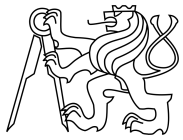
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Platforma pro automatizované obchodování s kryptoměnami
Student:	Bc. Petr Hejna
Vedoucí:	Ing. Jan Václavík
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem práce je vytvořit nástroj pro automatizované obchodování s kryptoměnami (Bitcoin, Ethereum a další), a to pomocí jejich vzájemného směňování. Do tohoto nástroje bude možné integrovat různé obchodní strategie. Jedna z těchto strategií bude implementována přímo v této práci.
- Využijte již existující řešení (případně i více) pro provádění transakcí mezi různými kryptoměnami. Například shapeshift.io nebo obdobnou službu.
- Analyzujte používané strategie pro obchodování a navrhněte vhodný reprezentativní model, a ten ověřte v praxi.
- Strategie obchodních modelů budou realizovány formou modulu přímo do nástroje. Mohou ale fungovat také tak, že budou volat API jiné služby/systému.
- Aplikace bude vizualizovat výslednou bilanci a průběh transakcí ve webovém rozhraní.
- Analýza a volba vhodných technologií je součástí práce.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 8. února 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Platforma pro automatizované obchodování s kryptoměnami

Bc. Petr Hejna

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jan Václavík

4. května 2018

Poděkování

Děkuji vedoucímu práce Ing. Janu Václavíkovi za rady a zpětnou vazbu během vývoje, Ing. Mirovi Hrončokovi za pomoc s jazykem Python a Bc. Jindřichu Samcovi za konzultace návrhu frontendové aplikace v Reactu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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 4. května 2018 .....

Odkaz na tuto práci

Hejna, Petr. *Platforma pro automatizované obchodování s kryptoměnami.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018. Dostupný také z WWW: ⟨`https://github.com/Achse/coinrat_thesis/blob/master/DP_Hejna_Petr_2018.pdf`⟩.

Abstrakt

Práce se zabývá návrhem a implementací modulovatelné platformy pro obchodování na burze se zaměřením na kryptoměny. Plaftorma poskytuje rozhraní pro vytvoření obchodních strategií, umožňuje jejich spouštění proti burze a poskytuje nástroje pro testování a měření výsledků na historických datech.

Klíčová slova automatické obchodování na burze, kryptoměny, strategie obchodování na burze

Abstract

The thesis deals with the design and implementation of a modular trading-platform with focus on cryptocurrencies. The platform provides an interface for creating trading strategies. The platform also provides tools for testing and evaluation of strategies based on historical data.

Keywords stockmarket autotrading, cryptocurrencies, stockmarket trading-strategies

Obsah

Seznam obrázků

Úvod

Kryptoměny jsou dnes velkým tématem. Jsou svobodné a neregulované. Mnoho lidí z laické veřejnosti i profesionálních obchodníků v nich vidí příležitost k zisku a pouští se do obchodování s těmito (vysoce volatilními a rizikovými) komoditami.

Vytváření automatických botů[1] je přirozeným dalším krokem. Existují řešení, která nabízejí takovéto boty jako SaaS[2] a také několik volně použitelných projektů s otevřenými zdrojovými kódy. Existují také řešení, kde je možné si pronajmout již vytvořenou strategii a za poplatek ji nechat běžet. Tato řešení jsou podrobněji popsána v kapitole 2.1 Rešerše stávajících řešení).

Hlavní motivací pro vznik této práce bylo zjištění absence řešení s otevřenými zdrojovými kódy, které by poskytovalo:

- absolutní rozšiřitelnost (nejen strategií, ale i burz a dalších částí aplikace),
- přehlednou vizualizaci,
- nevyžadovalo by důvěru v službu třetí strany.

Kromě výše zmíněného byla zjištěna absence řešení, které by mělo dobře navržený doménový model, tak aby jej bylo možno snadno rozšiřovat a například bylo v budoucnu schopno spouštět strategie dělající meziburzovní arbitráže (tato problematika je více diskutována v kapitole 2.4.1 Arbitráže).

Cílovou skupinou pro tento produkt jsou lidé znalí programování, neboť strategie je nutno programovat jako moduly. Nicméně modulární systém by umožňoval vytvoření interpreta uživatelsky přívětivého zápisu pro lidi bez zkušeností s programováním.

[1] Robot nebo jen bot je automatický proces, který samostatně vykonává nějakou činnost. Převážně v prostředí internetu.

[2] Software as a Service je označení pro model nasazení softwaru, ve kterém aplikace běží mimo prostředí uživatele a je mu poskytována jako služba přes internet.

Vize této platformy je stát se úspěšným soupeřem na trhu platforem obchodních botů.

První kapitola práce popisuje vytyčené cíle, druhá se věnuje analýze. Druhá kapitola začíná uvedením problematiky burzovních obchodů a poté je analyzováno technické řešení. Ve třetí kapitole je podrobně popsána samotná realizace platformy. Tato kapitola je rozdělena na část věnující se frontendu a část věnující se backendu. V poslední kapitole je zpracována implementace modulů pro ukládání dat, vytváření strategií a napojení na burzu.

Kapitola 1

Cíl práce

Cílem práce je vytvoření modulární platformy, která umožní psát obchodní strategie nezávisle na burze a zdroji burzovních dat. Platforma bude řešit abstrakci tak, aby všechny klíčové součásti byly vyměnitelné a rozšiřitelné. Strategie, konektor na burzu, zdroj burzovních dat a datové uložiště budou realizovány formou vyměnitelných modulů.

Modularitou si práce klade za cíl umožnit škálování tvorby a validace obchodních strategií. Cílem práce je tedy vyvinout řešení, které:

- Umožňuje snadno implementovat různé obchodní strategie bez nutnosti vázat se na konkrétní zdroj dat nebo burzu.
- Vizualizuje průběh strategie v reálném čase.
- Poskytuje nástroje pro zhodnocení výkonnosti (profitability) obchodní strategie.
- Umožňuje testování obchodní strategie na testovacích datech. Výsledky těchto testů umí vhodně měřit a vizualizovat.

Dále si práce klade za cíl implementovat ukázkové strategie a na nich implementované řešení ověřit v praxi.

Kapitola 2

Analýza a návrh

2.1 Rešerše stávajících řešení

Na začátku této kapitoly je sepsán přehled existujících platforem pro vytváření obchodních botů. Vzhledem k tomu, že obchodování na burze je hra s nulovým součtem[3], je zde poměrně malá motivace tyto nástroje na tvorbu botů sdílet.

2.1.0.1 Gekko

Jedná se o program[4] s otevřeným zdrojovým kódem, napsaný v nodejs. Je dobře zdokumentovaný s nízkou vstupní bariérou. Obsahuje jak podporu pro spouštění strategií v simulaci, tak vizualizaci výsledku ve webovém prostředí. Pracuje na jednoduchém principu, který funguje tak, že vždy když přijde nová svíčka, strategie může zareagovat signálem *long* nebo *short*[5]

Bohužel tímto jsou možnosti strategií velmi limitované. Nemají k dispozici ostatní obchody ani přístup ke stavu portfolia. Z toho pak plyne velmi zásadní limitace a to taková, že obchody probíhají pouze v režimu *all-in/all-out* (vždy přesouvá celý kapitál mezi komoditami v páru).

2.1.1 Zenbot

Tato platforma s otevřenými zdrojovými kódy, je napsaná v nodejs a obsahuje velké množství předem připravených strategií, které je možné psát jako rozšíření. Disponuje vizualizací, která je ale limitovaná svou statičností. V podstatě se jedná o vizualizaci snapshotu stavu, který je vždy nutno ručně přegenerovat.

[3] Jedná se o systém, ve kterém jedna strana může něco získat jen za předpokladu, že jiná strana něco ztratí.

[4] `https://github.com/askmike/gekko`

[5] *Long* je termín označující situaci, kdy investor nakoupí v očekávání, že prodá se ziskem. Oproti tomu *short* označuje situaci, kdy investor prodá komoditu s tím, že ji později nakoupí levněji.

Tato platforma obsahuje možnost odesílání notifikací do mnoha kanálů (například Slack, IFTT, Pushbullet, ...). Obsahuje také rozhraní v příkazové řádce pro přímé zadávání objednávek.

Platforma dále obsahuje mechanismus pro obchodování pomocí limitních objednávek s cílem minimalizovat cenu za poplatky[6]. Je však pravděpodobné, že tento mechanismus v budoucnu opustí[7] kvůli jeho nespolehlivosti, byť za cenu vyšších poplatků.

I přes nedostatky ve vizualizaci se jedná o současné nejlepší volně použitelné (licence MIT) řešení. Obsahuje mnoho pokročilých funkcí, které mohou být inspirací pro další rozvoj aplikace realizované v této práci.

2.1.2 Bowhead

Tento boilerplate[8]/framework je napsaný v PHP[9]. Výsledný bot se ovládá čistě přes konzoli. Architektura celého projektu není příliš čitelná a pravděpodobně se hodí spíše k prototypování.

V době realizace této práce však byla oznámena reimplementace mnoha částí projektu Bowhead. Byla slíbena podpora pro arbitráže a podpora pro snadné vytváření vlastních strategií (tzv. *strategy builder*), včetně simulací, sdílení strategií a rozšíření dokumentace.

2.1.3 Komerční uzavřená řešení

Jelikož se tato práce zabývá implementací otevřeného řešení, které si bude moci každý provozovat sám, jsou placené a uzavřené platformy pouze stručně shrnuty.

- **Haasbot** – je populární ale poměrně nákladné řešení, provozované uživatelem. Obsahuje simulace (tzv. *backtesting*) a umí arbitráže.
- **Gunbot** – stejně jako Haasbot je toto velmi populární placený bot, který je provozován uživatelem. Dále Gunbot nabízí tržiště existujících strategií. Spouštění simulací Gunbot neobsahuje.
- **Cryptotrader** – jedná se o SaaS platformu. Jednotlivé strategie se píší jako scripty v JavaScriptu. Platforma obsahuje možnost simulace strategií a vizualizací. Navíc také poskytuje tržiště strategií. Je tedy možné si pronajmout strategii někoho jiného a tu provozovat se svou konfigurací.

[6] Objednávky s fixní (limitní) cenou jsou typicky burzami přijímány s nižším poplatkem než objednávky za aktuální cenu. Princip fungování těchto poplatků je podrobněji popsán v kapitole 4.5.

[7] `https://github.com/DeviaVir/zenbot#current-status`

[8] Termín `Boilerplate` označuje zpravidla kód, který se opakovaně používá. Často ale bez jakékoliv abstrakce nebo definovaného rozhraní.

[9] `https://github.com/joeldg/bowhead`

2.2 Požadavky

Před vlastní implementací byly identifikovány následující požadavky na systém.

2.2.1 Funkční požadavky

- Podle strategie lze obchodovat proti reálné burze.
- Strategii je možno ověřit na historických nebo syntetických datech.
- Strategie a konektory na jednotlivé burzy lze snadno integrovat do systému. Autor strategie nebo konektoru nemusí znát technické detaily celého systému a pracuje jen s rozhraním pro daný modul.
- Výsledky běhu strategií a jejich simulací je možno vizualizovat do grafů a tabulek.
- Jsou dostupné základní metriky pro vyhodnocení běhu strategie nebo její simulace.

2.2.2 Nefunkční požadavky

- Systém bude moci provozovat každý uživatel sám na vlastních serverech (tzv. *on-premises*).
- Prostředí pro vizualizaci bude dostupné přes webový prohlížeč.
- Aplikace bude snadno auditovatelná pro zajištění důvěryhodnosti.

2.3 Výběr technologií

Výběr použitých technologií se odvíjel od cílů stanovených v předchozí kapitole a analýzy požadavků.

Pro oddělení aplikační logiky od uživatelského rozhraní a vizualizací byla celá aplikace rozdělena na backendovou a frontendovou část. Důvodem tohoto rozdělení bylo oddělení zodpovědností a z toho plynoucí redukce komplexity jednotlivých komponent.

Backend spouští strategie, poskytuje připojení na burzy, zajišťuje stahování dat a jejich persistenci. Frontend poskytuje uživatelské rozhraní pro spouštění simulací a vizualizuje burzovní data a výsledky běhů strategií.

2.3.1 Komunikační protokol

Další navrhovanou částí práce je rozhraní a komunikační protokol mezi backendem a frontendem.

Protože cílem práce je implementovat platformu, která umožňuje vizualizovat burzovní data a operace obchodní strategie v reálném čase, bylo potřeba najít technologii, která takto rychlou komunikaci umožňuje. V úvahu připadaly dvě varianty: polling a socket.

2.3.1.1 Polling

Polling je technika, kdy se klient opakovaně dotazuje serveru, zda se stav změnil. Je však zřejmé, že takovéto opakované dotazování klade velké nároky na síť a také prodlevou mezi jednotlivými dotazy způsobuje zpoždění. Dochází tedy k situaci, ve které je síťová náročnost vyměňována za nižší latenci a naopak.

Tento problém lze řešit takzvaným HTTP Long Pollingem, který používá například Facebook [1]. Při Long Pollingu dojde k otevření HTTP spojení, které server drží otevřené až do chvíle, kdy nastane změna stavu. Informaci o změně stavu server následně odešle skrze toto spojení klientovi. Tím se efektivně vyřeší jak problém náročnosti na síť (velmi výrazně se omezí počet spojení), tak i problém zpoždění přenosu informace.

Server však stále musí obsluhovat výrazný počet spojení, i když menší než u klasického pollingu. Dále Long Polling není doopravdy realtime, jelikož během zavírání spojení a vytváření nového se nachází slepé místo, kdy komunikace nemůže probíhat.

2.3.1.2 Socket

Při komunikaci přes socket je spojení drženo neustále. Nedochází tedy ke zbytečným dotazům na server. Komunikace může probíhat oboustranně, zprávy jsou odesílány jak ze serveru, tak z klienta okamžitě.

Tím, že server posílá aktivně data klientovi bez nutnosti čekat na klientův dotaz, je komunikace velmi rychlá. Zároveň je ale náročnější na implementaci. Například je složitější ošetřit stavy, kdy dojde k rozpadnutí spojení kvůli chybám na síti. Zatímco v prvním případě selže HTTP request a klient dotaz zopakuje, v druhém případě, kdy server aktivně posílá data klientovi, musí klient detekovat ztrátu spojení a následně po znovunavázání spojení synchronizovat celý stav. Navíc je třeba implementovat mechanismus, který klientovi umožňuje přihlásit se ke konkrétní podmnožině změn, aby server klienta nezahlcoval informacemi o změně stavu, které klient nepotřebuje.

2.3.1.3 Zvolená technologie

Pro implementaci platformy byl zvolen přístup komunikace přes socket s využitím knihovny socket.io[10], která je na webu současným průmyslovým standardem. Jedná se o poměrně vysokoúrovňovou knihovnu, která díky svému jednoduchému používání kompenzuje výše diskutované nevýhody.

[10] `https://socket.io`

Výhodou této knihovny je, že komunikační protokol je nezávislý na transportním mechanismu. Podporuje tedy jak komunikaci přes socketové spojení, tak i komunikaci přes polling[2] jako zálohu.

2.3.2 Backend

Technologické nároky, které jsou kladené na backendovou část, jsou ovlivněné zejména implementačními nároky obchodních strategií. Nároky na implementaci strategií jsou tyto:

- rozšířenost technologie,
- podpora pro vědecké výpočty,
- výkonnost (strategie mohou mít velké nároky na výpočetní výkon),
- dobrá podpora pro psaní modulů,
- schopnost komunikovat s frontendem v reálném čase (podpora pro socket.io).

Na základě těchto nároků byl zvolen jazyk Python (v major verzi 3). Jedná se o vyspělou technologii, která je na trhu již více než 10 let. Python je velmi populární ve vědecké komunitě a má mnoho knihoven pro vědecké výpočty (například knihovny NumPy nebo SciPy). Python umožňuje optimalizovat výpočetně náročné operace vytvořením Python modulu přímo v jazyce C, nebo použitím knihovny Cython[11]. Implementace socket.io je pro jazyk Python dostupná díky knihovně.

2.3.3 Frontend

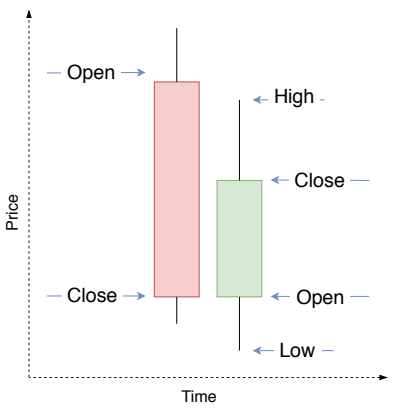
Nároky na frontendovou část jsou ovlivněné zejména požadavkem na vizualizaci dat ve webovém rozhraní a nutností snadné implementace. Celý výčet nároků tedy je:

- schopnost kvalitně vizualizovat burzovní data ve webovém rozhraní,
- komunikace s backendem v reálném čase (podpora pro socket.io),
- rozšířenost technologie,
- snadná implementace.

[11] `http://cython.org`

Na základě těchto nároků byl zvolen jazyk JavaScript s použitím UI frameworku ReactJS[12]. Pro vizualizaci burzovních dat existuje knihovna React Stockcharts[13]. Implementace socket.io pro komunikaci v reálném čase je podporována formou oficiální knihovny. ReactJS je moderní, dobře etablovaná technologie, která je zaštítěná společností Facebook. Kolem ReactJS existuje velká komunita a množství knihoven tvořící rozsáhlý ekosystém.

Jelikož JavaScript neobsahuje syntaxi pro statické typování, byla doplněna knihovna Flow[14].



Obrázek 2.1: Burzovní OHLC svíčka

2.4 Analýza domény

Přestože se tato práce zabývá převážně obchodováním s kryptoměnami, byla analýza domény provedena tak, aby pokrývala obecně obchodování s jakýmikoliv statky na jakémkoliv tržišti. Kryptoměny jsou pak jen speciálním případem užití.

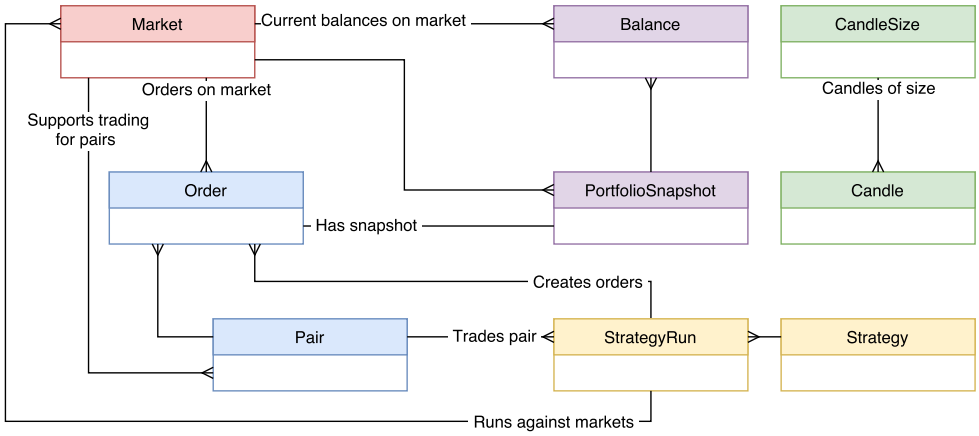
Klíčové doménové entity byly identifikovány takto (v závorce je zvýrazněným textem uveden název pro použití v implementaci):

- Burza (`Market`) – tržiště, na kterém je možné směnit statek za jiný statek. Příkladem takových marketů pro kryptoměny jsou: Bittrex, Bitfinex, Shapeshift. Příkladem z klasického obchodování je pak velmi známý Forex.
- Portfolio (`PortfolioSnapshot`) – souhrn všech statků, které obchodník v danou chvíli drží.
- Zůstatek (`Balance`) – množství statků vlastněné obchodníkem na dané burze.
- Obchodovaný pár (`Pair`) – dvojice komodit (v kontextu této práce klasických měn a kryptoměn), které lze na dané burze směnit. Příkladem je třeba dvojice dolar–bitcoin označována jako `USD-BTC`.
- Objednávka (`Order`) – směna daného množství dvou statků za danou cenu. Objednávka může obsahovat podmínku, za jaké se provede. Dokud není uzavřena, může být zrušena.

[12] `https://reactjs.org`
[13] `http://rrag.github.io/react-stockcharts`
[14] `https://flow.org`

- Svíčka burzovního grafu (`Candle`) – jedná se o agregovanou informaci o ceně dvou komodit v páru za určité časové období. Svíčka obsahuje čtyři informace (viz. obrázek 2.1):
  - *Open* – cena na začátku časového intervalu.
  - *High* – maximální cena v časovém intervalu.
  - *Low* – minimální cena v časovém intervalu.
  - *Close* – cena na konci časového intervalu.
- Velikost svíčky (`CandleSize`) – velikost intervalu svíčky. Častým nárokem je bezezbytková dělitelnost nadřazené časové jednotky (například šest čtyřhodinových svíček se přesně vejde do jedné denní svíčky). Typické velikosti intervalu jsou: 1 minuta, 5 minut, 15 minut, 1 hodina, 4 hodiny, 6 hodin, 12 hodin, 1 den, 1 týden.
- Obchodní strategie (`Strategy`) – algoritmus podle kterého se obchodník rozhoduje jak nakupovat a prodávat statky.
- Exekuce obchodní strategie (`StrategyRun`) – běh strategie pro nějaký pár a jednu nebo více burz.

Diagram entit a jejich vazeb je znázorněn na následujícím obrázku 2.2.



Obrázek 2.2: Diagram doménových entit a jejich vazeb

2.4.1 Arbitráže

Při arbitráži dochází k nakupování statků na jednom tržišti a následně k okamžitému prodeji na jiném tržišti za vyšší cenu za účelem zisku na rozdílných cenách mezi tržišti. Dochází tak k přirozenému srovnávání ceny mezi burzami.

Návrh byl vytvořen tak, aby strategie mohla arbitráž provádět. Jedna strategie tedy může obchodovat na více tržištích naráz.

2.5 Analýza strategií a obecný návrh jejich rozhraní

Pro vhodný návrh celé architektury bylo nutné zmapovat, jakým způsobem se na burzách obchoduje a jaké jsou možné typy strategií. Z těchto poznatků pak vychází návrh rozhraní pro strategie.

Vstupem všech strategií jsou jednoduše data. Nejčastěji se jedná o historický vývoj ceny a objemu obchodované komodity. Avšak může se jednat o libovolná data, která autor strategie považuje za relevantní. Z těchto dat se následně derivují indikátory (někdy též signály). Na základě nich se pak strategie rozhodne, zda a jakou zadat objednávku. Vstupem pro strategii je i aktuální stav objednávek na burzách, na kterých strategie obchoduje, a disponibilní zůstatky jednotlivých komodit.

Jelikož data o historickém vývoji ceny jsou zdaleka nejpoužívanější, zabývá se práce pouze jimi. V případě, že by některá strategie chtěla využívat dalších dat, je nutné, aby si jejich získání, zpracování a persistenci zajistila sama.

Rozhraní pro čtení dat přímo z účtu na burze, který má strategie k dispozici, bylo navrženo přímo do platformy, jelikož se jedná o rozhraní snadno abstrahovatelné napříč burzami a je pro všechny strategie totožné.

Oproti tomu, různých indikátorů je velké množství. V návrhu tedy nebyly zahrnuty jako součást platformy. Jejich zajištění je v režii každé strategie. To lze realizovat například použitím externí knihovny.

Výstupem každé strategie je série příkazů na burzu, kterými se vytvářejí a případně ruší objednávky. Jelikož se jedná o velmi snadno standardizovatelné rozhraní, bylo zahrnuto do návrhu platformy.

Strategiím tedy bude na jedné straně poskytováno rozhraní pro čtení burzovních dat (o vývoji ceny) a rozhraní pro komunikaci s burzou, na druhé straně budou mít strategie k dispozici rozhraní pro vytváření a rušení objednávek.

2.6 Návrh architektury

2.6.1 Modularita

Pro modularitu platformy bylo navrženo izolovat klíčové části. Tyto izolované části jsou nazvány zásuvné moduly (v implementaci *plugins*).

- `SynchronizerPlugin` – obsahuje jeden nebo více `Synchronizer`, který může být spuštěn platformou pro stažení a kontinuální synchronizaci burzov-

ních dat pro daný pár a burzu. Jeden zásuvný modul může podporovat více burz.

- `MarketPlugin` – obsahuje jeden nebo více `Market`. Market reprezentuje vždy jednu burzu a platforma skrze tento objekt s burzou komunikuje (skrze něj spravuje objednávky, zjišťuje zůstatky, ...).
- `StrategyPlugin` – obsahuje jednu nebo více `Strategy`. Platforma strategii poskytuje vysokoúrovňové rozhraní pro práci s burzovními daty a pro komunikaci se samotnou burzou.
- `StoragePlugins` – tyto zásuvné moduly obsahují třídy pro persistenci dat (`Candle`, `Order`, `PortforlioSnapshot`). Pokud se v praxi ukáže, že daná implementace nebo zvolená technologie nevyhovuje, je možné ji snadno vyměnit za jinou.

2.6.2 Paralelizace

V aplikaci bylo identifikováno několik procesů, které je třeba vykonávat paralelně:

- Běžící obchodní strategie, která v reálném čase odesílá signály k obchodování. Strategií může běžet více najednou.
- Synchronizační proces, který stahuje burzovní data. Těchto procesů může běžet také více najednou (pro různé páry a burzy).
- Socket.io server, který poskytuje rozhraní pro komunikaci s frontendem.

Nejpřímočařejším řešením paralelizace je použití vláken, které umožňuje synchronizaci přes sdílenou paměť. Bohužel, v Pythonu je synchronizace mezi vlákny řešena technikou *Global interpreter lock*[3], která brání využívání více jader procesoru naráz. Kvůli této limitaci bylo vícevláknové řešení zavrženo. Návrh tedy počítá s paralelizací na úrovni systémových procesů.

Jelikož jednotlivé procesy jsou dosti nezávislé, jejich synchronizace není implementačně příliš náročná. Proces stahující burzovní data pouze zapisuje data do úložiště. Strategie burzovní data pouze čte a zapisuje jen data, která jsou vlastní každému běhu strategie a tedy nikdy nekolidují. Frontend (resp. socket.io server) data pouze čte a vizualizuje.

2.6.2.1 Synchronizace stavu s frontendem v reálném čase

Jediným místem, které je z návrhu implementačně náročnější, je propagace informace o nových datech (burzovní data, objednávky a podobně) do frontendu, který je má v reálném čase vizualizovat. Jednotlivé procesy tedy musí o změně stavu dát vědět socket.io serveru, který je odešle do frontendu.

Pro komunikaci mezi procesy byla zvolena technologie RabbitMQ, která umožňuje posílání zpráv mezi procesy pomocí AMQP. Důvodem pro zvolení této technologie je především fakt, že na aplikaci není kladena zodpovědnost za orchestraci procesů. Oproti tomu nativní řešení (například knihovnou `multiprocessing`), vyžaduje od aplikace aby procesy orchestrovala sama. Dalším důvodem pro použití této technologie je její rozšířenost a dobrá podpora v jazyce Python.

V této práci tedy návrh počítá s tím, že procesy si orchestruje uživatel sám, například jejich spouštěním jako `systemd`[15] služby. Rozšířením tohoto návrhu do budoucna je pak přidání samostatné vrstvy, která by orchestrovala spouštění jednotlivých procesů.

2.6.3 Komponentový model

Na základě předchozí analýzy domény a procesů v aplikaci byly navrženy komponenty a jejich závislosti. Znázorněno na obrázku 2.3):

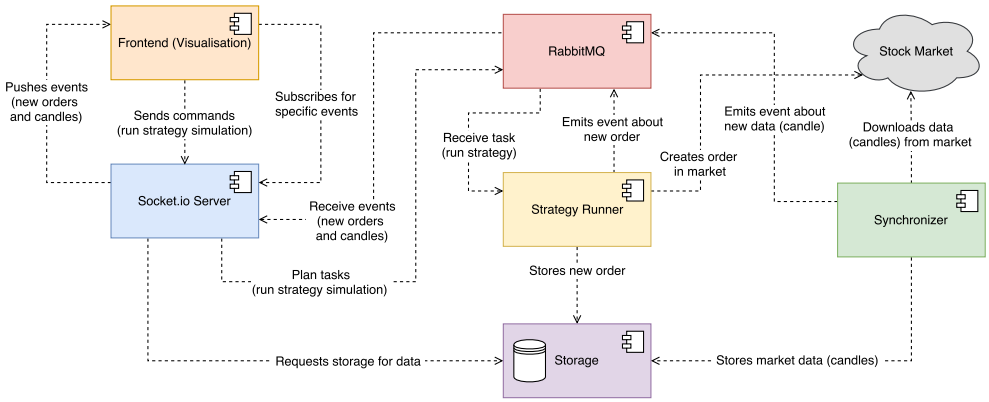
Komponenty:

- `Frontend` – poskytuje uživatelské rozhraní, vizualizuje burzovní data a běhy strategií. Poskytuje také rozhraní pro spuštění simulace strategie.
- `Socket.io server` – poskytuje rozhraní pro frontend.
- `StrategyRunner` – spouští strategie a poskytuje jim API pro čtení burzovních dat a obchodování.
- `Synchronizer` – komponenta pro stahování dat z burz.
- Datové úložiště pro objednávky a burzovní data.
- Prostředník komunikace mezi procesy (RabbitMQ).

Závislosti:

- Frontend závisí na socket.io serveru, skrze který načítá data a přihlašuje se u něj k odběru nových dat (nově příchozích svíček z burz a nových objednávek z běžících strategií). Dále skrze socket.io deleguje spuštění simulace.
- Socket.io server závisí přímo na datovém úložišti, ze kterého načítá data pro frontend. Dále obsahuje závislost na frontend. Vidí frontend jako klienta, se kterým má otevřené spojení a na základě předchozího přihlášení k odběru mu aktivně zasílá nové svíčky a objednávky. Tato cyklická

[15] `https://www.freedesktop.org/wiki/Software/systemd`



Obrázek 2.3: Diagram komponent

závislost je nevyhnutelná z důvodu oboustranné komunikace mezi frontendem a socket.io serverem. Další závislostí socket.io serveru je RabbitMQ. Z něj do socket.io serveru přicházejí informace o nově příchozí svíčce nebo objednávce. Tato zpráva je nazvána `Event`. Požadavek od frontendu na spuštění simulace socket.io server naplánuje do RabbitMQ. Takto naplánovaný požadavek je nazván `Task`.

- `StrategyRunner` závisí přímo na datovém úložišti, ze kterého zprostředkovává data běžící strategii. Dále závisí na službě RabbitMQ, ze které dostává pokyn ke spuštění simulace. Také do ní emituje `Event` při vytvoření objednávky strategií.

- `Synchronizer` závisí pouze na datovém úložišti, kam ukládá stáhnutá data, a na službě RabbitMQ, do kterého emituje `Eventy` o nových svíčkách.

2.6.4 Zabezpečení aplikace

Aplikace je určena pro provoz *on-premises*, tedy na strojích, které má uživatel pod úplnou kontrolou. Návrh tedy nepočítá s více uživateli, kteří interagují s aplikací. Neřeší tedy řízení přístupu ani úrovně oprávnění. Zároveň přenáší některé zodpovědnosti za zabezpečení na uživatele, resp. provozovatele aplikace.

V návrhu byly identifikovány možné vektory útoku na aplikaci a k nim bylo navrhnuto příslušné opatření.

2.6.4.1 Podvodný zásuvný modul

Je možné, že útočník vydá zásuvný modul, jehož součástí bude škodlivý kód. Tento modul se pak bude vydávat za seriozní modul nebo se pokusí imitovat

již existující modul.

U všech zásuvných modulů, které jsou provozovány uživatelem v platformě, je třeba vždy auditovat zdrojový kód. Implementace v jazyce Python zde přináší výhodu. Tím, že se jedná o interpretovaný jazyk, je kód snadno dostupný pro audit. Je však důležité si dát pozor, pokud zásuvný modul používá předkompilovaný kód. Zvláštní pozornost je pak třeba věnovat auditu knihoven a modulů, které daný zásuvný modul využívá jako závislosti.

2.6.4.2 Komunikace s burzou

Podvrhnutí rozhraní pro komunikaci s burzou nebo odposlechnutí komunikace za účelem získání přístupu k burze je další možný vektor útoku.

Zodpovědnost za obranu proti tomuto útoku leží na autoru zásuvného modulu pro připojení k burze. Je nezbytně nutné vynutit s burzou komunikaci po šifrovaném protokolu – v případě HTTP tedy vyžadovat HTTPS. Zároveň je třeba vždy ověřit identitu protistrany zda opravdu dochází ke komunikaci s žádanou burzou, a tedy nikdy nevypínat ověřování certifikátu, který bývá ve většině knihoven (v Python knihovna `requests`) ve výchozím stavu zapnutý.

2.6.4.3 Rozhraní socket.io serveru

V návrhu je počítáno s tím, že socket.io server by v produkčním prostředí vůbec neběžel, aby se tento vektor útoku zcela eliminoval. Případně by komunikace s ním probíhala výhradně přes SSH tunel. Tím je zodpovědnost za zabezpečení komunikace přenesena na provozovatele.

Cílem této práce není tuto službu provozovat jako SaaS, a není tedy nutné poskytovat webové rozhraní přímo na serveru. Server drží přístupové klíče k burzám, což je velmi kritická informace. Vzhledem k těmto skutečnostem byla komunikace mezi frontendem a backendem navržena jako nezabezpečená. Na konci této kapitoly je stručně diskutováno možné budoucí rozšíření (2.6.4.4).

V návrhu toho, jak se aplikace provozuje, se počítá s tím, že vizualizace a nástroje poskytované frontendem, by se využívaly jen pro testování a simulace v kontrolovaném prostředí. V produkčním prostředí by pak běžely pouze synchronizátory burzovních dat a strategie. V případě, že by uživatel chtěl využívat vizualizace v produkčním prostředí, je nutné frontend spustit lokálně a připojit jej k produkčnímu serveru přes SSH tunel. Zároveň je silně doporučeno socket.io server zapínat jen na dobu nezbytně nutnou a po skončení práce s lokálním frontendem jej na serveru vypnout.

2.6.4.4 Možné rozšíření pro zabezpečení socket.io komunikace

Samotný protokol socket.io v základu podporuje SSL. Samotná komunikace lze tedy zabezpečit proti odposlechu na úrovni transportního protokolu.

Autorizace by pak bylo vhodné implementovat standardizovanou cestou, například technologií OAuth 2.0. Při přihlašování do aplikace by uživatel zadal přihlašovací údaje, server by na základě nich vygeneroval časově omezený token, kterým by se frontend aplikace autorizovala [4].

2.6.5 Konfigurace aplikace

Návrh počítá s konfigurovatelností pomocí proměnných prostředí v souladu s principem: *Store config in the environment*[5]. Veškerou konfiguraci platformy (včetně konfigurace jednotlivých zásuvných modulů) je možno uvést v souboru `.env`.

Kapitola 3

Implementace platformy

Platforma byla rozdělena na dvě části frontend a backend, které byly implementovány samostatně.

3.1 Backend

3.1.1 Technologický stack

Backend platformy byl na základě analýzy (2.3.2) implementován v jazyce Python ve verzi 3.6 se striktně dodržovanou typovostí.

3.1.1.1 Manažer závislostí

Pro správu závislostí byl použit poměrně nový systém `pipenv`[16]. Tento manager obaluje tradiční `pip`[17]a přináší moderní koncepty známé z manažerů závislostí pro jiné jazyky jako je Composer (PHP) nebo Yarn (JavaScript). Zároveň interně vytváří virtuální prostředí `virtualenv`, do kterého nainstaluje závislosti. Tím vyřeší izolaci aplikace od prostředí systému a zabrání kolizím mezi různými verzemi balíčků.

Při zavádění závislosti do aplikace během vývoje dojde k zamknutí verze do souboru `Pipfile.lock`. Ten je distribuovaný společně s aplikací. Při instalaci závislostí na klientské straně pak dojde ke stažení přesných verzí, a tím se zabrání případným nekonzistencím mezi různými verzemi závislosti a aplikací.

3.1.1.2 Typový systém

Zkušenost ukazuje, že statické typování je velmi užitečné pro robustnost aplikace. Díky neustálé kontrole typů během (nebo před) kompilací dochází k odhalování chyb v propojení (tzv. *wiringu*) jednotlivých komponent.

[16]Dostupné z: `https://github.com/pypa/pipenv`

[17]Pip je nástroj pro instalování Python balíčků. Používá se buď přímo k instalaci balíčku do prostředí systému, nebo pro instalaci balíčku do izolovaného prostředí (tzv. `virtualenv`).

V mnoha původně dynamických jazycích je patrný trend zavádění statických typových systémů. V jazyce PHP je od verze 7 možné používat skalární typy v anotacích metod a byla doplněna možnost striktně definovat návratový typ metody, v jazyce JavaScript je patrný trend zavádění rozšíření jazyka FlowType a TypeScript.

Stejně tak Python, který je dynamicky typovaný, umožňuje od verze 3.5 psát *type hints*[18][6]. Tento typový systém byl v implementaci aplikace striktně vyžadován. Jeho kontrola byla prováděna pomocí nástroje MyPy[19].

3.1.1.3 Continuous integration

Pro vývoj softwaru je velmi výhodné, když je zpětná vazba co nejrychlejší. Jednou z technik, jak tento cyklus zpětné vazby (tzv. *feedback loop*) zkrátit, je neustálá integrace jednotlivých částí systému dohromady a testování jejich funkčnosti.

Pro tento účel byla využita služba CircleCI, která při napojení na Github umožňuje tuto integraci provádět automaticky při každé změně v kódu aplikace. Spouští se jednotkové a integrační testy a již zmíněná kontrola typů pomocí MyPy.

3.1.1.4 Testování

Pro psaní jak jednotkových tak integračních testů byl použit framework `pytest`[20] společně s mockovací knihovnou `fexmock`. Testy byly v implementaci psány průběžně a pokrývají téměř 80 % kódu. Pokrytí kódu testy se vyhodnocuje při každém spuštění testů v rámci *continuous integration*.

3.1.1.5 Logování

Pro logování se využívá vestavěného modulu `logging`. Logování probíhá podle standardu PEP 282 [7]. Každý Python modul v této práci má vlastní jmenný prostor. Tím se elegantně zpřehlední výstup logu a je možné zapínat logování jednotlivých modulů nezávisle na sobě.

3.1.2 Modelování domény

Doménové entity byly na základě analýzy v kapitole 2.4 vymodelovány zvlášť jako soubor submodulů v modulu `coinrat.domain`.

Pro každou z entit `Candle`, `Order` a `PortfolioSnapshot` bylo definováno rozhraní pro úložiště:

[19] `http://mypy-lang.org`

[19] Vyjadřování typů pomocí *type hints* je technika, kdy typy nejsou přímo v jazyce povinné a často ani kontrolované a to jak v čase kompilace, tak při běhu aplikace. Takto zapsané typy pak slouží jen jako pomůcka vývojáři a jejich kontrolu je třeba provádět samostatně.

[20] `https://docs.pytest.org/en/latest`

- `CandleStorage` – rozhraní umožňuje zapisovat a načítat svíčky. Klíčovou schopností je pak vracet svíčky agregované do různých velikostí (podle `CandleSize`).

  Způsob ukládání svíček je implementační detail každého zásuvného modulu. Platforma ale počítá s nejmenší možnou svíčkou o velikosti 1 minuta.

- `OrderStorage` – rozhraní předepisuje schopnost vytvářet, vyhledávat a mazat objednávky. Aktualizace objednávky není v současné implementaci rozhraní vyžadována a řeší se jejím smazáním a následným opětovným vložením.

- `PortfolioSnapshotStorage` – snapshot portfolia potřebuje platforma jen pro danou objednávku. Rozhraní tedy vyžaduje pouze metody pro uložení a načtení snapshotu podle dané objednávky.

Celá backendová aplikace pak pracuje s těmito rozhraními a konkrétní implementace jsou zodpovědností zásuvných modulů.

3.1.2.1 Import/Export doménových entit

Jako součást domény byla dále implementována schopnost serializace a deserializace všech doménových objektů do datového typu slovníku (v Pythonu typ `Dict`). Tento typ se pak dá triviálně převést na JSON, čehož se v aplikaci hojně využívá.

Pro dodržení principu jedné zodpovědnosti (*SRP – Single responsibility principle*[21]) byly tyto metody implementovány zvlášť mimo doménové entity.

Společně s těmito metodami byl naimplementován `CandleExporter` a `OrderExporter`. Tyto třídy umožňují s použitím výše zmíněných serializačních a deserializačních metod uložit svíčky a objednávky v daném časovém intervalu do souboru a následně je z něj načíst.

Hlavním využitím těchto tříd pro import a export dat je snapshotové testování obchodních strategií, kterým se zabývá kapitola 3.1.9 Snapshot testování strategií.

3.1.3 Přesnost čísel s desetinnou čárkou

Jelikož celý systém pracuje s penězi, je naprosto nezbytné vyhnout se reprezentacím hodnot v plovoucí desetinné čárce, aby nedošlo k nepřesnostem. Navíc ve světě kryptoměn je nutné pracovat s často velmi malými zlomky mincí

[21] *Single responsibility principle* (často uváděn pod zkratkou SRP) je v objektově orientovaném programování princip, podle kterého musí mít každý objekt jen jednu zodpovědnost (tedy být zodpovědný jen za jednu věc).

a důsledky i drobné chyby by se tím umocnily. Například nejmenší hodnota, kterou lze v kryptoměně bitcoin vyjádřit, je 1×10^{-8} BTC[22].

V implentaci se pro reprezentaci čísel s pevnou desetinnou čárkou striktně využívá typ `Decimal`, který je nativně obsažen v jazyce Python.

V případě nutné konverze hodnoty z reprezentace plovoucí čárky je vždy na vstupu zafixována přesnost na 9 desetinných míst.

3.1.4 Dependency injection

Kód, který obsahuje komplikovanější aplikační logiku je zapouzdřen do tříd (nazývaných služba). Podle principu jedné zodpovědnosti by pak každá služba měla být zodpovědná jen za jednu věc a delegovat jiné zodpovědnosti na další služby.

Aby bylo možné závislosti jednotlivých služeb vyměňovat (v případě využívání abstrakce nebo v testech), jsou jim předávány konstruktorem (tzv. *constructor dependecy injection*). Jedna třída (služba) tedy nikdy nevytváří instanci jiné, ale dostává ji jako závislost. Třída, jejíž zodpovědností je pak sestavení služby, se nazývá *dependecy injection container*. Obsahuje definice všech služeb v aplikaci a na požádání je umí sestavit. Tento kontejner si lze představit jako skupinu továrniček (*factories*), které umí na požádání sestavit každou službu se všemi jejími závislostmi.

Protože v Pythonu neexistuje žádná vhodná implementace tohoto kontejneru, byl kontejner implementován v rámci práce tak, aby vyhovoval jejím potřebám. Jedná se o jednoduchou třídu obsahující metodu pro vytvoření každé služby (tzv. *factory pattern*). Tím bylo dosaženo lazy-loadingu, který umožňuje vytvořit instance třídy až ve chvíli, kdy je to potřeba. Zároveň si kontejner hlídá, zda již má službu vytvořenou, a při druhém dotazu ji nevytváří znovu, nýbrž vrátí službu již existující (není-li explicitně řečeno jinak).

Dependency injection container je implementován obecně, a tak je umožněno, aby každý zásuvný modul definoval pro své potřeby vlastní konkrétní kontejner. Například výchozí modul `coinrat_influx_db_storage` pro ukládání dat takovýto kontejner obsahuje. Tyto moduly jsou blíže popsány v kapitole 4.1 Zásuvné moduly pro ukládání dat.

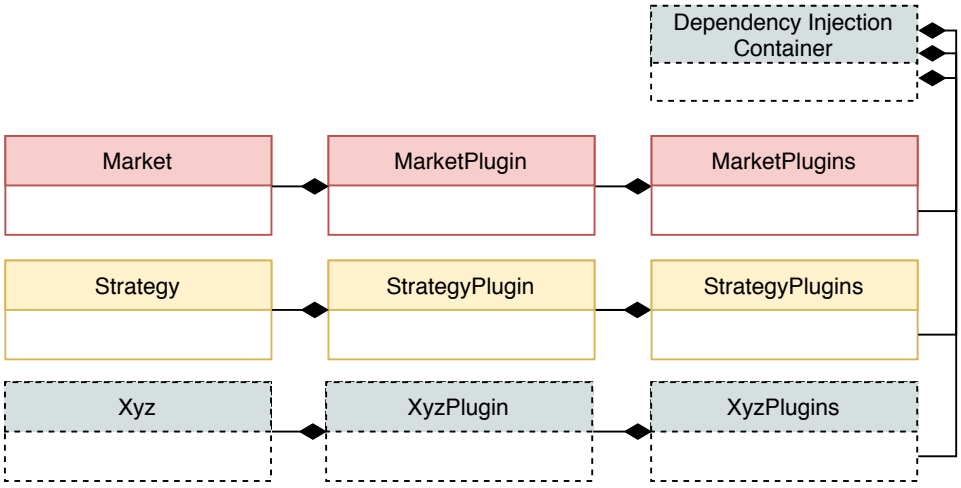
3.1.5 Zásuvné moduly

Pro zásuvné moduly byla použita knihovna Pluggy[23]. Ta zajišťuje, aby aplikace načetla zásuvné moduly, aniž by je musela přímo importovat, a nedochází tak k cyklické závislosti. Pluginy jsou definovány standardně v souboru `setup.py` v sekci `entry_points`.

Na obrázku 3.1 je znázorněno, jak pro každý typ zásuvného modulu existuje služba (`XyzPlugins`), která je dostupná v třídě *dependency injection con-*

[22] Nejmenší fragment bitcoinu se nazývá Satoshi podle autora této kryptoměny.

[23] `https://pypi.python.org/pypi/pluggy`



Obrázek 3.1: Diagram kompozice tříd pro zásuvné moduly

tainer a zprostředkovává přístup k jednotlivým zásuvným modulům. Každý modul pak může poskytovat jednu nebo více servisních tříd (například `Market`, `Strategy`, ...).

Dependency injection container je zodpovědný za vytvoření instancí služeb pro načítání zásuvných modulů. Tyto služby jsou pak zodpovědné za vytvoření instancí samotných modulů (podle konfigurace v souboru `setup.py`). Instance zásuvných modulů jsou poté zodpovědné za vytváření instancí doménových služeb (`Market`, `Strategy`, ...).

3.1.6 Práce s časem

3.1.6.1 Časová pásma

V Pythonu, stejně jako v mnoha dalších jazycích, je časové pásmo, ve kterém je datum a čas reprezentován součástí rozhraní datetime-objektu. Při nesprávné manipulaci s datetime-objektem je pak velmi snadné dospět k chybnému výsledku.

Všechny doménové objekty vyžadují čas v UTC kvůli odolnosti proti chybám, které vznikají díky časovým pásmům.

3.1.6.2 Aktuální čas

Jelikož získání aktuálního času je *impure fuction*[24], je třeba tuto funkci izolovat jako závislost. Díky tomu je možné ovlivnit hodnotu aktuálního času

[24] *Impure fuction* je funkce, jejíž výstup závisí kromě vstupních parametrů ještě na dalších proměnných (tzv. *free variables*). Takovými proměnnými jsou typicky čas nebo náhoda.

jak v testech, tak při spouštění strategií v režimu simulace. Možnost ovlivňovat informaci o aktuálním čase umožňuje elegantně spouštět strategie nad historickými daty.

Pro tento účel byla zavedena služba `DateTimeFactory`, která je v kódu popsána stejnojmenným rozhraním (*interface*). Pro toto rozhraní byly vytvořeny dvě implementace `CurrentUtcDateTimeFactory` a `FrozenDateTimeFactory`.

Simulátor strategií pak může předat strategii službu, která může mezi jednotlivými kroky strategie posouvat čas a simulovat tak průběh na historických datech.

Na obrázku 3.2 je vidět, jak různé spouštěče strategií využívají různé `DateTimeFactory`.

3.1.7 Rozhraní pro obchodní strategie

Každá strategie ze zásuvného modulu má dostupné tyto závislosti: službu pro čtení svíčkových grafů, rozhraní pro vytváření a zjišťování stavu objednávek, službu zprostředkovávající aktuální čas a instanci objektu s konfigurací běhu strategie (`StrategyRun`).

Dependency injection container v současné implementaci neobsahuje autowiring[25] a ani dynamickou konfiguraci. Není tedy možné definovat různé závislosti pro různé strategie.

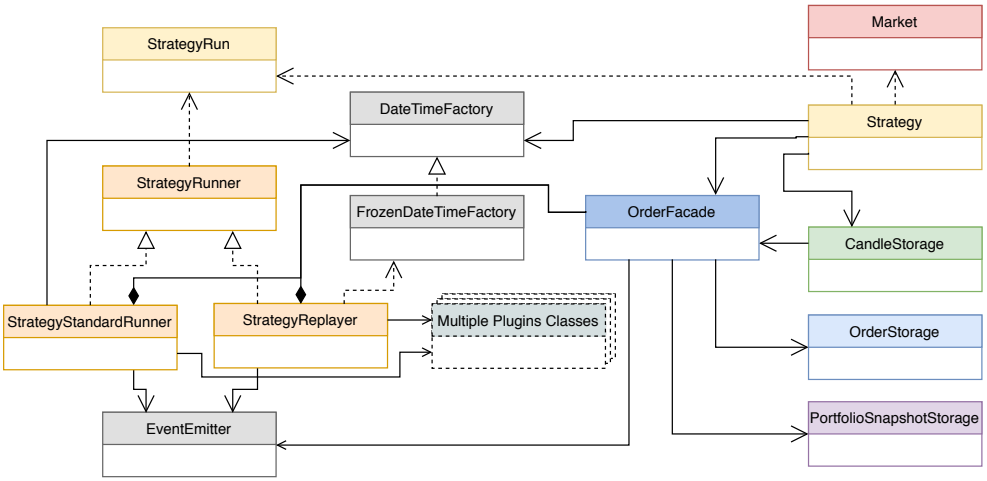
Strategie má předepsané toto rozhraní:

- `tick()` – metoda, která je provolána každých N-sekund. Strategie má možnost vyhodnotit situaci a případně udělat objednávku.
- `get_configuration_structure()` – každá strategie má svou specifickou konfiguraci. Touto metodou vrací definici této konfigurace a uživateli je podle této definice nabídnuta možnost spustit strategii s jím definovanými konfiguračními parametry.
- `get_seconds_delay_between_ticks()` – metoda, která vrací informaci o tom, jak často je třeba spouštět `tick` strategie. Často se odvíjí od konkrétní konfigurace (např. velikost svíček).

3.1.8 Spouštění strategie

Spouštěč strategií je definován rozhraním `StrategyRunner`, který má dvě implementace. První implementací je `StrategyStandardRunner` pro běh strategie oproti burze a druhou `StrategyReplayer` pro simulovaný běh.

[25] Jedná se o schopnost *dependency injection* kontejneru umožňující sestavení závislostí pro službu na základě informací dostupných buď v konfiguraci nebo přímo v deklaraci konstruktoru samotné služby.



Obrázek 3.2: Diagram tříd pro spouštění strategie

Pro `StrategyRunner` je vstupem objekt `StrategyRun`, který reprezentuje spuštění strategie. Na základě tohoto objektu získá `StrategyRunner` správné závislosti a předá je službě `StrategyPlugins` k sestavení strategie. `StrategyRunner` následně strategii spustí a zodpovídá za řízení jejího běhu. Na obrázku 3.2 je celé uspořádání znázorněno.

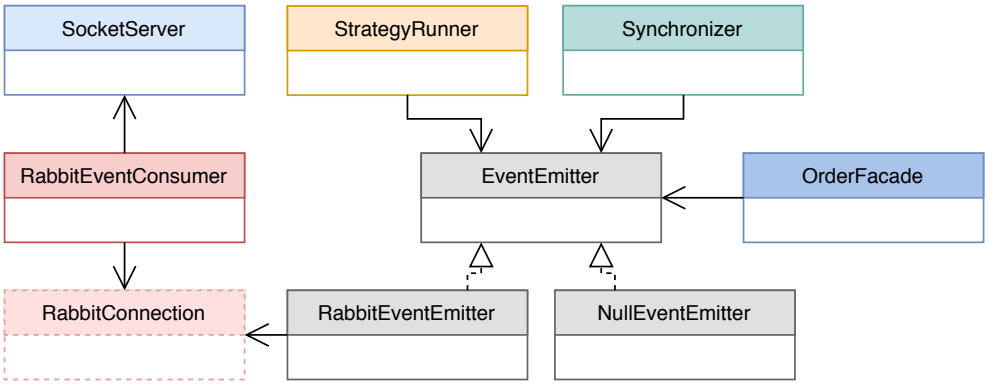
Dále stojí za povšimnutí, že objekty spouštěčů strategií nedostávají `OrderFacade` jako závislost, ale jsou přímo zodpovědné za vytvoření její instance. `OrderFacade` se totiž liší podle konfigurace každého běhu strategie. Spouštěč tedy na základě konfigurace, obsažené ve `StrategyRun` získá konkrétní závislosti z *dependency injection containeru* a z nich službu `OrderFacade` sám sestaví.

3.1.9 Snapshot testování strategií

Snapshotové testování je technika, která umožňuje poměrně snadno testovat systém s komplexním výstupem. Výstup se porovná proti tzv. snapshotu, ověřenému výsledku toho samého kódu. Výhodou je snadné vytváření takovýchto testů a to, že umožňují pokrýt poměrně velkou část kódu s minimální námahou. Nevýhodou je, že díky tomu, že test pokrývá velkou část kódu agregovaně, je snadné rozbít takovýto test i zdánlivě nesouvisející změnou. Je tedy poměrně obtížné takovýto test opravovat a dohledávat chybu v kódu. Test dává velmi spolehlivě signál o tom, že něco nefunguje správně, ale již neposkytuje přesnou lokalizaci této chyby.

V této práci byly implementovány nástroje určené pro testování strategií. Vstupem běhu strategie je snapshot svíček za určitou dobu. Výstupem strategie je série objednávek, která se porovná proti výstupnímu snapshotu.

Tyto testy velmi dobře detekují, že došlo k nějaké změně v chování stra-



Obrázek 3.3: Diagram tříd pro vyvolání a zpracování událostí

tegie. Velmi těžko se ale hledá, v čem tato změna spočívá. I přesto jsou velmi užitečným nástrojem při úpravách, u kterých by se výsledky strategie neměly změnit a velmi dobře detekují neočekávanou změnu chování strategie.

3.1.10 Události a úkoly

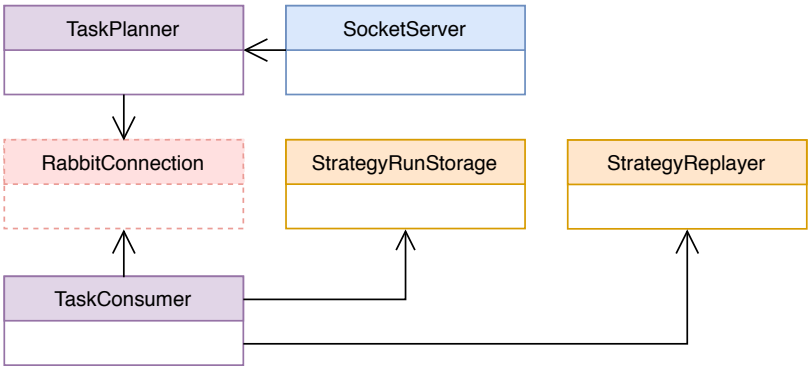
Jelikož systém pracuje v několika procesech, je nutné je synchronizovat. K tomuto účelu byly implementovány dva mechanismy: události (`Events`) a úkoly (`Tasks`).

3.1.10.1 Události

Události reprezentují důležité situace v systému. Například vytvoření objednávky nebo synchronizace nové svíčky. O jejich vyvolání se stará třída `EventEmitter`. Tato třída má dvě implementace. Jednu zapisující události do fronty v RabbitMQ a druhou, která s nimi nedělá nic a ignoruje je.

Tato druhá implementace slouží pro vypnutí událostí v případě, že jich není třeba. Tato situace nastává například v produkčním modu, když není spuštěn socket-server. Technika, pomocí které dojde k vytvoření objektu, který nic nedělá, ale splňuje rozhraní nějakého logického celku, se nazývá *null object pattern*. Jedná se o velmi elegantní způsob, jak ovlivnit chování aplikace bez nutnosti zásahu do její vnitřní implementace.

Na obrázku 3.3 je možno vidět, jak jsou třídy pro vyvolávání a zpracovávání událostí zakomponovány. Třídy obsahující business logiku vyvolávají události skrze `EventEmitter`, který je případně ukládá do RabitMQ. Odtamtud je vyzvedává `RabbitEventConsumer`, který je skrze `SocketServer` odesílá do frontendu. Více se komunikaci s frontendem věnuje kapitola 3.1.11 Komunikace s frontendem.



Obrázek 3.4: Diagram tříd pro vyvolání a zpracování úkolů

3.1.10.2 Úkoly

Jelikož spuštění simulace strategie je poměrně náročné, bylo třeba jej provádět asynchronně. Uživatel na frontendu spustí simulaci strategie. Tento úkol se uloží do fronty v RabbitMQ. Následně je vyzvednut a zpracován jiným procesem. Kompozici tříd pak blíže popisuje obrázek 3.4. Na tomto obrázku je možno vidět, jak třída reprezentující proces pro zpracování úkolu (`TaskConsumer`) závisí na třídách pro spuštění simulace strategie.

Pro další rozvoj aplikace, díky kterému pravděpodobně přibudou další typy úkolů, by bylo vhodné třídu `TaskConsumer` dekomponovat, aby za zpracování každého typu úkolu byla zodpovědná samostatná třída. Třída reprezentující proces by se pak starala jen o rozdělení úkolů z RabbitMQ fronty podle typu na tyto dílčí třídy.

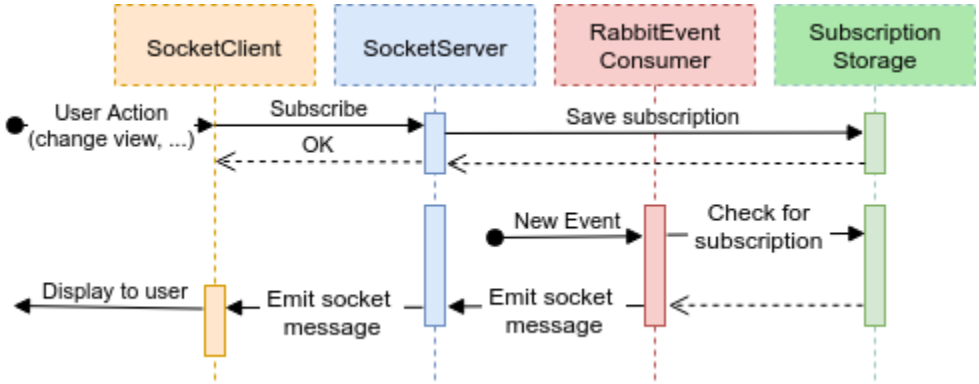
3.1.11 Komunikace s frontendem

Komunikaci s frontendem zajišťuje třída `SocketServer`. Ta poskytuje rozhraní pro synchronní načítání dat a zároveň aktivně odesílá nová data do frontendu. Aby frontend dostával jen ta data, která potřebuje, byl naimplementován publish-subscribe mechanismus. Komunikace je znázorněna na obrázku 3.5.

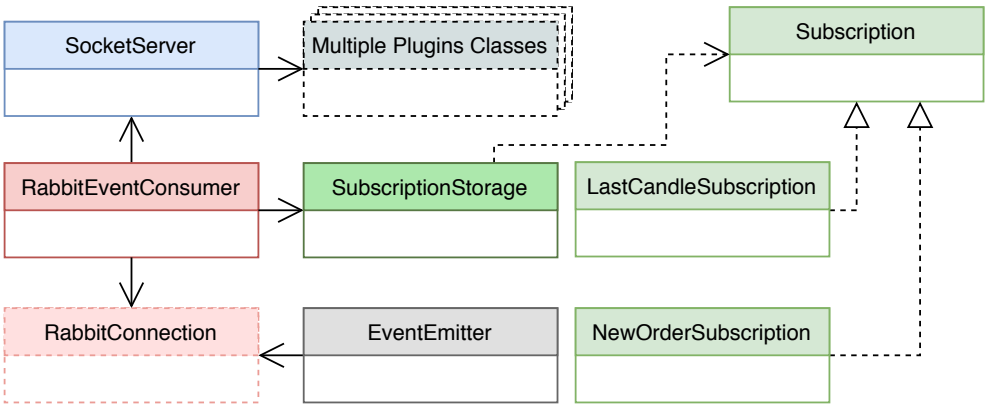
Frontend odešle na backend informaci, o jaký typ dat má zájem (velikost svíček, časový rozsah, pár, ...). Backend si tuto informaci uloží a následně události filtruje a odesílá do frontendu jen ty, ke kterým se přihlásil. Obdobně funguje i odhlášení, jen s tím rozdílem, že vždy proběhne kompletní odhlášení od všech odběrů a klient se musí znovu přihlásit ke všem, o které má zájem.

Třída `RabbitEventConsumer` zpracovává všechny události, které byly v systému vyvolány, vyfiltruje je podle toho, k čemu se klient přihlásil a předává je socket-serveru k odeslání na frontend. Diagram kompozice tříd je znázorněn na obrázku 3.6.

Samotné filtrování je naimplementováno zcela obecně tak, aby bylo možné jej snadno rozšiřovat o další typy událostí. K tomu slouží stavová služba



Obrázek 3.5: Sequence diagram pro publish-subscribe mezi frontendem a backendem



Obrázek 3.6: Diagram tříd pro publish-subscribe komunikaci s frontendem

`SubscriptionStorage`, která udržuje všechna přihlášení k odběru v paměti.

Existují různé typy událostí a pro každý z nich implementace rozhraní `Subscription`. Když je následně v systému vyvolána událost, jednotlivé objekty `Subscription` samy poznají, že jsou přihlášené k jejímu odběru (implementováno v metodě `is_subscribed_for`).

Když se frontend přihlašuje k odběru, odesílá typ události a metadata k filtrování. Za vytvoření objektu `Subscription` podle typu události je zodpovědná oddělená továrna obsažená zvlášť v souboru `subscription_factory.py`.

Ukázka rozhraní služby `SubscriptionStorage` a objektů `Subscription` je uvedena zde (obr. 3.1).

Třídy `RabbitEventConsumer` a `SocketServer` jsou spuštěny každá ve vlastním vlákně v rámci jednoho procesu. Obě musí zároveň udržovat spojení mimo

```
class Subscription:
  def is_subscribed_for(
    self, event_name: Union[str, None] = None,
    event_data: Union[Dict, None] = None
  ) -> bool

class SubscriptionStorage:
  def subscribe(self, subscription: Subscription)

  def find_subscriptions_for_event(
    self,
    event_name: str,
    event_data: Union[Dict, None] = None
  ) -> List[Subscription]

  def unsubscribe(self, event_name: str, session_id: str)
```

Ukázka 3.1: Rozhraní třídy SubscriptionStorage

aplikaci (Socket.io a RabbitMQ) a příchod události z RabbitMQ musí mít možnost vyvolat odeslání zprávy do frontendu skrze socket.io.

K jakým událostem se frontend přihlásil, je v současné implementaci uloženo pouze v paměti. Při restartu backendu je třeba, aby se frontend znovu přihlásil k odběru.

3.1.12 Agregace svíček

Jelikož všechna burzovní data jsou ukládána v nejmenší granularitě minutových svíček, je nutné provádět jejich agregaci do svíček s větším intervalem. Backend pro reprezentaci této vlastnosti obsahuje již zmíněnou entitu `CandleSize`.

Pro čtení svíček pak rozhraní `CandleStorage` obsahuje možnost definovat velikost svíček. Samotná implementace agregace je zodpovědností každého modulu, jelikož může těžit z optimalizace databázové vrstvy.

V rámci této práce byl implementován zásuvný modul pro ukládání dat nad databází InfluxDB, který je podrobně popsán v kapitole 4.1 Zásuvné moduly pro ukládání dat.

V rámci platformy je zajištěno, že jednotlivé intervaly svíček vždy beze zbytku dělí hodiny, resp. dny. Tímto omezením se platforma snaží zajistit, aby jednotlivé svíčky vždy korespondovaly s přirozenými jednotkami času. Ukázáno na příkladu: aby patnáctiminutová svíčka byla vždy agregací minutových svíček od první svíčky v hodině až po patnáctou svíčku v hodině.

Samozřejmě platí, že platforma není schopna ověřit dodržení tohoto pravidla pro agregace a je tedy na autoru zásuvného modulu, aby ji dodržel.

Použitá databáze InfluxDB tvoří agregace podle času výše popsaným způsobem. V následující citaci z dokumentace [8] je způsob agregace podrobněji

vysvětlen.

> InfluxDB queries the `GROUP BY time()` intervals that fall within the `WHERE` time clause. `GROUP BY time()` intervals always fall on rounded calendar time boundaries. Because they're rounded time boundaries, the start and end timestamps may appear to include more data than those covered by the query's `WHERE` time clause.

V případě agregace podle násobků dní přes větší interval než je jeden rok může dojít k posunu v druhém roce. Vzhledem k tomu, jak složité je tento problém vyřešit a o jak okrajový případ se jedná, nebylo řešení tohoto problému do práce zahrnuto.

Pro účely testování a případně pro využití dalšími moduly obsahuje entita `CandleSize` metodu `get_interval_for_datetime`, která pro daný bod v čase poskytuje interval, do kterého daný bod při požadované velikosti svíčky spadá.

3.1.13 Rozhraní příkazové řádky

Backendová aplikace disponuje rozhraním v příkazové řádce a je ji možno spouštět na frontendu zcela nezávisle.

Pro parsování vstupu byla použita knihovna `Click`[26]. Jedná se o aktuálně nejvyspělejší implementaci (tzv. *state of the art*) pro psaní rozhraní příkazové řádky v Pythonu.

Byly naimplementovány následujícími příkazy:

- `candle_storages` – seznam dostupných úložišť pro svíčky.
- `database_migrate` – spuštění migrací struktury databáze. Některá data nejsou ukládána do úložišť poskytovaných zásuvnými moduly a místo toho jsou složena v SQL databázi. Tento příkaz umožňuje měnit strukturu databáze mezi jednotlivými verzemi aplikace.
- `export_candles` – export svíček do JSON souboru.
- `export_orders` – export objednávek do JSON souboru.
- `market` – zobrazí detail daného marketu v daném zásuvném modulu, poskytuje strukturu konfiguračních parametrů.
- `markets` – seznam dostupných marketů v jednotlivých zásuvných modulech.
- `order_storages` – seznam dostupných úložišť pro objednávky.
- `portfolio_snapshots` – seznam dostupných úložišť pro snapshoty marketů.

[26] `https://pypi.python.org/pypi/click`

- `run_strategy` – spuštění běhu strategie. Parametry jsou název strategie, burzovní pár a seznam marketů, které je třeba strategii předat (většinou jeden, ale pro meziburzovní arbitráže je jich možno zadat více).
- `start_server` – spuštěná socket.io serveru pro připojení frontendu.
- `start_task_consumer` – spuštění workera pro zpracování úkolů (například simulace strategie).
- `strategies` – seznam dostupných strategií.
- `strategy` – zobrazí detail strategie, poskytuje strukturu konfiguračních parametrů.
- `synchronize` – spuštění procesu synchronizování dat pro daný synchronizátor, burzu a pár.
- `synchronizers` – seznam dostupných synchronizátorů pro stahování dat z burz (svíček).

3.2 Frontend

3.2.1 Technologický stack

Backend platformy byl na základě analýzy (2.3.3) implementován v jazyce JavaScript s použitím technologie React a Flow.

Pro správu stavu byla použita poměrně mladá technologie MobX[27], která je alternativou dnes již zaběhlé technologie Redux[28]. Její hlavní výhodou oproti technologii Redux je, že je stručnější na zápis a nevyžaduje od programátora psaní velkého množství kódu pro stejnou funkcionalitu.

Pro Continuous Integration bylo využito služby CircleCI stejně jako v případě backendu (3.1.1.3).

Na sestavování závislostí byl použit dnes již zaběhlý nástroj `yarn`[29] (jedná se o alternativu ke klasickému `npm`[30]).

Stejně jako v případě backendu je frontend konfigurován pomocí proměnných prostředí. Pro jejich zpracování se využívá knihovny `dotenv`[31].

3.2.1.1 React

React umožňuje vytvářet šablony pomocí kompozice komponent, které svým zápisem připomínají značkovací jazyk HTML.

[27] `https://github.com/mobxjs/mobx`
[28] `https://redux.js.org`
[29] `https://yarnpkg.com`
[30] `https://www.npmjs.com`
[31] `https://github.com/motdotla/dotenv`

```
const ColoredDotComponent = ({color}: Props) => {
  return <span
    className="pt-icon pt-icon-dot"
    style={{color: color}}
  />
}
```

Ukázka 3.2: React komponenta

```
return <Table numRows={orders.length}>
  <Column
    name="id"
    renderCell={(row: number) => {
      return <Cell>{orders[row].orderId}</Cell>
    }}
  />
</Table>
```

Ukázka 3.3: JSX se zanořením komponent

Kód se organizuje do tzv. komponent. Jedná se o jednoduchou funkci jejímž vstupem jsou konfigurační parametry a výstupem HTML kód. Alternativně jsou tyto komponenty zapisovány jako třídy s metodou `render`.

Tímto se kód dekomponuje do malých logických celků, které se snadno rozšiřují a udržují. V ukázce kódu 3.2 je vidět, jakým způsobem se kód komponent zapisuje.

Jedná se o JavaScript kód rozšířený o zápis podobný XML, který umožňuje psát HTML a komponenty dohromady. Zároveň je možno vkládat JavaScript proměnné. Tento zápis se nazývá JSX. Komponenty lze do sebe elegantně zanořovat stejně, jako jsou do sebe zanořeny HTML tagy. V ukázce 3.3 je znázorněno, jak na příkladu komponenty tabulka takovéto zanoření funguje.

V ukázce 3.2 stojí za povšimnutí atribut `renderCell`, skrze který se do komponenty `Column` předává callback pro vykreslení jedné buňky tabulky. Tento callback taktéž obsahuje JSX kód pro vykreslení buňky.

Výše uvedený JSX kód 3.3 je pomocí BabelJS[32] převeden do následujícího výstupu zobrazeném v 3.4.

Jedná se tedy o stromovou strukturu zanořených funkcí, které se umí zmaterializovat na základě svých parametrů do HTML.

Při změně parametrů dojde k překreslení všech sub-komponent pouze v paměti (tzv. *vitual DOM*). Následně React identifikuje ty části UI ve *virtual DOM*, které je nutné skutečně překreslit. Ty jsou následně překresleny v DOMu v prohlížeči. Tento proces se nazývá *reconciliation* [9] a díky němu dosahuje React velmi dobrých výkonnostních výsledků.

[32] `https://babeljs.io`

```
1  React.createElement(
2    Table,
3    { numRows: orders.length },
4    React.createElement(Column, {
5      name: "id",
6      renderCell: function renderCell(row) {
7        return React.createElement(
8          Cell,
9          null,
10         orders[row].orderId
11       );
12     }
13   })
14 );
```

Ukázka 3.4: JavaScript kód převedený z JSX

3.2.2 Testování

Pro psaní jak jednotkových, tak integračních testů byl použit framework `Jest` od společnosti Facebook. Hojně bylo využíváno snapshotových testů pro testování React-component. Jedná se o stejnou techniku, jaká je popsána v kapitole 3.1.9, avšak aplikovanou na vykreslování React komponent. Test vykreslí komponentu do HTML a tento výstup porovná s očekávaným snapshotem.

3.2.3 Organizace kódu

Kód byl rozdělen na dvě skupiny. V první je kód, který závisí na konkrétním běhu aplikace a přímo ovlivňuje stav objektů v běžící aplikaci. V druhé je kód, který je nezávislý na konkrétním běhu aplikace a teoreticky je znovupoužitelný v jiném kontextu.

Kód z první skupiny byl uložen do složky `App`, jelikož se jedná o kód aplikaci vlastní.

Ostatní kód byl organizován podle principu *domain-first*, který zlepšuje přehlednost aplikace a především přirozeně koncentruje kód, který na sobě silně závisí, na jedno místo. Tento princip se nazývá *high cohersion*. Při nutnosti rozšířit implementaci nějaké části není nutné hledat všechna místo v kódu, ale všechen kód je koncentrován na jednom místě.

3.2.3.1 Nezávislost UI komponenty

V této práci se striktně dodržuje to, že každá komponenta dostává svůj stav přes tzv. *props* (alternativní konstruktor) a nijak neovlivňuje cokoliv mimo sebe sama. Komponenta tedy nikdy nezávisí na aplikaci.

V případě, že je třeba, aby událost v komponentě ovlivnila stav mimo komponentu, je to vždy realizováno callbackem, který komponenta dostane skrz *props*.

V aplikaci se tak snižuje provázanost (princip *low coupling*). Jasně definované závislosti komponent usnadňují testování a znovupoužitelnost.

3.2.3.2 Použití komponent v aplikaci

Všechny komponenty, které interagují s jinými komponentami nebo s globálním stavem aplikace, byly obaleny tzv. kontejnery. Technicky se také jedná o React komponentu, avšak tato komponenta nemá žádný UI prvek kromě čisté komponenty, kterou obaluje. Zodpovědností tohoto kontejneru je tedy pouze sestavit závislosti pro obalovanou komponentu a napojit ji na stav aplikace.

3.2.3.3 Sestavení závislostí

Obdobně jako v případě backendu byly závislosti sestavovány technikou *dependency injection*, a to výhradně přes konstruktor tříd. Na rozdíl od backendu je zde Dependency Injection Container implementován jednodušeji.

Sestavení závislostí probíhá v souboru `diContainer.js`. Toto sestavení je implementováno bez lazy-loadingu.

Jednotlivé instance služeb jsou pak přímo importovány do kontejnerů pro sestavení závislostí React komponent.

3.2.4 Doména

Doménové entity ve frontendu (objednávka, market, pár, ...) jsou vesměs stejné jako v backendu. Pouze jsou zjednodušené pro účely frontendu, který od nich většinou nevyžaduje žádnou logiku. Jejich účelem je především poskytovat data ve strukturované podobě k vizualizaci.

Jediným výrazným rozšířením je entita `OrderDirectionAggregate`, která reprezentuje agregaci objednávek přes určitý čas. Je nutná pro vizualizaci svíček a objednávek v jednom grafu, přičemž velikost svíček může být různá.

3.2.5 Agregace objednávek

Jelikož objednávek je oproti svíčkám velmi malý počet, jejich agregace do svíčkového grafu probíhá na frontendu. Algoritmus této agregace je implementován ve funkci `createAggregateFromData`.

Vstupem je pole svíček a objednávek, které jsou již ve storu uloženy roztříděné na nakupující a prodávající.

Algoritmus nejprve seřadí všechna pole podle času a poté iteruje ve vnějším cyklu přes svíčky a ve dvou vnitřních cyklech přes objednávky. Vždy když je objednávka zařazena do přihrádky podle intervalu svíčky, je vyřazena z pole. Pseudokód algoritmu je popsán v 3.5.

```
sort(candles)
sort(buyOrders)
sort(sellOrders)

for (let i = 0; i < candles.length; i++) {
  const key = calculateBucketKey(candle[i])
  data[key].candle = candle[i]

  let lastBuyOrder = buyOrders[buyOrders.length - 1]
  while (lastBuyOrder !== undefined
    && checkShouldBeInBucket(key, lastBuyOrder)
  ) {
    data[key].buyOrder.aggregate(lastBuyOrder)
    buyOrders.pop()
    lastBuyOrder = buyOrders[buyOrders.length - 1]
  }

  let lastSellOrder = sellOrders[sellOrders.length - 1]
  while (lastSellOrder !== undefined
    && checkShouldBeInBucket(key, lastSellOrder)
  ) {
    data[key].sellOrder.aggregate(lastSellOrder)
    sellOrders.pop()
    lastSellOrder = sellOrders[sellOrders.length - 1]
  }
}
```

Ukázka 3.5: Pseudokód agregaci objednávek do svíčkového grafu

Díky tomu, že jsou svíčky i objednávky seřazeny a že se již zařazené objednávky znovu neprochází, je asymptotická složitost algoritmu $O(n \cdot log(n))$ protože:

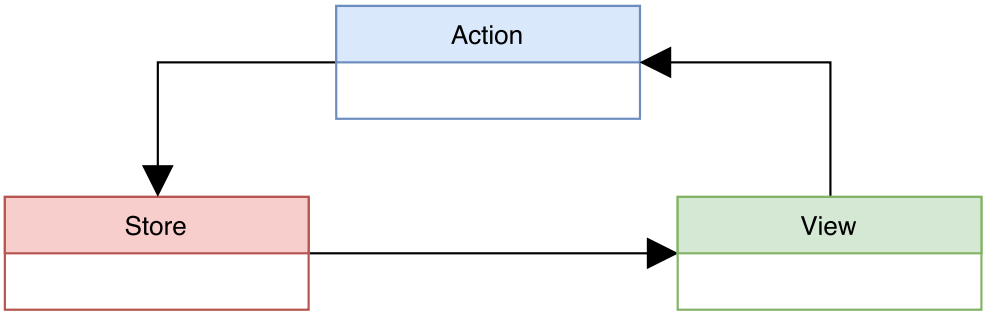
$$O(n_c + n_{ob} + n_{os}) + O(n_c \cdot log(n_c)) + O(n_{ob} \cdot log(n_{ob})) + O(n_{os} \cdot log(n_{os})) \quad (3.1)$$

Kde n_c je počet svíček a n_{ob}, resp. n_{ob} je počet kupujících, resp. prodávajících objednávek. A nikoliv kvadratická:

$$O(n_c \cdot (n_{ob} + n_{os}) \quad (3.2)$$

Interpret JavaScriptu v prohlížečích řadí pole buď QuickSortem (a malá pole InsertionSortem) [10], nebo MergeSortem [11], které mají oba asymptotickou složitost $O(n \cdot log(n))$.

Funkce `checkShouldBeInBucket` a `aggregate` mají konstantní složitost.



Obrázek 3.7: Jednosměrný tok dat v React aplikaci

3.2.6 Tok dat v aplikaci

Dříve bylo běžné, že každá akce ve webové aplikaci změní stav a zároveň překreslí HTML. Bylo nutné si pečlivě hlídat, aby nedošlo k nekonzistenci mezi stavem aplikace a tím, co je uživateli prezentováno. Často se také ukládal stav aplikace přímo do HTML. Tím docházelo k velké provázanosti a bylo těžké celý systém udržovat.

Tento problém elegantně řeší princip *unidirectional data flow*. Kód je rozdělen na tři oblasti podle toho, jak pracuje s daty:

- **Store** – uchovává stav aplikace a zajišťuje jeho konzistenci.
- **Action** – modifikuje stav aplikace. Reprezentuje transformaci z jednoho stavu do jiného.
- **View** – prezentuje stav uživateli a obsahuje rozhraní pro vyvolávání akcí. View je *vyjádřením* stavu pro uživatele. Jedinou jeho zodpovědností je vykreslit DOM[33].

Jak je znázorněno na obrázku 3.7, view je vždy překresleno jen v návaznosti na změnu stavu. Akce je spuštěna vždy jen událostí vycházející z view. Stav je změněn jen na základě vykonání akce. Každý blok má jen jeden vstup a jeden výstup.

3.2.7 Stav

V aplikaci jsou dva typy stavů:

- **Stav každé komponenty.** Tento stav je izolovaný v komponentě a žádný objekt mimo komponentu jej nemůže číst ani přímo měnit. Často se

[33]DOM (Document Object Model) je objektová stromová struktura pro reprezentaci HTML.

jedná o neměnný stav a jediný způsob, jak takovýto stav zvenčí ovlivnit, je překreslit komponentu s jinými *props*. Někdy je to stav, který komponenta využívá pro svoje vykreslení nebo vykreslení subkomponent. Stále ale platí, že tento stav nikdy není dostupný z nadřazených komponent. Do vnořených komponent se předává vždy skrze *props*.

- **Globální stav aplikace.** Toto jsou data, která je třeba číst a měnit z více komponent. Tento stav je organizován do objektů zvaných *stores*. Tyto objekty ukládají stav aplikace (nebo jeho část) ve své vnitřní struktuře a svým rozhraním zajišťují jeho konzistentnost.

Jelikož je stav přímo v komponentě velmi izolovaný a práce s ním je triviální, zbytek této kapitoly se zabývá jen globálním stavem aplikace.

Objekty pro ukládání stavu jsou rozděleny, podle domény, stejně jako všechen ostatní kód. Ty *stores*, jejichž stav může být ovlivněn z backendu skrze socket.io komunikaci, mají závislost na příslušné službě skrze kompozici.

3.2.7.1 MobX

Tato technologie propojuje komponenty a story skrze *observer pattern*. V případě této práce jsou takto propojeny pouze kontejnery, jelikož čisté komponenty nesmí záviset přímo na stavu aplikace (blíže popsáno v kapitole 3.2.3.3 Sestavení závislostí).

Kontejner je tedy dekorován[34] funkcí `observer`, která hlídá, zda nedošlo ke změně ve stavu aplikace a vyvolá v případě potřeby překreslení komponenty.

Ve store objektech jsou pak příslušné struktury obsahující data obaleny strukturami, které jsou tzv. *observable*.

Při změně stavu hodnoty ve storu dojde k notifikaci všech komponent, které danou hodnotu observují.

Jedinou nevýhodou MobXu, (která se v této práci bohužel projevila) je, že tyto vazby mezi změnou stavu a překreslením komponenty nejsou explicitně vyjádřené v kódu, čímž dochází ke snížení čitelnosti.

Dobrým příkladem této neprůhledné situace je doménová entita `ConfigurationDirective`, která obsahuje proměnnou `_value`. Tato entita je navíc zapouzdřená v entitách `Market` a `Strategy`. Změna této proměnné `_value` pak vyvolává překreslování `TextInput` komponenty až poměrně hluboko v komponentovém stromu, kam navíc protéká přes *props* původně ze stavu entity `Market`.

Z tohoto příkladu je patrné, že na jednu stranu poskytuje MobX silný nástroj, který s malým množstvím kódu umožňuje řešit komplexní problémy, na druhou stranu ale obnáší nižší expresivitu v kódu.

[34] *Observer pattern* je návrhový vzor, který rozšiřuje funkcionalitu objektu nebo metody tím, že se obalí jiným objektem nebo metodou se stejným rozhraním.

3.2.8 Socket.io

Pro komunikaci s backendem skrze socket.io byla vytvořena služba `AppSocket`, která zastřešuje veškerou komunikaci s backendem. Tu následně používají skrze kompozici konkrétnější služby jako například `OrdersSocket` nebo `CandleSocket`.

Služby `OrdersSocket` a `CandleSocket` jsou zodpovědné za mapování dat přicházející v JSON formátu z backendu na objekty. Store, který tyto služby používá, jim předává callback funkci, skrze kterou dochází k uložení objektů do storu samotného.

Při startu aplikace dojde k načtení všech dat z backendu. Toto načtení je iniciováno metodou `componentDidMount`, která je zavolána při vykreslení komponenty. Tím je zajištěno, že se vždy načtou ta data, která jsou potřebná pro danou obrazovku.

Při prvotním načtení dat dojde k přihlášení odběru dat z backendu. Služby `OrdersSocket` a `CandleSocket` odešlou zprávu s informací, jaký typ událostí a s jakými filtry chtějí dostávat. Backend následně aktivně posílá vybrané události (nově příchozí svíčky, objednávky, ...).

Způsob, jakým backend zpracovává přihlášení odběru událostí, byl podrobně popsán v kapitole 3.1.11 Komunikace s frontendem.

Další načítání dat je pak iniciováno událostí v komponentě. Tuto událost příslušný kontejner deleguje na instanci storu v aplikaci (typickým příkladem je metoda `reloadByFilter` ve storu `OrderStore`). Ta následně skrze služby zmíněné na začátku této kapitoly (v tomto příklade `OrdersSocket`) načte data a uloží je do svého storu. Změna ve storu následně vyvolá překreslení příslušných komponent, které na této části stavu závisí.

3.2.9 Uživatelské rozhraní

3.2.9.1 Blueprint.js

Pro prvky UI byl zvolen framework Blueprint.js[35]. Jedná se o již nastylovaný soubor React komponent, který řeší téměř všechny opakující se problémy s UI. Bleprint.js obsahuje formulářové prvky, menu, tlačítka, ikony a mnoho dalšího.

V ukázce kódu 3.6 je uveden příklad použití Blueprint.js. Zodpovědností komponenty `HelpIconComponent` je vykreslit ikonu otazníku s tooltipem obsahujícím nápovědu. Díky knihovně Blueprint.js je elegantně dosaženo této funkcionality použitím komponenty `Icon` obalené komponentou `Tooltip2`.

Komponentu `Icon` lze konfigurovat pro vzhled (barvu, velikost) i obsah (obrázek ikony). Obdobně snadno lze využívat i ostatní komponenty. Se znalostí dostupných CSS stylů poskytovaných knihovnou lze v případě potřeby implementovat komponentu vlastní. V této práci byla takto implementována komponenta `NumberComponent` zobrazující čísla ve finančním formátu.

[35] `http://blueprintjs.com`

```
1  import React from "react"
2  import {Icon, Intent} from "@blueprintjs/core"
3  import {Tooltip2} from "@blueprintjs/labs"
4
5  type Props = { helpText: string }
6
7  const HelpIconComponent = ({helpText}: Props) => {
8    return <Tooltip2 content={helpText}>
9      <Icon
10       iconSize={Icon.SIZE_STANDARD}
11       intent={Intent.PRIMARY}
12       iconName="pt-icon-info-sign"
13     />
14   </Tooltip2>
15 }
16
17 export default HelpIconComponent
```

Ukázka 3.6: Příklad použití knihovny Blueprint.js

3.2.9.2 Flexbox

Pro organizaci elementů v prostoru stránky byla použita knihovna `reflexbox`[36], která obaluje moderní způsob CSS pozicování elementů [12] na stránce.

Díky těmto moderním knihovnám nebylo třeba investovat do vlastní implementace a velmi to urychlilo vývoj aplikace.

Uživatelské rozhraní bylo navrženo jako *single page application*, přičemž uživatel nikdy nepustí jednu stránku. Layout a menu aplikace zůstává vždy stejný a uživatel se přepíná jen mezi jednotlivými sekcemi. Pro směrování a navigaci v aplikaci byla použita knihovna `react-router-dom`[37].

3.2.9.3 Neideální stav aplikace

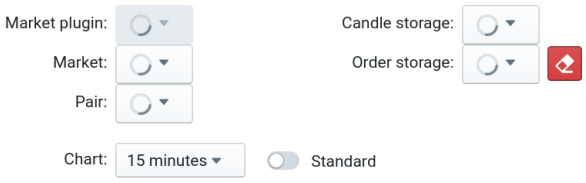
UI aplikace korektně reaguje na tzv. *non-ideal state*, kdy nastane něco neočekávaného. Například když je spojení s backendem přerušeno nebo backend nevrátí žádná data a podobně. Příkladem pak může být čekání na data z backendu (3.8) nebo situace, kdy backend nevrátí pro daný pár žádná data (3.9).

Komponenty pro vizualizaci neideálního stavu obsahuje přímo knihovna Blueprint.js.

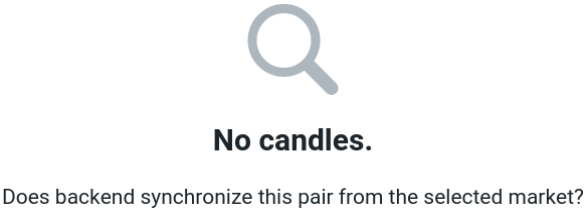
3.2.9.4 Navigace v UI

Pro zlepšení orientace uživatele v aplikaci obsahují všechny klíčové prvky tooltip s vysvětlením. Pro ještě podrobnější vysvětlení pak slouží ikonka s nápovědou (obrázek 3.10).

[36] `https://github.com/jxnblk/reflexbox`
[37] `https://github.com/ReactTraining/react-router`



Obrázek 3.8: Ukázka stavu aplikace když čeká na data z backendu



Obrázek 3.9: Ukázka stavu aplikace když backend nevrátí žádné svíčky

Obrázek 3.10: Ukázka tooltipu s ikonou nápovědy

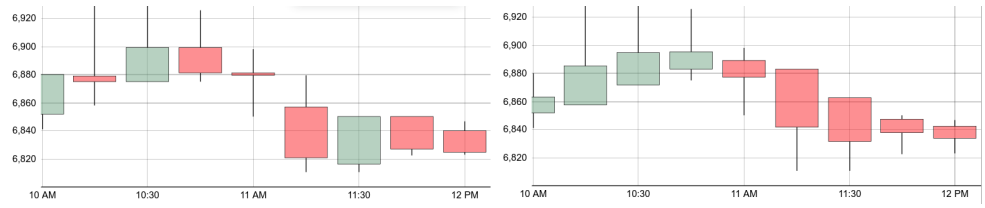
3.2.9.5 Simulační a read-only režim aplikace

Celý frontend je navržen ke dvěma primárním účelům: vizualizovat proběhlé nebo probíhající běhy strategií a umožňovat konfigurovat a spouštět simulační běhy strategií.

Pro odlišení těchto dvou účelů byly do aplikace přidány režimy. Ve výchozím režimu je možné data pouze prohlížet. Zapnutím simulačního režimu se v aplikaci zpřístupní další UI prvky, které umožňují spouštět simulace. Zároveň dojde k tomu, že se uzamkne možnost vybírat si market a předvolí se specifický `MockMarket`, který je třeba pro spouštění simulací.

3.2.10 Kompozice UI aplikace

Všechny obrazovky aplikace sdílí společný toolbar (podrobněji popsaný v 3.2.10.1). Ten obsahuje filtr pro filtrování vizualizovaných dat a elementy pro konfiguraci a spuštění simulací. Celá aplikace je rozdělena na tří obrazovky:



Obrázek 3.11: Rozdíl mezi OHLC (vlevo) a Heikin–Ashi svíčkami (vpravo)

- Dashboard – obsahuje komplexní graf pro vizualizaci burzovních dat a objednávek v grafu.
- Orders – nabízí tabulkový přehled objednávek a vyhodnocení běhů strategií. Zobrazuje performance, výnosnost, průběh zůstatků na burze během běhu strategie.
- Balances – jedná se o přehled aktuálních zůstatků na burze.

3.2.10.1 Komponenta: Toolbar

Toolbar je komponenta, která se zobrazuje na všech obrazovkách. Je konfigurovatelná tak, aby bylo možné některé její elementy skrýt. Tím je možné zobrazit vždy jen ty prvky, které dávají na dané obrazovce smysl. Toho například využívá již zmiňované přepínání režimu pro čtení/simulačního režimu.

Toolbar obsahuje filtry pro všechny dostupné moduly, burzy, strategie a páry. Je tedy možné spustit simulaci s jakoukoliv kombinací zmiňovaných komponent. Obsahuje také filtr pro konkrétní běhy strategie, je tedy možné se zaměřit na jeden zvolený běh.

Dále umožňuje výběr granularity svíček a přepínání jejich typů mezi klasickými OHLC a formátem Heikin–Ashi[38]). Podrobný popis toho, jak se liší svíčky Heikin–Ashi od standardních OHLC svíček je možno nalézt v kapitole 4.7.1 Svíčky Heikin–Ashi.

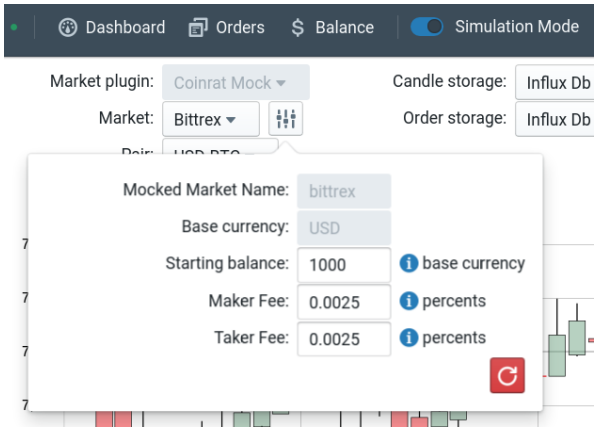
V simulačním režimu obsahuje tlačítka pro konfiguraci strategie a marketu společně s tlačítkem pro spuštění simulace.

3.2.10.2 Obrazovka: Dashboard

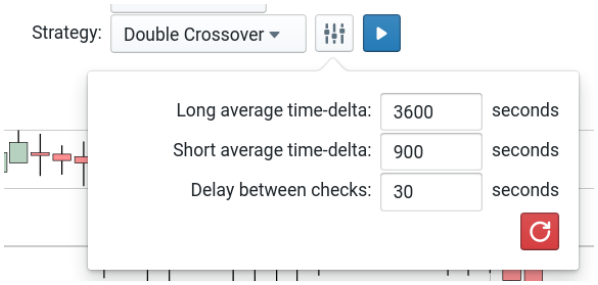
Dashboard je hlavní obrazovkou celé aplikace. Obsahuje graf, kde je možno vidět všechny svíčky v čase a společně s nimi objednávky.

Celý graf je implementován jako komponenta skládající se ze tří subkomponent. První obsahuje graf svíček a další dvě obsahují grafy objednávek nákupů

[38]Heikin–Ashi je odlišný způsob vizualizace svíček, ve kterém následující svíčka nezačíná na uzavírající ceně svíčky předchozí, ale v jejím průměru z *open* a *close* hodnoty.



Obrázek 3.12: Konfigurace simulované burzy



Obrázek 3.13: Rozhraní pro konfiguraci strategie

a prodejů. Nové aktuální svíčky a objednávky se v grafu zobrazují v reálném čase. Ukázka grafu je vyobrazena v příloze B.1.

Z této obrazovky je také možno spouštět simulace strategií.

Na obrázku 3.12 a 3.13 je vidět komponenta `ConfigurationStructureComponent`, která umožňuje konfigurovat simulovanou burzu a vybrané strategie. Struktura konfigurace je poskytována backendem a pro jednotlivé strategie se může lišit. Na obrázku 3.13 je vidět *Double Crossover Moving Average Strategy* implementovanou v rámci této práce, kterou je možno konfigurovat velikostí jednotlivých klouzavých průměrů. Oproti tomu strategie Heikin–Ashi (taktéž implementovaná v této práci) je konfigurovatelná pouze velikostí svíček.

Implementace těchto ukázkových strategií je podrobněji popsána v podkapitolách 4.6 a 4.7.

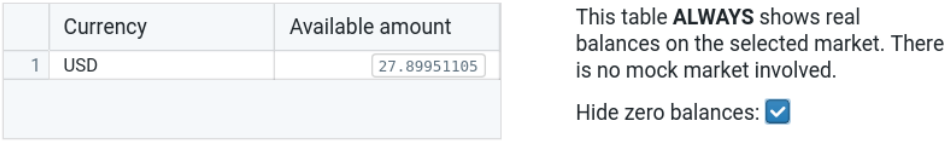
3.2.10.3 Obrazovka: Orders

Přehled objednávek umožňuje prohlížet objednávky v detailní tabulce. Objednávky je možno filtrovat podle všech kriterií obsažených v toolbaru. Nejužitečnější je pak pohled na konkrétní běh strategie, který navíc obsahuje

základní metriky o její úspěšnosti v tomto běhu. Náhled přehledu simulace je v příloze B.2.

3.2.10.4 Obrazovka: Balances

Na této obrazovce je možno sledovat stav portfolia na jednotlivých burzách. Komponenta je vyobrazena na obrázku 3.14. Za povšimnutí stojí možnost skrýt ty měny, které mají nulovou bilanci.



Obrázek 3.14: Komponenta pro zobrazení aktuální bilance portfolia

Kapitola 4

Výchozí zásuvné moduly

V rámci této práce bylo naimplementováno několik výchozích zásuvných modulů, které umožňují ověřit celý koncept platformy v praxi.

4.1 Zásuvné moduly pro ukládání dat

Pro ukládání burzovních dat byla zvolena databáze InfluxDB[39]. Jedná se o specializovanou databázi na tzv. *time-series data*. Jsou to data, která jsou indexovaná časem a jsou podle něj řazena.

Databáze InlfuxDB byla zvolena především proto, že se jedná o software s otevřeným zdrojovým kódem pod licencí MIT[40]. Dalším důvodem bylo její snadné používání díky tomu, že používá jazyk velmi podobný SQL.

Balíček pojmenovaný `coinrat_influx_db_storage` je součástí distribuce a obsahuje: modul pro ukládání objednávek, modul pro ukládání svíček a modul pro ukládání snapshotů portfolia na burze.

Všechny zmíněné moduly pro ukládání dat používají přímo oficiálního klienta pro Python `InfluxDBClient`[41].

V rozsahu této práce nebyla identifikována potřeba pro komplexní ORM[42].

Pluginy tedy sestavují SQL dotazy přímo a zajišťují mapování mezi daty v InfluxDB a objekty doménových entit.

4.1.0.1 Ukládání objednávek

Rozhraní zásuvných modulů pro ukládání objednávek obsahuje možnost mít více úložišť. Toto slouží pro validaci návrhu platformy z toho pohledu, že

[39] `https://www.influxdata.com`

[40] MIT je velmi jednoduchá a nerestriktivní licence. Jedinými podmínkami pro použití softwaru pod licencí MIT je zachování textu licence a uvedení autora.

[41] `https://github.com/influxdata/influxdb-python`

[42] *Object-relational mapping (tool)*, zkráceně ORM je technika (někdy se takto označuje i přímo nástroj) pro konvertování netypových dat v databázi a objekty v prostředí aplikace.

více zásuvných modulů bude poskytovat více úložišť a bude třeba mezi nimi přepínat.

Každé úložiště je identifikováno svým jedinečným jménem. Zodpovědnost za vytváření těchto úložišť je delegována na *dependency injection container*, který tento plugin pro tyto účely obsahuje. Ten zajišťuje sestavení závislostí pro jednotlivé služby a zároveň umožňuje jejich opětovné použití v případě, že přijde požadavek na úložiště se stejným jménem (tzv. drží *pull* služeb).

Tento přístup elegantně rozděluje závislosti mezi třídy a umožňuje třídě reprezentující modul řešit pouze platformou předepsané rozhraní a zodpovědnost za sestavení služeb delegovat.

Využívá se implementace *dependency injection* kontejneru přímo z platformy. Pouze obsahuje jinou definici služeb a je rozšířen o *pull* pro úložiště. Tato schopnost bez překážek znovu použít komponentu ukazuje na její správný návrh.

4.2 Shapeshift

Směnárna Shapeshift provádí transakce tzv. *on-chain*. To znamená, že se transakce zapíše do blockchainů směňovaných kryptoměn a potvrdí decentralizovaným konsensem těžařů. Z toho důvodu je poplatek za obchod na této směnárně řádově vyšší a jeho provedení pomalejší.

Kurzovní rozdíl mezi směnárnou Shapeshift a burzou sečtený s *miner fee*[43] přesahuje dvě procenta. Zatímco burzy typu Bittrex nebo Bitfinex účtují přibližně 0,025% poplatek z transakce.

Objednávka je na burze vyřízena prakticky okamžitě. Zpracování směnárnou Shapeshift trvá jednotky minut.

Směnárna Shapeshift tedy nebyla z těchto důvodů, po konzultaci s vedoucím práce, využita. Místo původně zamýšlené směnárny Shapeshift byla použita burza Bittrex.

4.3 Konektor burzy Bittrex

Balíček `coinrat_bittrex` obsahuje dva zásuvné moduly. Jeden pro komunikaci s burzou Bittrex (vytváření objednávek, ...) a druhý pro stahování svíček.

Komunikace s API burzy Bittrex zajišťuje knihovna `python-bittrex`[44]. Kvalita návrhu API a jejich dokumentace obecně není u burz příliš vysoká. Kromě těchto technických obtíží bylo nutné využít obou verzi API (1.1 i 2.0), jelikož ne všechna data jsou dostupná jen z jedné verze.

[43] Jedná se o poplatek těžařům, kteří zprostředkovávají transakce v síti.

[44] `https://github.com/ericsomdahl/python-bittrex`

4.4 Pluginy pro stahování dat

V rámci této práce byly implementovány dva zásuvné moduly pro stahování dat z burzy. První byl implementován v rámci balíčku `coinrat_bittrex`, který stahuje data přímo z burzy Bittrex. Druhým byl modul napojující agregátor Cryptocompare[45].

4.4.1 Bittrex

Implementace tohoto synchronizačního modulu je velmi přímočará, každou minutu se stáhne nová svíčka a ta se uloží. Zajímavostí je, že Bittrex umožňuje stáhnout naráz všechny minutové svíčky za posledních sedm dní, čehož se využívá a při startu synchronizace se nejprve stáhne celá dostupná historie.

Limitací tohoto modulu je skutečnost, že data z burzy Bittrex jsou (pravděpodobně uměle) zpožděná a svíčky jsou dostupné přibližně s dvouminutovým zpožděním.

4.4.2 Cryptocompare

Cryptocompare je agregátor všech možných informací ohledně prostředí kryptoměn. Obsahuje přehledy a recenze na jednotlivé burzy a měny. Dále volně (pod licencí *Creative Commons – Attribution Non-Commercial license*) zprostředkovává burzovní data. Je tedy možné stahovat data z téměř libovolné burzy.

Cryptocompare poskytuje historická data minutových svíček v rozsahu sedmi dní. V rámci tohoto zásuvného modulu byla implementována pouze synchronizace aktuálního stavu.

Data jsou poskytována s minimálním zpožděním, nepřesahujícím jednu minutu.

4.5 Burza pro simulace

Pro spouštění simulací je nezbytné, aby bylo možné použít burzu, která ve skutečnosti žádné obchody nedělá a je konfigurovatelná pro potřeby testování strategií. Takováto implementace burzy byla realizována v balíčku `coinrat_mock`.

U tohoto burzovního konektoru je možné nakonfigurovat:

- Za jakou burzu se má vydávat (například Bittrext, Bitfinex, ...).
- Jaké množství prostředků bude pro strategii dostupných.

[45] `https://www.cryptocompare.com/`

- Jaká bude velikost poplatků za každý obchod. Tyto poplatky lze nastavit zvlášť pro *maker fee* a zvlášť pro *taker fee*[46].

Každá objednávka zadaná do tohoto falešného burzovního konektoru je okamžitě uspokojena a uzavřena, pokud se eviduje dostatek prostředků pro její provedení. Jedná se o zjednodušení situace, jelikož při obchodování na reálné burze je běžné, že se uzavření objednávky může výrazně zpozdit a nebo ještě hůře, cena se změní tak rychle, že objednávka nebude nikdy uspokojena. Při vytváření strategií je třeba mít toto zjednodušení v simulačním režimu na paměti a implementovat mechanismy, které se umí s takovou situací na reálné burze vypořádat.

Na frontendu se při zapnutí simulačního režimu nastaví tato burza automaticky a není možno ji v tomto režimu změnit.

4.6 Double crossover moving average strategy

První implementovaná strategie je postavena na průniku dvou prostých klouzavých průměrů.

4.6.1 Prostý klouzavý průměr

Prostý klouzavý průměr $m(x)$ je definován v každém bodě x jako prostý aritmetický průměr konstantního počtu (n) předchozích hodnot průměrované veličiny p.

$$m(x) = \frac{1}{n} \sum_{x}^{x-n} p(x) \tag{4.1}$$

4.6.2 Strategie

Jedná se o velmi známou a jednoduchou strategii, kterou je možno snadno konfigurovat pro zvýšení frekvence objednávek (poměrem velikosti n mezi průměry). Výsledná sekvence obchodů je snadno ověřitelná pohledem do grafu. Díky těmto vlastnostem byla tato strategie zvolena pro ověření fungování platformy.

Průměry jsou počítány z hodnoty *price-close*[47] u minutových svíček.

Princip strategie je znázorněn na obrázku 4.1. Průměr s menším n je dynamičtější (na obrázku červeně), průměr s větším n se pak mění s menší dynamikou (na obrázku modře). Zeleně je označena aktuální cena.

[46] Poplatky za transakci jsou na většině burz rozděleny na *maker fee* a *taker fee*. *Maker fee* bývá menší a uplatňuje se, když obchodník umísťuje objednávku na burzu. Naopak *taker fee* se uplatní, když obchodník svým obchodem objednávku jiného obchodníka uspokojí a tím se odebere z burzy.

[47] Cena na konci intervalu svíčky. Více je možno dočíst v kapitole 2.4 Analýza domény.

Obrázek 4.1: Klouzavý průměr

Situace, kdy rychlejší průměr protne pomalejší shora, je signálem k prodeji. Pokud jej protne zespodu, je signálem k nákupu.

Strategie může výrazně prodělávat ve chvíli, kdy cena kolísá tak málo, že poplatky převyšují zisk z obchodů. V této práci byl experimentálně implementován mechanizmus, který se snaží tuto situaci rozpoznat a takovéto obchody nedělat. Avšak neukázal se jako příliš efektivní, jelikož strategie pak vynechávala důležité signály jen proto, že rozdíl od posledního obchodu byl příliš malý.

Může nastat situace, kdy strategie zadá objednávku, ale díky prudkému pohybu ceny na burze se tato objednávka nestihne zpracovat. Aby nedošlo k uzamčení kapitálu v neproveditelné objednávce, strategie vždy s následujícím signálem zruší všechny objednávky v opačném směru. Pokud tedy zůstala nakupující objednávka nevyřízena, při následujícím signálu na prodej se tato objednávka zruší a *vice versa*.

4.6.3 Ověření funkčnosti strategie

Strategie byla ověřena spuštěním na několika vzorcích dat a její výsledek ověřen proti grafu s vynesenými klouzavými průměry. Na obrázku 4.2 je ukázka tohoto kontrolního vynesení objednávek do grafu.

Na grafu je možno si všimnout velmi drobných nepřesností, a to, že je objednávka zadána přibližně o minutu dříve či později. Jedná se pouze o nepřesnost měření, protože nástroj pro vizualizaci dat v InfluxDB (Chronograph) zahrnuje do agregace data lehce časově posunutá oproti tomu, jak je vyhodnocuje strategie.

Hned na začátku implementace byl pro tuto strategii vytvořen snapshotový test. V dalším rozvíjení strategie pak bylo využíváno tohoto testu pro detekci změn chování. Více o snapshotových testech bylo zmíněno v kapitole 3.1.9.

Obrázek 4.2: Ověření strategie Double crossover moving average

4.7 Heikin–Ashi strategy

Druhá implementovaná strategie byla zvolena na základě toho, že pracuje s odlišným typem svíček a je konfigurovatelná jejich velikostí. Vhodně tedy doplňuje předchozí implementovanou strategii. Zároveň je i tato strategie poměrně jednoduchá a díky tomu i snadno ověřitelná.

4.7.1 Svíčky Heikin–Ashi

Jelikož celá platforma pracuje s OHLC svíčkami, do strategie byla naimplementována příslušná konverze. Definice Heikin–Ashi svíček je následující:

$$HeikinClose_n = \frac{open_n + high_n + low_n + close_n}{4} \tag{4.2}$$

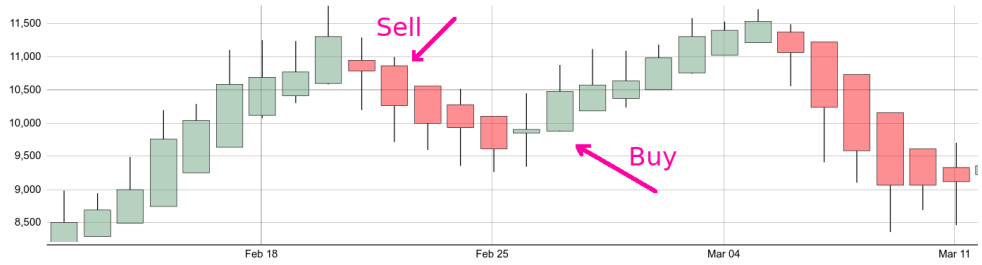
$$HeikinOpen_n = \frac{open_{n-1} + open_{n-1}}{2} \tag{4.3}$$

$$HeikinHigh_n = max(high_n, low_n, HeikinOpen_n, HeikinClose_n) \tag{4.4}$$

$$HeikinLow_n = min(high_n, low_n, HeikinOpen_n, HeikinClose_n) \tag{4.5}$$

V rovnici 4.3 je patrné, že jedna svíčka začíná v průměru předchozí. Z toho plyne problém sestavení první svíčky, který je řešen tak, že je pro účely této strategie první svíčka definována takto:

$$HeikinClose_n = \frac{open_n + high_n + low_n + close_n}{4} \tag{4.6}$$



Obrázek 4.3: Strategie Heiking–Ashi

$$HeikinOpen_n = \frac{open_n + open_n}{2} \tag{4.7}$$

$$HeikinHigh_n = high_n \tag{4.8}$$

$$HeikinLow_n = low_n \tag{4.9}$$

Tato první svíčka se sestavuje vždy relativně k začátku intervalu, čímž se zavádí drobná chyba. Ta se ale velmi rychle opraví v několika málo následujících svíčkách.

4.7.2 Strategie

Samotná strategie je založena na předpokladu, že trh je trendový. Jelikož svíčky Heikin–Ashi, díky své závislosti na předchozí svíčce mění směr jen při změně trendu, jsou velmi vhodným nástrojem.

Algoritmus pro vyhodnocení signálu k nákupu nebo prodeji je pak následující, když se objeví dvě po sobě jdoucí svíčky v jednom směru po sérii svíček ve směru opačném, předpokládá se, že je trend vyčerpán a dochází k jeho zvratu.

Když tedy po sérii býčích[48] svíček přijdou dvě medvědí, situace se vyhodnotí jako signál k prodeji. V opačné situaci pak jako signál k nákupu. Situace je vyobrazena na obrázku 4.3. Velikost svíček je zde jediným konfiguračním parametrem.

Trend je detekován naivní metodou sčítání trendových svíček. Za každou býčí svíčku se přičte k trendovému koeficientu jednička a za každou medvědí svíčku se jednička odečte. Toto se provádí až do chvíle, než se narazí na mez. Tím je omezena paměť trendu jen na lokální oblast.

[48]Trh, který má vzestupný trend, se označuje jako býčí (*bullish*). Trh při sestupném trendu se pak oproti tomu nazývá medvědí (*bearish*). Stejně tak se pak označují i svíčky.

4.7.3 Ověření funkčnosti strategie

Ověření funkčnosti strategie bylo provedeno spuštěním na různých datsetech a správnost ověřena přímo v grafu aplikace. Příklad takového grafu je zobrazen v příloze B.3.

Závěr

Na základě analýzy domény a obchodních strategií byla úspěšně implementována platforma pro obchodování na kryptoměnových burzách. Implementací dvou strategií a konektoru na burzu Bittrex bylo ověřeno, že platformu lze snadno a elegantně rozšiřovat o zásuvné moduly. Díky dobře navrženému rozhraní je jejich implementace přímočará a odpadá nutnost řešit cokoliv mimo doménu zásuvného modulu. Pomocí webového nástroje vytvořeného v rámci této práce je možné strategie vizualizovat. K dispozici je komplexní graf a obsáhlá tabulka s vývojem bilance. Oproti zadání je možno strategie navíc simulovat na historických datech.

Cíl práce i všechny body zadání práce byly splněny. Funkčnost výsledné aplikace byla experimentálně ověřena.

Tato práce mně byla velkým přínosem, neboť jsem si rozšířil znalosti o fungování burzovních trhů a získal jsem praktické zkušenosti s jazykem Python a JavaScript v kontextu středně velkého projektu.

Práce byla poměrně náročná kvůli identifikaci vhodné úrovně abstrakce pro vytvoření funkčního rozhraní mezi platformou a zásuvnými moduly. Během implementace bylo třeba některé koncepty přepracovat. Díky tomu jsem získal vhled do návrhu komplexnějších systémů.

Potenciál k dalšímu rozvoji je obrovský. Prvním nutným krokem je implementace zásuvného modulu s knihovnou `cctx`[49]. Tato knihovna poskytuje rozhraní pro komunikaci s více než sto burzami. Plugin používající tuto knihovnu by tak efektivně poskytoval konektor do všech těchto burz.

Dalším důležitým krokem je integrace knihovny `ta-lib`[50]. Knihovna obsahuje nástroje pro technickou analýzu burzovních dat. Jednotlivé strategie by pak nemusely řešit implementaci indikátorů samy, ale měly by je k dispozici přímo skrze rozhraní jako součást platformy.

Z první validace trhem jistě vyplyne potřeba rozšířit rozhraní pro strategie o další možnosti. Jednou z nich by mohlo být rozšiřování domény o další typy

[49] `https://github.com/ccxt/ccxt`

[50] `https://github.com/mrjbq7/ta-lib`

dat. A to jak burzovních (například celkový obrat trhu), tak i externích (data ze sociálních sítí a podobně).

Jedním z dalších směrů, kterým by se vývoj aplikace mohl ubírat, je zpracování dat v reálném čase. Tedy že by strategie mohla reagovat synchronně na příchod burzovní svíčky (a nebo změny stavu burzovního stocku objednávek, pokud jsou k dispozici) a ne jen v intervalu každých n sekund.

Celý projekt je volně k dispozici na Githubu[51] pod licencí MIT. Je tedy možné jej svobodně rozšiřovat a měnit. Projekt byl zároveň publikován v české komunitě obchodníků s kryptoměnami[52] s velmi pozitivní reakcí.

Následujícím krokem je pokusit se projekt validovat trhem a v případě úspěchu vybudovat kolem projektu stabilní komunitu a bohatý ekosystém rozšíření.

[51] `https://github.com/Achse/coinrat` a `https://github.com/Achse/coinrat_ui`

[52] `https://www.facebook.com/groups/BitcoinoviSpekulanti/permalink/1622279041181148/` (vyžaduje členství)

Literatura

[1] Grigorik, I.: *High Performance Browser Networking*. O'Reilly Media, Inc., 2013, ISBN 1449344747, 9781449344740.

[2] Socket.io: *Socket.IO — Server API*. [cit. 2018-04-15]. Dostupné z: `https://socket.io/docs/server-api/`

[3] The Python Software Foundation: *Glossary — Python 3.6.4 documentation*. [cit. 2018-03-13]. Dostupné z: `https://docs.python.org/3/glossary.html`

[4] IETF Trust: *RFC 6749 - The OAuth 2.0 Authorization Framework*. [cit. 2018-04-16]. Dostupné z: `https://tools.ietf.org/html/rfc6749`

[5] Wiggins, A.: The twelve-factor app. 2017. Dostupné z: `https://12factor.net/config`

[6] The Python Software Foundation: *Support for type hints — Python 3.6.4 documentation*. [cit. 2018-03-15]. Dostupné z: `https://docs.python.org/3/library/typing.html`

[7] Python.org: *PEP 282 – A Logging System*. [cit. 2018-04-16]. Dostupné z: `https://www.python.org/dev/peps/pep-0282/`

[8] InfluxData: *Understanding the time intervals returned from GROUP BY time() queries*. [cit. 2018-04-17]. Dostupné z: `https://docs.influxdata.com/influxdb/v0.9/troubleshooting/frequently_encountered_issues`

[9] React.org: *Reconciliation - React*. [cit. 2018-04-19]. Dostupné z: `https://reactjs.org/docs/reconciliation.html`

[10] Chromium.org: *V8 doesn't stable sort*. [cit. 2018-04-17]. Dostupné z: `https://bugs.chromium.org/p/v8/issues/detail?id=90`

[11] Mozilla.org: *Array.sort isn't a stable sort (switch to MergeSort)*. [cit. 2018-04-17]. Dostupné z: `https://bugzilla.mozilla.org/show_bug.cgi?id=224128`

[12] W3Schools: *CSS Flexbox (Flexible Box)*. [cit. 2018-04-16]. Dostupné z: `https://www.w3schools.com/css/css3_flexbox.asp`

Příloha **A**

Seznam použitých zkratek

AMQP Advanced Message Queuing Protocol

API Application programming interface

BTC Bitcoin

CSS Cascading Style Sheets

DOM Document Object Model

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

JSX JavaScript XML

MIT Massachusetts Institute of Technology

OHLC Open High Low Close

ORM Object-relational mapping

REST Representational state transfer

SRP Single responsibility principle

SaaS Software as a Service

SSH Secure Shell

SSL Secure Sockets Layer

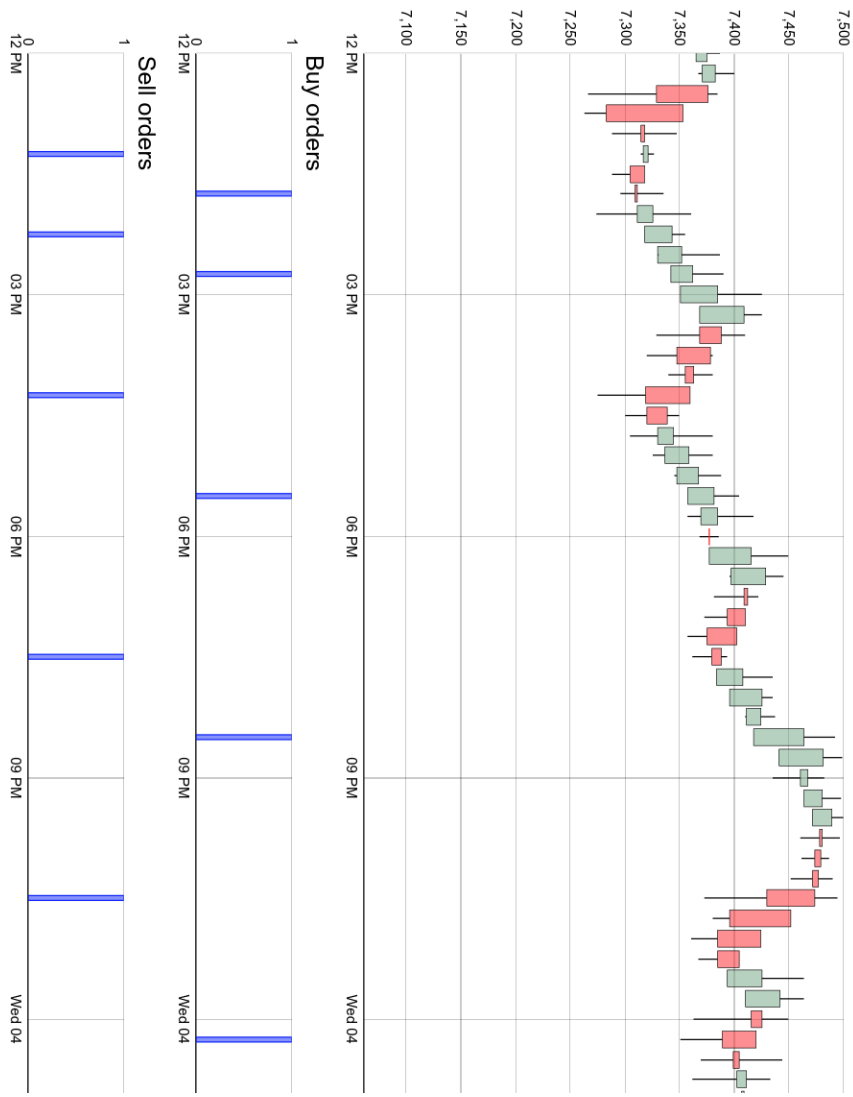
SQL Structured Query Language

UI User interface

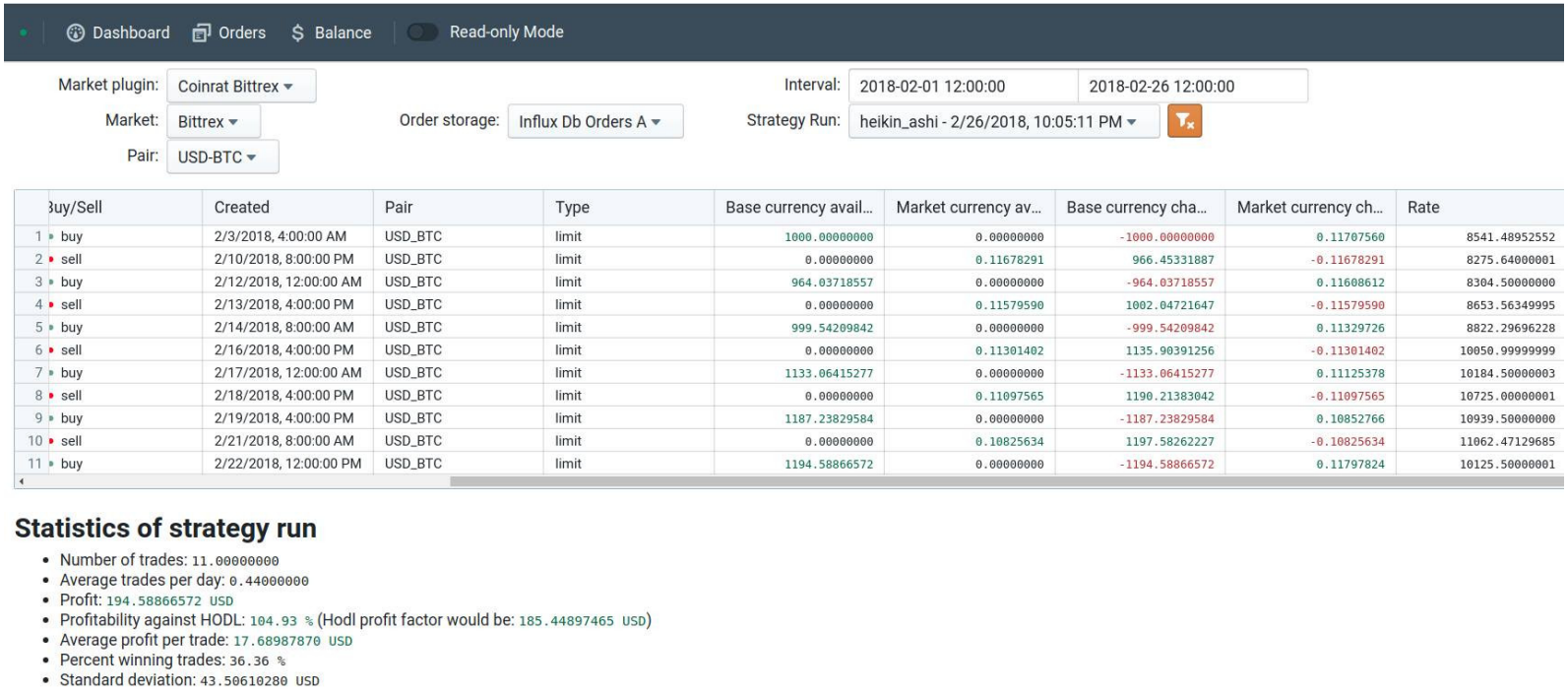
UTC Coordinated Universal Time

Příloha B

Náhledy aplikace

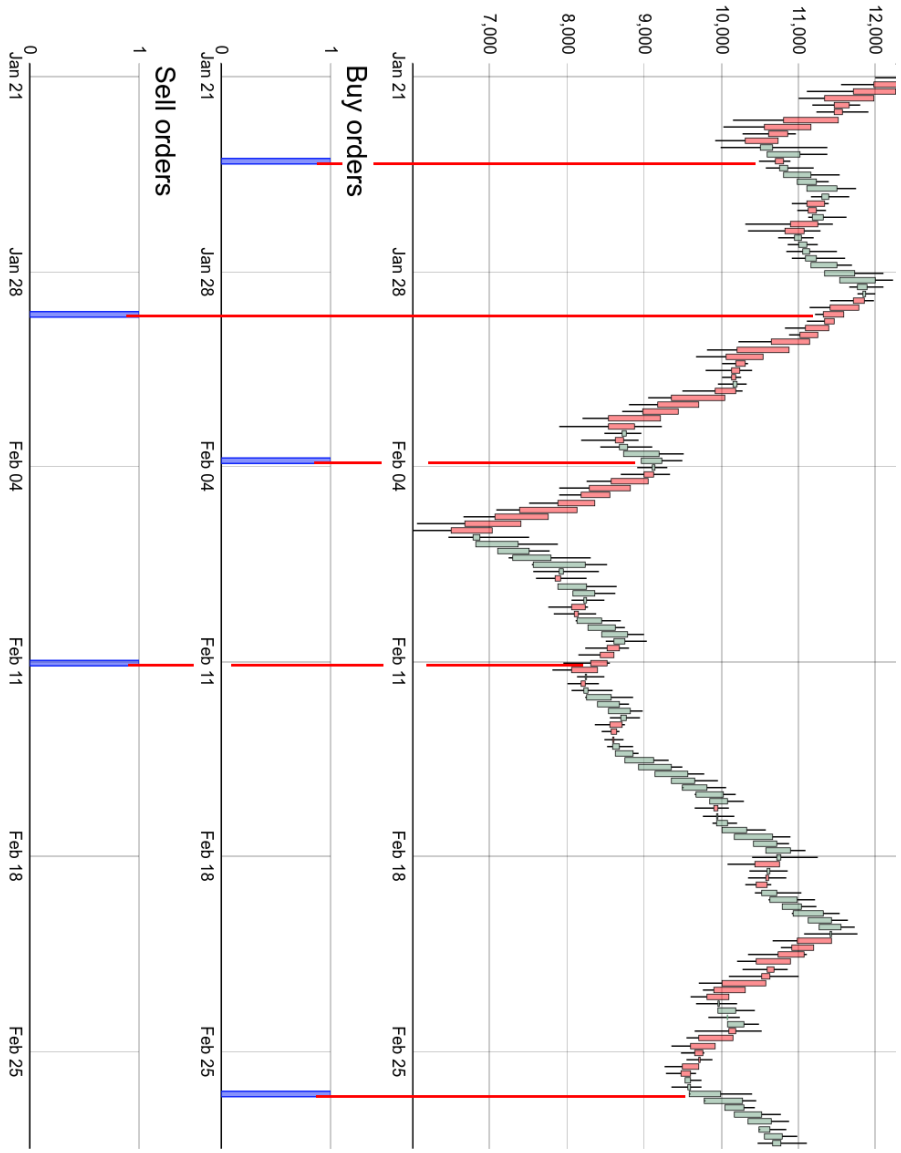


Obrázek B.1: Náhled grafu pro vizualizace svíček a objednávek



	Buy/Sell	Created	Pair	Type	Base currency avail...	Market currency av...	Base currency cha...	Market currency ch...	Rate
1	buy	2/3/2018, 4:00:00 AM	USD_BTC	limit	1000.00000000	0.00000000	-1000.00000000	0.11707560	8541.48952552
2	sell	2/10/2018, 8:00:00 PM	USD_BTC	limit	0.00000000	0.11678291	966.45331887	-0.11678291	8275.64000001
3	buy	2/12/2018, 12:00:00 AM	USD_BTC	limit	964.03718557	0.00000000	-964.03718557	0.11608612	8304.50000000
4	sell	2/13/2018, 4:00:00 PM	USD_BTC	limit	0.00000000	0.11579590	1002.04721647	-0.11579590	8653.56349995
5	buy	2/14/2018, 8:00:00 AM	USD_BTC	limit	999.54209842	0.00000000	-999.54209842	0.11329726	8822.29696228
6	sell	2/16/2018, 4:00:00 PM	USD_BTC	limit	0.00000000	0.11301402	1135.90391256	-0.11301402	10050.99999999
7	buy	2/17/2018, 12:00:00 AM	USD_BTC	limit	1133.06415277	0.00000000	-1133.06415277	0.11125378	10184.50000003
8	sell	2/18/2018, 4:00:00 PM	USD_BTC	limit	0.00000000	0.11097565	1190.21383042	-0.11097565	10725.00000001
9	buy	2/19/2018, 4:00:00 PM	USD_BTC	limit	1187.23829584	0.00000000	-1187.23829584	0.10852766	10939.50000000
10	sell	2/21/2018, 8:00:00 AM	USD_BTC	limit	0.00000000	0.10825634	1197.58262227	-0.10825634	11062.47129685
11	buy	2/22/2018, 12:00:00 PM	USD_BTC	limit	1194.58866572	0.00000000	-1194.58866572	0.11797824	10125.50000001

Obrázek B.2: Přehled objednávek v simulačním běhu strategie



Obrázek B.3: Ověření běhu strategie Heikin–Ashi

Příloha C

Obsah přiloženého USB flash disku

- `readme.txt` ..... stručný popis obsahu
- `src`
  - `coinrat` ..... implementace backendu
  - `coinrat_thesis` ..... zdrojové kódy diplomové práce
  - `coinrat_ui` ..... implementce frontendu
    - `DP_Hejna_Petr_2018.pdf` ..... text diplomové práce