
  - Je zřejmé, jak se strategie během doby vyvíjela?
  - Jsou nabízené informace o strategie užitečné? Nehodily by se další metriky?

Jelikož klientská část není příliš složitá, nebylo objeveno ani mnoho chyb – většina se týkala chybějících prvků indikujících načítání dat. Takto nahlášené problémy byly obratem opraveny a v aplikaci by se tak již neměly vyskytovat.

U návrhů na vylepšení tomu bylo alespoň co se počtu týče jinak a sešla se jich celá řada. Na mnohé z nich dokonce upozornilo a více testerů. Následuje seznam těch, které byly shledány jako nejzajímavější.

- **UP1 – Vizualizace momentu, kdy proběhl obchod** – časová osa zobrazující vývoj strategie by mohla zobrazovat momenty, kdy proběhly jednotlivé obchody. Uživatel tak získá základní povědomí o tom, jaký vliv měly.
- **UP2 – Vizualizace vývoje měnových párů** – u obchodů by rovněž bylo dobré vidět vývoj měnového páru, který by naznačil úspěch dané objednávky.
- **UP3 – Zbytečně složitá správa burzovních API klíčů** – aplikace umožňuje spravovat API klíče skrz modální dialog obsahující rozevírací seznam. To působí zbytečně složitě a dalo by se zjednodušit.
- **UP4 – Lokalizace** – neměly by chybět další světové jazyky.
- **UP5 – Oddělení zastavených a běžících strategií** – v seznamu strategií není v současné době na první pohled zřejmé, jaké strategie jsou již zastavené a jaké jaké ještě běží. Pro uživatele by bylo přínosné od sebe tyto dvě skupiny lépe odlišit.
- **UP6 – Seznam prostředků, kterými strategii disponuje** – detail strategie by mohl obsahovat seznam prostředků, které jsou jí v dané době alokovány.

Všechny body byly důkladně zváženy a mnohé rovnou implementovány – UP1, UP3, UP5 a UP6, Zbylé dva požadavky (UP2 a UP4) sice dávají smysl, ale ne během této prvotní fáze – především kvůli časové náročnosti. S jejich nasazením se ale počítá v budoucnu v dalších kolech vývoje.

5.3.2 Testovací strategie

Vzhledem k tomu, že část aplikačního rozhraní není určená pro uživatele (respektive klientskou část programu), ale pro samotné strategie, se nabízí otázka, jak otestovat právě tyto funkcionality v běžném provozu. Z toho důvodu byla implementována testovací strategie, která měla za úkol sbírat dostupná data a simulovat rozhodnutí o nákupech/prodejích. Takto vytvořený program bude pracovat především s prvkem náhody. Není zde snaha jakkoliv zohledňovat reálná data o stavu trhu, jako spíš ověřit, že aplikace plní vzešlé rozkazy tak, jak by měla. Pro účely testování, které by opět mělo napodobovat různá chování, je strategii možné konfigurovat. To se týká především následujících parametrů:

- **Četnost obchodování** – hodnota větší než jedna, která v minutách označuje interval, na jehož konci by měl program provést obchod.
- **Četnost získávání dat** – hodnota větší než jedna, která v minutách označuje interval, na jehož konci by se program měl informovat o aktuálním stavu trhu.

- **Simulace chyb** – boolean hodnota udávající, jestli má program záměrně posílat chybné objednávky. Zaměřeno především na nedostatečný stav prostředků pro nákup/prodej, neexistující trhy atd.
- **Procentuální zastoupení chyb** – hodnota v rozmezí 0–100 udávající, kolik objednávek má být chybových. Bere se v potaz jen v případě, že parametr Simulace chyb je pravda.

Pro účely testování byl rovněž připraven skript, který se postará o vytvoření prostředků pro obchodování, založení několika strategií a jejich následné spuštění s předem definovanou sadou výše zmíněných parametrů. Testování je tak opět automatizované a odpadá nutnost strategie zakládat ručně tak, jak by to normálně musel dělat uživatel.

Stejně jako v případě unit testů, i v tomto případě strategie pomohly odhalit chyby, které tak mohly být napraveny před reálným spuštění služby.

Závěr

Cílem práce bylo vytvořit platformu umožňující snadnější provoz a testování strategií operujících na kryptoměnových trzích. Měla být rozhraním existujícím mezi burzami samotnými a strategiemi, které jsou zodpovědné za rozhodnutí o nákupech a prodejích. Z této pozice pak měla zaznamenávat a exekuovat přicházející obchody, poskytovat jednotné rozhraní pro jejich vykonání a získávání dat a v neposlední řadě také hlídat, že strategie obchodují jen s prostředky, které mají exkluzivně přiděleny.

Prvním krokem bylo vytvoření analýzy mapující požadavky, případy užití a také API rozhraní vybraných burz. Dalším krokem byl
návrh řešení – především architektury systému, databázového modelu a v neposlední řadě také uživatelského rozhraní. S těmito podklady v ruce došlo k samotné implementaci zvoleného řešení. Jako technologie pro vývoj backendových částí byl vybrán jazyk C# z frameworku .NET. Pro klientskou část aplikace pak bylo v vsazeno na moderní javascriptovou knihovnu React. Všem jednotlivým částem také byla dodána podpora pro jejich spuštění uvnitř Docker kontejneru. To usnadnilo především práce s nasazením a tím pádem i testování aplikace.

Z požadavků, které vyplynuly z analytické části práce byly naplněny
všechny. I tak je aplikace teprve v začátcích a do budoucna se počítá s celou řadou vylepšení, jejichž cílem je především posunout ji směrem ke konkurenceschopným a životaschopným službám spolehlivě sloužící svým uživatelům.

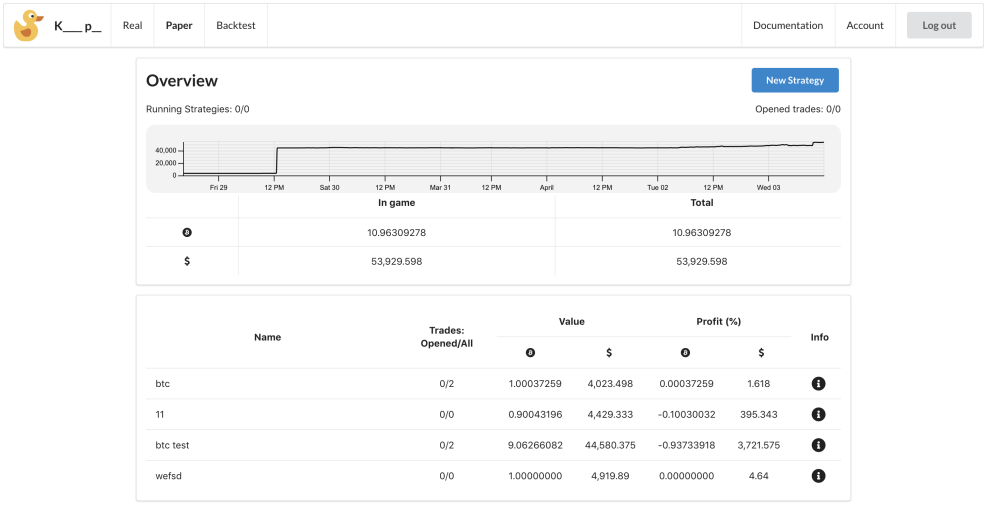
Body, které pro takto stanovený cíl bude nutné splnit jsou uvedené
v následující sekci.

Další rozvoj

- **Vytvoření zázemí pro správu uživatelů** – v případě ambicí na vytvoření plnohodnotné služby by bylo potřeba implementovat administrátorské rozhraní umožňující snazší správu uživatelů v systému.

- **Platební brána** – pro životaschopnost aplikace je nutné získat od uživatelů prostředky. Napojení aplikace na platební bránu je tím pádem dalším krokem do budoucna.
- **Rozšíření podporovaných burz** – zvýšení zájmu uživatelů by mohlo zapříčinit i podporování více burz. Vzhledem k návrhu, který s takovým rozšířením počítá, by navíc nešlo o implementačně náročný úkol.
- **Přidávání datových zdrojů a indikátorů** – zdrojů, které se pro rozhodnutí
  o nákupu/prodeji dají použít, je nepřeberné množství a do budoucna je v plánu jejich nabídku rozšiřovat podle zájmu uživatelů.
- **Lokalizace** – případné šance na úspěch by určitě zvýšila lokalizace. V ideálním případě by se to netýkalo jenom jazyka (spolu s různými tvary zápasů čísel, datumů atd.), ale i možnosti sledování ceny portfolia v různých měnách. Jelikož většina burz, pokud vůbec obchoduje s fiat měnami, nabízí pouze dolar, by bylo nutné zajistit službu poskytující ceny podporovaných párů k dolaru.
- **Zabezpečení** – jelikož se jedná o aplikaci, která manipuluje s reálnými peněžními prostředky, by před případným nutným provozem bylo zodpovědné nechat provést testy zabezpečení ideálně třetí stranou. Uživatelé jsou sice nabádáni, aby klíče, které aplikaci svěří, nepovolovaly případné výběry, ale historie ukázala, že útočníci jsou schopni zneužít i ty.
  Například zadáním objednávek na nákup měn s nízkým objemem obchodů za cenu výrazně převyšující tu aktuální – tržní, které byly obratem pokryty nabídkami zadanými útočníky.
- **Nové nástroje vizualizace** – V současné době aplikace nabízí jenom pohled na to, jaká byla v čase hodnota portfolia. Pro potřeby analýzy chování strategií by ovšem bylo dobré vidět, na jakých cenách měn, se kterými obchoduje, byly pozice otevřeny. Jelikož ale aplikace nijak neomezuje množství obchodovaných měn a povoluje i obchody na různých burzách, bude se jednat o poměrně komplikovaný nástroj, který je v tuto chvíli nad rámec této práce.

Ukázka aplikace



Obrázek 51: Snímek obrazovky aplikace – detail portfolia

Obrázek 52: Snímek obrazovky aplikace – detail strategie

Literatura

[1] Types of Algorithmic Strategies. [online], [cit 29-04-2019]. Dostupné z: https://www.babypips.com/news/forex-algorithmic-trading-20141112

[2] TradingView. [software], [cit 29-04-2019]. Dostupné z: https://www.tradingview.com

[3] Signals. [software], [cit 29-04-2019]. Dostupné z: https://signals.network

[4] Cryptohopper. [software], [cit 29-04-2019]. Dostupné z: https://www.cryptohopper.com

[5] van Rossum, M.: Gekko. [software], [cit 29-04-2019]. Dostupné z: https://gekko.wizb.it

[6] Binance. [software], [cit 29-04-2019]. Dostupné z: https://www.binance.com

[7] Bittrex. [software], [cit 29-04-2019]. Dostupné z: https://international.bittrex.com

[8] Kraken. [software], [cit 29-04-2019]. Dostupné z: https://www.kraken.com

[9] Poloniex. [software], [cit 29-04-2019]. Dostupné z: https://poloniex.com

[10] Newman, S.: *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015, ISBN 978-1491950357.

[11] Galitz, W.: *The essential guide to user interface design : an introduction to GUI design principles and techniques*. Indianapolis, IN: Wiley Pub, 2007, ISBN 0-470-05342-9.

[12] Žikovský, P.: *Návrh uživatelského rozhraní*. Praha: ČVUT, 2011.

[13] ASP.NET. [software], [cit 29-04-2019]. Dostupné z: https://dotnet.microsoft.com/learn/web/what-is-aspnet-core

[14] SQLite. [software], [cit 29-04-2019]. Dostupné z: https://www.sqlite.org

[15] Visual Studio. [software], [cit 29-04-2019]. Dostupné z: https://visualstudio.microsoft.com

[16] Visual Studio Code. [software], [cit 29-04-2019]. Dostupné z: https://code.visualstudio.com

[17] DB Browser for SQLite. [software], [cit 29-04-2019]. Dostupné z: https://sqlitebrowser.org/

[18] Insomnia REST client. [software], [cit 29-04-2019]. Dostupné z: https://insomnia.rest

[19] GitHub. [software], [cit 29-04-2019]. Dostupné z: https://github.com

[20] Hunt, A.; Thomas, D.: *Programátor pragmatik: jak se stát lepším programátorem a vytvářet kvalitní software*. Brno: Computer Press, první vydání, 2007, ISBN 978-80-251-1660-9.

[21] Fowler, D.: Hosted Service. [online], [cit 29-04-2019]. Dostupné z: https://gist.github.com/davidfowl/a7dd5064d9dcf35b6eae1a7953d615e3

[22] Cache in-memory in ASP.NET Core. [online], [cit 29-04-2019]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-2.2

[23] React. [software], [cit 29-04-2019]. Dostupné z: https://reactjs.org

[24] Axios. [software], [cit 29-04-2019]. Dostupné z: https://github.com/axios/axios

[25] Reselect. [software], [cit 29-04-2019]. Dostupné z: https://github.com/reduxjs/reselect

[26] Semantic UI React. [software], [cit 29-04-2019]. Dostupné z: https://react.semantic-ui.com

[27] VX visualization components. [software], [cit 29-04-2019]. Dostupné z: https://github.com/hshoff/vx

[28] What is a Container. [online], [cit 29-04-2019]. Dostupné z: https://www.docker.com/resources/what-container

[29] Wilder, J.: Nginx proxy container. [software], [cit 29-04-2019]. Dostupné z: https://github.com/jwilder/nginx-proxy

[30] Blusseau, Y.: LetsEncrypt companion container for nginx-proxy. [software], [cit 29-04-2019]. Dostupné z: https://github.com/JrCs/docker-letsencrypt-nginx-proxy-companion

[31] Seemann, M.: *Dependency injection in .NET*. Shelter Island, NY: Manning, 2012, ISBN 1-935182-50-1.

[32] Moq. [software], [cit 22-04-2019]. Dostupné z: https://github.com/Moq/moq4/wiki/Quickstart

[33] Nielsen, J.: 10 Usability Heuristics for User Interface Design. [online], [cit 29-04-2019]. Dostupné z: https://www.nngroup.com/articles/ten-usability-heuristics/

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

XML Extensible markup language

JSON JavaScript Object Notation

API Aplication Programming Interface

YAML Yet Another Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

TLS Transport Layer Security

DOM Document Object Model

SPA Single Page Application

CSS Cascading Style Sheets

ORM Object-relational mapping

LINQ Language Integrated Query

IP Internet Protocol

DRY Don't repeat yourself

CRUD Create, Read, Update, Delete

DI Dependency injection

UML Unified Modeling Language

IDE Integrated Development Environment

HFT High-frequency Trading

ICO Initial Coin Offering

REST Representational State Transfer

JWT JSON Web Token

URL Uniform Resource Locator

Příloha B

Dockerfile

```
version: '2'

services:
 masterdatamanager:
  image: masterdatamanager
  expose:
   - "80"
  volumes:
   - "./database/Database.db:/app/Database/Database.db"
  build:
   context: ./masterDataManager
   dockerfile: Dockerfile
  environment:
   ASPNETCORE_ENVIRONMENT: Development
   VIRTUAL_HOST: api.DOMENA.cz,www.api.DOMENA.cz
   LETSENCRYPT_HOST: api.DOMENA.cz,www.api.DOMENA.cz
   LETSENCRYPT_EMAIL: h.kirchner@seznam.cz

 marketdataprovider:
  image: marketdataprovider
  environment:
   - ASPNETCORE_ENVIRONMENT=Development
  expose:
   - "80"
  volumes:
   - "./database/MarketsDb.db:/app/Database/MarketsDb.db"
  build:
   context: ./marketDataProvider
   dockerfile: Dockerfile
  environment:
   VIRTUAL_HOST: market.DOMENA.cz,www.market.DOMENA.cz
   LETSENCRYPT_HOST: market.DOMENA.cz,www.market.DOMENA.cz
   LETSENCRYPT_EMAIL: h.kirchner@seznam.cz
```

```
web:
 image: kryplweb
 build:
  context: ./newclient
  dockerfile: Dockerfile
 expose:
  - "80"
 environment:
  VIRTUAL_HOST: DOMENA.cz,www.DOMENA.cz
  LETSENCRYPT_HOST: DOMENA.cz,www.DOMENA.cz
  LETSENCRYPT_EMAIL: h.kirchner@seznam.cz

nginx-proxy:
 image: jwilder/nginx-proxy
 ports:
  - "80:80"
  - "443:443"
 volumes:
  - "/etc/nginx/vhost.d"
  - "/usr/share/nginx/html"
  - "/var/run/docker.sock:/tmp/docker.sock:ro"
  - "/etc/nginx/certs"

letsencrypt-nginx-proxy-companion:
 image: jrcs/letsencrypt-nginx-proxy-companion
 volumes:
  - "/var/run/docker.sock:/var/run/docker.sock:ro"
 volumes_from:
  - "nginx-proxy"
```

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `client` ..... zdrojové kódy klientské části
    - `database` ..... databázové soubory
    - `marketDataManager` ..... zdrojové kódy market data manager komponenty
    - `masterDataProvider` ..... zdrojové kódy master data provider komponenty
    - `strategies` ..... zdrojové kódy testovací strategie
    - `docker-compose.yml` ..... Docker file
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF