Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Webová aplikace na hledání arbitráží na burzách

Jiří Martinů

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jiří Martinů

Systémové inženýrství a informatika
Informatika

Název práce

Webová aplikace na hledání arbitráží na burzách

Název anglicky

Web application for searching for arbitrages on stock exchanges

---

Cíle práce

Tato bakalářská práce je zaměřena na problematiku vývoje webových aplikací. Jejím hlavním cílem je navrhnout a implementovat aplikaci, která získává real-time data z burz a hledá mezi nimi arbitráže. Dílčím cílem je popsat postupy implementace a využité přístupy a technologie.

Metodika

Práce sestává z teoretické a praktické části.

Metodika zpracování praktické části je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska pro zpracování praktické části.

Praktická část se zaměřuje na návrh a implementaci aplikace podle poznatků z teoretické části. Aplikace bude sloužit k vyhledávání arbitráží v real-time burzovních datech. Při návrhu a vývoji aplikace bude využito standardních metod a postupů softwarového inženýrství. Implementace bude provedena v prostředí ASP.NET Core. Výsledná aplikace bude nasazena, otestována a budou shrnuty poznatky z jejího vývoje a testování. Na základě těchto poznatků bude navržena možnost dalšího případného rozšíření aplikace v budoucnu.

Doporučený rozsah práce

35-40 stran

Klíčová slova

.NET Core, SignalR, Arbitráže, React, MVC, C#

Doporučené zdroje informací

Adam Freeman. Pro ASP.NET Core MVC 2. Berkeley, CA: APress, 2017. ISBN 978-1-4842-3149-4.

ASP.NET Documentation [online]. ©2018 [cit. 19.6.2019]. Dostupné z: https://docs.microsoft.com/en-gb/aspnet/?view=aspnetcore-2.2#pivot=core

Cryptocurrency arbitrage strategies [online]. Medium.com, ©2018 [cit. 19.6.2019]. Dostupné z: https://medium.com/coinmonks/cryptocurrency-arbitrage-strategies-part-i-20e9dd327919

Rami Vemula. Real-Time Web Application Development. Berkeley, CA: APress, 2017. ISBN 978-1-4842-3269-9.

React – A JavaScript library for building user interfaces [online]. React.org, ©2019 [cit. 19.6.2019]. Dostupné z: https://reactjs.org/

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 10. 03. 2021

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Webová aplikace na hledání arbitráží na burzách" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 10.03.2021 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Jiřímu Brožkovi, Ph.D. za vedení práce, konzultace a poskytnuté rady.

Webová aplikace na hledání arbitráží na burzách

Abstrakt

Tato bakalářská práce je zaměřena na problematiku návrhu a vývoje webové aplikace, která vyhledá arbitráže na kryptoměnových burzách. Finální aplikace by měla sloužit jako pomocný nástroj při obchodování.

Teoretická část byla rozdělena na dva oddíly. První z těchto oddílů bude obsahovat informace týkající se kryptoměn, burz, definici řešeného problému či ujasnění základních pojmů. Druhý oddíl popíše využité technologie, zdůvodní jejich použití a porovná je s alternativami.

V praktické části bude důkladně rozebrán životní cyklus aplikace počínaje návrhem, implementací přes testování a refaktorizaci až po nasazení na server.

Veškeré poznatky, jenž budou získány, ať už během vývoje, testování či při konečném nasazení, budou analyzovány a využity jako podklad pro diskuzi.

Klíčová slova: .NET Core, C#, MVC, SignalR, React, Redux, TypeScript, Docker, Ansible, GitLab, CI, Kryptoměny, Arbitráže

Web application for searching for arbitrages on stock exchanges

Abstract

This bachelor thesis is focused on the design and development of a web application that seeks arbitrage on cryptocurrency exchanges. The final application should serve as an auxiliary tool in trading.

The theoretical part was divided into two sections. The first of these sections will contain information on cryptocurrencies, stock exchanges, the definition of the problem to be solved or the clarification of basic concepts. The second section describes the technologies used, justifies their use and compares them with alternatives.

In the practical part, the life cycle of the application will be thoroughly analyzed, starting with the design, implementation through testing and refactoring to deployment to the server.

All knowledge gained, whether during development, testing or final deployment, will be analyzed and used as a basis for discussion.

Keywords: .NET Core, C#, MVC, SignalR, React, Redux, TypeScript, Docker, Ansible, GitLab, CI, Cryptocurrencies, Arbitrages

Obsah

Seznam obrázků

Seznam kódů

Seznam wireframů

Seznam použitých zkratek

API – rozhraní pro programování aplikací

PoW – Proof of work

PoS – Proof of stake

CPU – Centrální procesorová jednotka

GPU – Grafický procesor

FPGA – Programovatelné hradlové pole

ASIC – Integrovaný obvod specializovaný na jeden druh operací

UWP – Universal Windows Platform

CIL – Common Intermediate Language

CoreCLR – Common Language Runtime

JS – JavaScript

DI – Dependency Injection

1 Úvod

Prakticky každý stát má vlastní měnu, kterou jeho občané využívají na placení svých potřeb. Taková měna má mnoho výhod jako kupříkladu stabilita hodnoty nebo její všeobecná akceptace. Emitování a kontrola měny je zpravidla v rukou nějakého orgánu státu, což umožňuje státu provádět úpravy, které nemusí být ve prospěch všech vlastníků dané měny. A proto mimo jiné vznikly kryptoměny.

Kryptoměny se snaží tento problém eliminovat pomocí decentralizace, tudíž neexistuje žádná autorita, která by mohla svévolně měnu měnit. I přes tento markantní rozdíl, se kryptoměnami dá pracovat stejně jako s fiat měnami za podmínky, že jí akceptují obě strany směny.

Stejně jako existuje forex pro státní měny, existují burzy pro kryptoměny. Burzy kryptoměn poskytují obchodníkům možnost vytvořit nabídku, respektive poptávku na směnu měny na jinou za stanovenou cenu. Tyto a další operace se dají provádět přes webové stránky dané burzy, ale v některých případech je poskytnuto i WebSocket API. Rychlost komunikace přes WebSocket API se dá považovat za "Near Real-Time" a to dovoluje uživatelům získaná data zpracovat, analyzovat a na základě výsledků provést vhodné akce. Jedním z výsledků analýzy může být nalezení intermarketových arbitráží.

Intermarketová arbitráž je situace, kdy obchodník na jedné burze prodá vybrané aktivum dráž, než ho ve stejný moment nakoupí na druhé burze. Celý tento proces musí proběhnout v řádu několika sekund.

Aplikace, jenž bude výsledkem této práce, právě tyto arbitráže za pomocí moderních technologiích hledá. Cílem bylo navrhnout a naimplementovat program tak, aby vyžadoval minimum uživatelské obsluhy, ale zároveň poskytoval dostatečné množství informací s co nejmenším zpožděním.

2 Cíl práce a metodika

2.1 Cíl práce

Tato bakalářská práce je zaměřena na problematiku vývoje webových aplikací. Jejím hlavním cílem je navrhnout a implementovat aplikaci, která získává real-time data z burz a hledá mezi nimi arbitráže. Dílčím cílem je popsat postupy implementace a využité přístupy a technologie.

2.2 Metodika

Práce sestává z teoretické a praktické části.

Metodika zpracování praktické části je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska pro zpracování praktické části.

Praktická část se zaměřuje na návrh a implementaci aplikace podle poznatků z teoretické části. Aplikace bude sloužit k vyhledávání arbitráží v real-time burzovních datech. Při návrhu a vývoji aplikace bude využito standardních metod a postupů softwarového inženýrství. Implementace bude provedena v prostředí ASP.NET Core. Výsledná aplikace bude nasazena, otestována a budou shrnuty poznatky z jejího vývoje a testování. Na základě těchto poznatků bude navržena možnost dalšího případného rozšíření aplikace v budoucnu.

3 Teoretická východiska

3.1 Obchodní východiska

3.1.1 Kryptoměny

Kryptoměny jsou speciální druh digitální měn, které ke svému fungování využívání metody z kryptografie. Transakce mezi uživateli se zaznamenávají v blockchainu. Blockchain je distribuovaná decentralizovaná databáze obsahující všechny transakce za celou historii měny, a tudíž se znalostí blockchainu je možné určit kdo a jaké množství dané měny vlastní. Tyto transakce se v blockchainu slučují do bloků, jenž musí před tím některý z minerů potvrdit.

Miner je osoba, která za pomoci výpočetní techniky potvrzuje bloky. Potvrzení bloku probíhat tak, že se vezme hash předchozího bloku, z toho se vypočítá hlavička bloku. Přidáním tohoto hashe se zamezí možnosti zaměnění pořadí bloků v blockchainu. Následuje zahashování všech transakcí v bloku a na konec se přidá magické číslo. Celé se to opět zahashuje, a pokud výsledný hash začíná dostatečným počtem nul určeným současnou obtížností, tak se blok bere jako potvrzený. Magické číslo je voleno náhodně minerem bez možnosti toto číslo určit jinak než hádáním, a proto je nutné celý proces mnohokrát opakovat. To zajišťuje, že bloky jsou potvrzovány v určitém časovém odstupu a je tedy patrné, kdo daný blok potvrdil první a patří mu odměna za potvrzení bloku. [1, s. 45]

Tento mechanismus se nazývá Proof of work (PoW), a jedná se o jeden z nejvíce energeticky a technologicky náročných způsobů generování měny.

Měny typu PoW se dále dělí podle druhu hashovacího algoritmu (SHA-256, Ethash, Scrypt, X11) či dle výpočetní techniky, která je k jejich těžbě potřeba (CPU, GPU, FPGA či ASIC).

U některých kryptoměn se může způsob fungování lišit. Například u kryptoměny typu Proof of stake (PoS) je potvrzovatel bloku zvolen podle množství a staří měny na svém účtu. Tento typ těžby měny je značně úsporný oproti PoW a nevyžaduje žádný speciální hardware. Mezi další typy patří Proof of authority, Proof of burn či Proof of space.

Absence jakékoliv autority u kryptoměn není až tak úplně pravdou. V případě kryptoměn jsou touto autoritou vývojáři měny, jenž mají kompletní kontrolu nad jejím fungováním a mohou jí svévolně měnit. Uživatelé měny se mohou, ale tyto změny rozhodnout vetovat pomocí neaktualizace svého softwaru či v extrémním případech vlastním

forkem. Fork měny je akce, během které se jedna měna rozdělí na dvě měny, a přitom si obě zachovají historii transakcí (blockchain), ale začnou pracovat na jiných principech.

3.1.2 Burzy

Burzy kryptoměn jsou povětšinou internetové stránky, které zprostředkovávají směny kryptoměn mezi uživateli. Burzy žádné měny vlastní měny neobchodují a jsou tedy jen prostředníkem. Zájemce o směnu měny na jinou měnu vytvoří v systému poptávku, ve které stanoví cenu a množství. Pokud se v systému, již nachází nabídka, která může tuto poptávku alespoň částečně uspokojit, tak proběhne směna. Tento proces se opakuje, dokud není poptávce plně vyhověno.

Před vytvořením poptávky je nutné převést nabízenou měnu do peněženky přidělené burzou. Měna přijatá ze směny bývá převedena také do peněženky přidělené burzou, ale v některých případech je umožněn převod přímo do peněženky klienta a tím ušetřit jednu transakci. [2] Burzy vydělávají tím, že si berou část směňovaných měn jako poplatek za zprostředkování směny.

3.1.3 Arbitráže

Arbitráže jsou situace na trhu, ve které obchodník může profitovat na nevyváženosti cen. Dělí do dvou základních kategorií, a těmi jsou trojúhelníkové a intermarketové arbitráže.

O trojúhelníkovou arbitráž se jedná pokud, když obchodník nakoupí za aktivum A aktivum B, za to pořídí aktivum C, a nakonec za aktivum C nakoupí zpět aktivum A, tak je v zisku. Tento typ arbitráže je možné hledat na jedné, ale i na více burzách současně. Už samotné nalezení představuje velmi náročnou výpočetní operaci a vyžaduje provedení patřičných transakcí během několika sekund. A proto bývá hledání zpravidla omezeno na 3 až 4 burzy současně.

Druhou z kategorií jsou intermarketové arbitráže (dále jen arbitráže), jimiž se bude zabývat tato práce. Jedná se o zjednodušenou verzi trojúhelníkových arbitráží, kde probíhá hledání pouze na dvou burzách současně. Obchodník musí v tomto případě realizovat pouze dvě transakce. Nakoupit zvolené aktivum na burze A a současně prodat to samé aktivum na burze B, a tím realizovat zisk. [3]

3.2 Softwarové východiska

3.2.1 C#

C# je všestranně použitelný objektově orientovaný vyšší programovací jazyk inspirovaný jazyky C++ a Java. Při jeho vývoji byl kladen důraz na jednoduchost a přenositelnost se zachováním všech moderních funkcionality. Mezi tyto funkcionality patří silná typová kontrola, kontrola hranic pole či garbage collection. Garbage collection/collector je forma samočinné správy paměti. [4]

Vlastnosti jazyka C#:

- Lambda výrazy – anonymní funkce, které nejsou navázané na identifikátor.
- Delegáti – slouží k popisu funkcí.
- Linq – umožňuje vytvářet dotazy nad poli, xml dokumenty, databázemi či dalšími datovými zdroji.
- Generika – funkce a třídy mohou mít specifikovány některé z parametrů, respektive atributů později až při volání, respektive inicializaci.
- Partial (částečné) třídy – dovoluje rozdělení jedné třídy do více souborů. [5, s. 94]
- Generátory – metody, operátory či gettery mohou vracet iterátory.
- Metody rozšíření – metody přidané již zkompilovaného objektu.
- Asynchronní metody – metody, které se zpracovávají asynchronně a zároveň jsou neblokující

Kompilátor a runtime (prostředí určené pro běh zkompilovaného C# kódu) pro jazyk C# implementuje hned několik projektů. Od Microsoftu to jsou .NET Framework, .NET Core a od open-source komunity je to Mono.

3.2.2 .NET Core

.NET Core je multiplatformní softwarový framework primárně vyvíjený společností Microsoft. Distribuovaný je, ale jako open source pod licencí MIT, takže se na vývoji může podílet kdokoliv. Předchůdcem toho frameworku byl .NET Framework, který pracoval pouze na Windows, a to dalo za vznik Mono a DotGNU. V současnosti Microsoft stále ještě

pracuje na vývoji .NET, ale následující verze .NET 5 sloučí .NET, .NET Core a Mono v jeden projekt.

Aplikace v .NET Core mohou být psány v programovacích jazycích C# či F# a s jistými omezeními i v C++/CLI a Visual Basic .NET. [6, s. 23] Aktuální verze (2020) .NET Core 3.1 umožňuje tvorbu rozličných typů aplikaci: ASP.NET webové aplikace, aplikace určené pro spuštění v příkazové řádce, Windows Forms a UWP.

Před spuštěním aplikace se musí zdrojový kód se nejdříve zkompilovat .NET Core kompilátorem do CIL. CIL je set instrukcí, které splňují definici dle Common Language Infrastructure (CLI). Ten udává, jaký má mít formát kód a runtime vysokoúrovňového multiplatformního jazyka. Po kompilaci vznikne assembly, jenž se předá do CoreCLR. CoreCLR je virtuální stroj starající se o spouštění CIL kódů. CIL kód se v CLR zkompiluje na nativní kód. Následuje provedení kódu na CPU. [7, s. 16]

3.2.3 ASP.NET Core

Webový open-source framework primárně pracující na .NET Core, ale některé starší verze podporují i .NET Framework. Stejně jako je .NET Core nástupcem .NET, tak ASP.NET Core byl stvořen redesignem ASP.NETu. Výsledný framework se značně zjednodušil, zrychlil, nabízí větší modularitu a přenositelnost, takže již není zapotřebí, pro spuštění aplikací na unixových systémech, Mono či DotGNU. [8]

Součásti či související:

- Kestrel – vysokorychlostní web server určený pro hostování ASP.NET aplikací. Podporuje HTTPS, HTTP/2, WebSockety a další. Ve většině situací bývá nasazen společně s reverzní proxy jako Apache, Nginx či routerem jako Traefik, protože není schopný sdílet IP a port s ostatními procesy. [9]
- MVC – Model-View-Controller je typ softwarové architektury, která dělí aplikaci na tři nezávislé části, a tudíž případné úpravy v jedné části neovlivňují ostatní části nebo jen minimálně. Pokud uživatel pošle HTTP požadavek na aplikaci, která je postavená na MVC vzoru, tak je tento požadavek nejprve přesměrován na odpovídající controller. Controller má za úkol pracovat s modelem, volbu správného view a vrácení response (odpovědi) klientovi. Model reprezentuje business logiku či stav aplikace. Mezi operace na modelu se řadí například dotazy na databázi či zpracování

dat službou. Po získání všech potřebných dat jsou tyto data předána do view (pohled) enginu, jenž z nich vygeneruje HTML. [10, s. 7]

- Razor – "markup syntax" pro vkládání serverového kódu do webových stránek. Syntaxe se skládá z Razor značek, C# a HTML.
    1. Razor view engine – Stará se o transformaci Razor šablon do HTML. Obsahuje vlastní syntax, aby bylo možné přistupovat k předaným proměnným a dalším prostředkům přidáním prefixu "@". Jedním z možných rozšíření jsou Tag Helpery. Tag helper dovoluje tvorbu vlastních atributů tagů, které při renderování view provedou definovanou akci.
    2. Razor Pages – Tato část .NET Core frameworku umožňuje tvorbu aplikací bez nutnosti využívat controllery a model, tak jako v MVC. Část logiky se v tomto případě přesouvá do samotných šablon a do speciálních Page tříd.
- Blazor – Open source framework určený k tvorbě webových aplikací. Dostupný je v pěti různých variantách. Každá z těchto variant je rozšířením některé již existující součásti .NET Core ekosystému a jejich cílem je nabídnou nové metody vývoje webových stránek. Nejvýznamnější z variant je client-side framework napsaný v WebAssembly s názvem Blazor WebAssembly. Aplikace napsané v něm pak na frontendu namísto JavaScriptu využívají C# a většina výpočtu se provádí na straně uživatele.
- NuGet – Nuget je open source správce Nuget balíčků. Jeho hlavní funkcí je získávání patřičných balíčků a jejich závislostí. Zároveň umožňuje tvorbu balíčku a jejich další správu.
- Entity Framework (EF) Core – Open source ORM (object-relational mapping) framework postavený na ADO.NET. Slouží k zjednodušení práce s perzistentními daty. Jeho základem jsou Entity, jenž reprezentují objekty reálného světa. Mezi těmito Entitami mohou být vazby několika typů: "one-to-one", "one-to-many" nebo "many-to-many". Entity Framework dokáže tento systém Entity a jejich vazeb uložit do databáze a zase při potřebě zpětně vytvořit z databáze. [11]

3.2.4 WebSocket

WebSocket je komunikační protokol pracující na TCP. Vytváří obousměrný komunikační kanál mezi klientem a serverem. Oproti HTTP je rychlejší a spotřebovává méně dat. Byl navržen tak, aby mohl pracovat na HTTP portech, a tudíž byl kompatibilní s

HTTP. Na portu 80, respektive 443 využívá nezašifrované, respektive zašifrované připojení. Těchto podobností využívá při počátečním navazování spojení (handshake), kdy do HTTP požadavku přidá speciální header nazvaný "Upgrade header", který HTTP serveru říká, že daný klient má zájem o změnu protokolu na WebSocket. [12, s. 9]

3.2.5 SignalR

Open-source knihovna od Microsoftu pro ASP.NET Core sloužící k asynchronnímu posílání dat ve směru od serveru k připojeným klientům. K přepravě dat může využívat WebSockety, HTTP Post, Server-Sent Eventy či Long Polling. SignalR automaticky vybere nejvhodnější způsob podle dostupnosti, rychlosti a či požadovaném směru komunikace. [13]

Oproti samotným WebSocketům nabízí jednoduší API, a to jak na straně klienta, tak i na serverové straně.

3.2.6 React

React (ReactJS) je open-source, frontendová knihovna napsaná v JavaScriptu (JS) určená na tvorbu uživatelských rozhraní. První verze vznikla v roce 2013 jako projekt Facebooku, ale v současně době je distribuován pod MIT licencí. Primárně je používán při tvorbě SPA (Single Page Application), ale je možné v něm za pomocí React Native tvořit aplikace pro mobilní zařízení. Samotný React je vhodný pouze na tvorbu menších aplikací. K tvorbě větších aplikací je nutné přidat alespoň state manager a router. [14]

Součásti či související:

- Components – Jsou základním stavebním prvkem v Reactu. Mohou reprezentovat HTML tagy či další komponenty. Každá komponenta má "props" a "state". Data předané při deklarování komponenty jsou "props" a většinou se za život komponenty nemění. Data, která se mění, jsou uložena ve "state". Komponenty se dají zadefinovat funkcionálně či třídou. [15]
- JSX (JavaScript XML) – Rozšíření JS, které umožňuje vykreslování HTML s předanými daty z JS.
- Virtuální DOM – React si udržuje v paměti kopii DOM (Document Object Model) a všechny úpravy provádí nejprve na něm. Díky tomu může provést ty samé úpravy na skutečném DOM velmi efektivně.

- Lifecycle methods – Jsou metody, které nám dovolují ovlivňovat stav či chování komponenty v určité fázi životního cyklu. Nejdůležitější metoda se nazývá render. Tato metoda se volá po aktualizaci komponenty. Mezi další metody patří componentDidMount, respektive componentWillUnmount, které se spouštějí na začátku, respektive na konci existence komponenty.
- React hooks – Hooky dělají stejnou práci jako Lifecycle methods, ale jsou lépe čitelné a jednodušší na pochopení. Fungují pouze v komponentách, které byly zadefinovány funkcionálně. [16]

3.2.7 Redux

Webové aplikace potřebují ke svému fungování si pamatovat interakce s uživatelem a ostatní vykonané akce. Tyto zapamatované informace se nazývají "state" aplikace. Redux je state manager pro aplikace napsané v JS, nejčastěji v Reactu či Angularu. Inspirací při jeho tvorbě byl Flux pattern a „fold funkce" z funkcionální programování.

Základem Reduxu je store, což je JS objekt, který v sobě udržuje state aplikace. Jediný způsob, jak je možné měnit informace uložené v něm, je dispatchnutí action. Action je jednoduchý objekt, který popisuje, co se právě stalo. Dispatch takové action je proces, při kterém se dá na vědomí všem reducerům, že byla tato akce vykonána. Reducer je kus kódu, který definuje, jak se state aplikace změní při vyvolání nějaké action. Obvykle aplikace obsahuje několik reducerů a každý z nich dokáže zpracovat jen některé action, a proto se u action vyžaduje přítomnost neprázdné property "type", podle které je možné rozlišovat actiony. [17]

Tento systém sice vyžaduje napsání více kódu a u menší projektů zvyšuje složitost, ale má hned několik výhod:

- Time travel – Pokud developer využívá Redux DevTools, tak se všechny action společně se snapshoty státu ukládají a je tedy mezi nimi možné přecházet, a to i do minulosti. Toto slouží primárně koncovým uživatelům, aby se mohli pohybovat zpět v historii procházení, ale také k reprodukování bugů. [18]
- Neduplicitní data – Data jsou uložena v centrálním úložišti (store), takže k nim může přistupovat každá komponenta, která je připojená k Redux storu, a tudíž není nutné mít nějakou informaci uloženou na více místech naráz.
- Minimální překreslení – Není nutné překreslovat všechny komponenty, ale jen ty, které se doopravdy změnily.

3.2.8 TypeScript

Open-source programovací jazyk vyvinutý firmou Microsoft jako nástavba nad JS. Obohacuje JS o vlastnosti používané v C#, Javě či C++. Mezi přidané vlastnosti patří například možnost statického typování, třídy, moduly, interface, async/await či generika. Kód napsaný v TypeScriptu se musí nejprve před použitím transpilovat pomocí transpilátoru do JS. [19]

TypeScript je hojně využívaný na frontendu, ale s rozmachem Node.js se začal používat i na backendu. I kód, který je napsaný v pouze čistém JS, se považuje za plně validní TypeScript. TypeScript umožňuje popsání již existujícího JS kódu pomocí hlavičkového souboru a následně s ním pracovat jako s kódem napsaný v TypeScriptu. [20]

3.2.9 Docker

Docker je open-source software, který poskytuje virtualizaci na úrovni operačního systému. K tomu využívá kernel hostujícího stroje a jeho nativní izolační a virtualizační nástroje (namespaces a cgroups). Prostředí, ve kterém běží takto virtualizovaná aplikace, se nazývá container. Díky využití kernelu, containery obsahují pouze programy potřebné pro svůj chod, a to znamená, že nezabírají tolik místa na disku. Nevýhodou tohoto systému je závislost na hostujícím kernelu. [21, s. 11]

Docker se dá rozdělit na tři části:

- Docker daemon – Zkráceně dockerd, je daemon (proces běžící na pozadí), který spravuje containery. Ovládán je přes Docker Engine API. Uživatelům ke komunikaci s Docker Engine API slouží CLI program „docker“. [22]
- Objekty:
    1. Images – Stejně jako třídy v OOP jazycích, slouží Docker image jako předpis či šablona pro tvorbu jeho instance, v tomto případě conteineru. Nový image může být založený na již existujícím “base” image a přidávat do něj úpravy dle potřeb. Každá taková úprava vytvoří vlastní vrstvu v image. Ačkoliv se jedné o nový image, tak jeho velikost bude minimální, protože bude obsahovat pouze do něj přidané vrstvy a referenci na base image. Před tím, než je možné podle image vytvořit container, je nutné provést build image, což je operace, při které se provedou všechny operace uvedené v Dockerfile.

Dockerfile je soubor, který obsahuje jednotlivé operace potřebné k tvorbě image. Pokud jsou provedeny změny v Dockerfile, tak při opětovné buildu image se provedou jen operace, které byly změny.

2. Containers – Jedná se o instance daného Docker image. Uživatelé mohou ovládat conteinery pomocí CLI či Rest API. Mezi typické operace nad containerem patří zapnutí, vypnutí, vytvoření či smazání. Změny provedené v containeru se ukládají do doby, než je smazán. Defaultně je nově vytvořený container izolovaný od hostujícího systému a ostatních containerů. Container je možné připojit do Docker sítě, která dle svého nastavení, zprostředkovává komunikaci s ostatními containery ve stejné síti, s hostem či službami na internetu. Pokud container vyžaduje, aby některé data, které v něm vzniknou, přežily i jeho odstranění, může se při jeho tvorbě k němu přiřadit volume. Volume je složka na hostu či docker volume, do kterého může container ukládat data. [23]
3. Services – Pro zvýšení dostupnosti a load balancingu, services vytvářejí repliky containerů na různých hostech, na kterých běží Docker daemon. Aby tomu bylo možné, musí být tito daemoni spojeni do Docker Swarmu. Daemoni mezi sebou komunikují pomocí Docker API před Docker network, který mezi nimi vznikne při inicializaci Swarmu. [24, s. 7]

- Registries – Uložiště pro Docker image. Uživatelé si mohou vytvořit vlastní soukromé či využívat veřejné. Z těchto úložišť se pak dají stahovat image a nahrávat image za pomocí CLI. Mezi nejznámější veřejné registry patří Docker Hub a Docker Cloud.

3.2.10 Ansible

Ansible je víceúčelový open-source nástroj určený ke konfiguraci počítačů, nasazování aplikací či obdobným operacím. Při využití pro konfiguraci počítačů se využívá faktu, že Ansible dokáže z jednoho počítače (kontrolní node) konfiguraci aplikovat současně na více počítačů (node) na jednou. Pokud se Ansible používá pro nasazení aplikace na server, tak je to pravděpodobně proto, aby se nemusely provádět repetitivní operace manuálně a zároveň se ověřil stav již dříve provedených operací.

Ke svému fungování stačí Ansible pouze SSH a Python na kontrolním nodu. Na spravovaných nodech není potřeba instalovat Python, protože Ansible pracuje jako “agentless”.

Seznam nodů, na kterých se mají zvolené operace provést, se ukládá v Inventory souboru. Jedná se o soubor ve formátu INI, jež obsahu IP adresy či domain name nodů. Umožňuje tyto nody shlukovat do skupin a podle toho určovat, jaké operace se na nich provedou. Operace prováděné na nodech se vykonávají pomocí takzvaných modulů. Modul zpravidla zaobaluje nějaký skript, tak aby byl schopný pracovat s Ansiblem a splňoval dané standardy. Hlavním z těchto standardů je, že modul musí být idempotentní, a tudíž bude jeho několikanásobné spuštění mít vždy stejný výsledek. [25]

Pořadí spuštění modulů či atributy použité pro spuštění se zapisují do playbooků. Ansible playbook je yaml soubor, který, kromě dříve zmíněných dat, může obsahovat volání Ansible rolí, definování konstant či filtrování provedení modulů podle rolí z Inventory souboru. Role v Ansible je způsob sdružování playbooků do ucelených bloků, jenž je potom snadné přenášet a spouštět na jiných strojích. [26, s. 147]

3.2.11 Git

Distribuovaný systém určený ke správě verzí zdrojového kódu během jeho vývoje. Jeho zakladatele je Linus Torvald, jenž potřeboval náhradu za BitKeeper při vývoji Linuxového jádra. Byl navržen tak, aby usnadnil koordinaci více programátorů pracujících na stejném projektu.

Stěžejním prvkem Gitu jsou větve. Pokud programátor potřebuje pracovat na některé části projektu bez toho, aby ovlivňoval kód ostatním uživatelům, vytvoří si novou větev. Nová větev bude obsahovat historii zdrojové větve, ale změny přidané do ní neovlivní zdrojovou větev. Programátoři přidávají změny do větví postupně v menších dávkách zvaných “commit”. To dovoluje, v případě problémů, projít zpětně historii změn a vrátit kód do bodu, kdy ještě fungoval. Změny se zpětně do zdrojových větví dostávají pomocí “merge” akce. Merge je operace, při které se porovnají dvě větve, zjistí se rozdíly mezi nimi a ty se následně propíší do zvolené větve. [27, s. 34]

Jak už bylo výše naznačeno, Git byl od počátku tvořen, aby spravoval verze linuxového jádra, což je velmi velký projekt, na kterém se provádí simultánně stovky operací, takže Git musel být od počátku rychlý. Toho se dosáhlo chytrým návrhem, ale taky optimalizací. [28]

3.2.12 Gitlab

Webová aplikace sloužící pro správu a rozšíření funkcionalit Git repozitáře. Oproti interakci s Git repozitářem přes shell či jiné API, nabízí větší přehlednost a lepší vizualizaci prakticky všech prvků.

Přes GitLab je možné tvořit nové větve, zobrazovat soubory v určitém bodě historie jejich změn či zobrazení grafu větví. Jednou z hlavní výhod je tvorba merge požadavků, jenž jsou nástavbou nad merge větví. K merge požadavku mohou uživatelé přidávat komentáře nebo označovat problémové kusy kódu. Novější verze GitLabu obsahují WebIDE, přes které je možné upravovat kód rovnou v prohlížeči a vytvořit tak commit. Nápady na nové vlastnosti a bugy se trackují v issues, které je možné provázat s merge požadavky, aby se daly později označit jako hotové. GitLab se velmi zaměřuje na bezpečnost, a proto nabízí několik úrovní práv pro uživatele. Dokumentace projektů se provádí pomocí zabudované wiki. GitLab obsahuje vlastní Continuous Integration řešení, zkráceně CI. [29, s. 327]

Vydávají se dvě varianty GitLabu, EE (Enterprise Edition) a CE (Community Edition) s tím, že CE je zdarma a EE je sice placená, ale za to nabízí lepší nástroje a podporu. Backend je napsán primárně v Ruby, ale novější části jsou napsané v Go. Frontend je postaven na Vue.js. [30]

3.2.13 CI/CD

Zkratka používaná v softwarovém inženýrstvím, kde CI znamená “continuous integration” a CD stojí za “continuous delivery/deployment”. Společně tyto dva procesy, jenž každý z nich v sobě skrývá určité množství kroků, překlenují mezeru mezi vývojem aplikace, testováním a jejím finálním nasazením na produkci.

Continuous integration vyžaduje, aby vývojáři do společného úložiště aplikace přidávali postupně menší části kódu nejlépe do vlastní větve. Když pak dojde k merge této větve do některé z nadřazených větví, aplikace je ihned s novým kódem zkompilována a následně na ní proběhne testování. Tento postup nemusí být standardem u všech vývojářů, ale bývá velmi častý. [31]

Termíny continuous delivery a continuous deployment není možné plně vzájemně zaměňovat, ale oba popisují kroky, které se provedou s aplikací po opuštění z CI části. Hlavním rozdílem mezi delivery a deployment je, že při deploymentu je aplikace automaticky nasazena na produkci, zatím co při delivery je potřeba manuální nasazení. Tyto

kroky mohou obsahovat další úrovně testování, jak kódu samotného, tak i funkcionalit na byznys úrovni. Aplikace bývá v těchto krocích nasazena na několik testovacích prostředí, aby se dalo plně otestovat chování s nově přidaným kódem. Finálním krokem je nasazení aplikace na produkční server, její následné monitorování, jenž má za výstup požadavky na změny jako například hotfixy či vylepšení. [32]

3.2.14 Let's Encrypt

Certifikační autorita (CA), jenž vydává bez úplaty certifikáty určené k bezpečné komunikaci po internetu. Nejčastější využití těchto certifikátu je na webových aplikacích pro podporu HTTPS, ale je možné je využít například i pro Radius server pro PEAP. Oproti ostatním CA neověřuje identitu žadatele o certifikát, ale pouze to, zda je žadatel vlastníkem či správcem domény.

Toto ověření může být provedeno splnění jedné z podporovaných výzev (challenge). Nejjednodušší je, byť zastaralá, httpChallenge, při které má žadatel za úkol nahrát na server soubor, tak aby byl dostupný pro Let's Encrypt na dané doméně. [33] Jako náhrada za tuto challenge, je doporučována tlsChallenge, kdy žadatel nahraje na server dočasný certifikát.

Let's Encrypt vybízí k použití Certbot aplikace, která tento proces plně zautomatizuje, a i sama vytvoří cron na obnovu certifikátu před jeho vypršením.

4 Vlastní práce

4.1 Analýza požadavků a proveditelnosti

Cílem aplikace od počátku vždy bylo, poskytnou autorovi a ostatním uživatelům nástroj, který by umožnil efektivně vyhledávat obchodní příležitosti a předpovídat vývoj ceny kryptoměn na kryptoměnových burzách. Výběr vhodných funkcionalit, které bude aplikace nabízet, byl omezen na ty, jenž měly předpoklad toho, že je bude možné nastudovat a naimplementovat v řádu několika měsíců a zároveň nebudou neúměrné náročné na výpočetní výkon. Původní návrh aplikace obsahoval dvě primární funkcionality, a to hledání trojúhelníkových a intermarketových arbitráží.

Autor měl již naprogramovanou aplikaci na hledání trojúhelníkových arbitráží z předchozího studia v jazyku C++, a proto byla proveditelnost této funkcionality otestována v minimální implementaci jako první. Na testovacích datech byla aplikace schopná nalézt hledané trojúhelníkové arbitráže, ale po napojení na zdroje reálných dat (tří kryptoměnových burz) aplikace nalezla za 30 hodin provozu jen velmi omezený počet trojúhelníkových arbitráží, které navíc nebyly nepříliš výnosné. Během tohoto testu aplikace využívala průměrně 80 % osmivláknového procesoru. Algoritmus byl následně upraven, aby ignoroval poplatky za směny, ale ani takové nastavení příliš nezlepšilo výsledky. Autor se domnívá, že jsou tyto arbitráže vyobchodovány samotnými kryptoměnovými burzami, jelikož mají k poptávkám a nabídkám na směny jako první přístup a nemusí počítat s poplatky. Z těchto dvou důvodů nebyla tato funkcionalita implementována do produkční verze.

Po odstranění trojúhelníkových arbitráží z původního návrh se intermarketové arbitráže staly hlavní funkcionalitou, takže bylo provedeno ověření proveditelnosti za pomocí zjednodušené implementace v JS. Mezi hlavní požadované kritéria patřila rychlost nalezení arbitráže a menší využívání procesoru než předchozí testovaná funkcionalita, tak aby bylo možné aplikaci na jednom počítači zároveň vyvíjet i spouštět.

Ačkoliv autor programuje již několik let v PHP, při volbě použitých technologií se rozhodl využít pro backend programovací jazyk C# s frameworkem .NET Core, jelikož PHP není vhodné pro vícevláknové výpočty a s velkými obtížemi pracuje s WebSockety. Bylo zvoleno, že samotné WebSockety budou využité pouze k získávání dat a k předávání dat na frontend bude probíhat přes SignalR, protože nabízí robustnější řešení a je triviální jej implementovat v .NET Core.

Protože se aplikací zobrazovaná data mění několikrát během sekundy, nebylo možné postavit frontend, pro .NET Core na nativním, Razor view engine. Byl tedy zvolen ReactJS, jelikož zvládá překreslovat velké množství komponent za krátký čas. Kvůli nutnosti spravovat na frontend velké množství dat, byl do aplikace zakomponován Redux, jenž ukládá veškerá data na jedno místo. Navíc se Reduxem omezil počet překreslovaných ReactJS komponent, poněvadž Redux předává data do komponent pouze v případě, že se provázená data změnily. U kódu frontendu se předpokládalo, že bude obsahovat velké množství zanoření, a proto byl frontend napsán v TypeScriptu, který zaručuje čitelnost a přehlednost kódu.

4.2 Analýza existujících řešení

4.2.1 Intra-Exchange-Crypto-Arbitrage (IECA)

Menší jednoúčelová aplikace určená k hledání trojúhelníkových arbitráží na kryptoměnových burzách napsaná v programovacím jazyce R. Stejně jako tato práce, má IECA aplikace webové rozhraní. Autor IECA aplikace v původní verzi nabízel vyhledávání na 6 burzách, ale v poslední aktualizaci podporuje pouze dvě, jelikož API burz se často mění a vyžadují tak konstantní úpravy v aplikaci. IECA získává data poptávek a nabídek z REST API burz, což je sice jednodušší na komunikaci než WebSockety používané touto prací, ale je výrazně pomalejší. Hlavní nevýhodou IECA aplikace je, že si sama neumí získat obchodované kryptoměnové páry na burzách, ale je nutné, aby je kódu aplikace programátor přidal. [34]

4.2.2 OneExBit

Jedná se o víceúčelový nástroj pro obchodování a analýzu kryptoměnových burz dostupný jako aplikace pro Windows či Mac OS. Obsahuje širokou škálu základních funkcionalit například real-time výpisy zvolených párů, technické indikátory, přehledové dashboardy, vytváření nabídek a poptávek na burzách či historii uskutečněných obchodů. Hlavním rozdílem od ostatních aplikací je to, že zahrnuje algoritmy na vyhledávání intramarketových (trojúhelníkových) a intermarketových arbitráží, a zároveň roboty, jež jsou schopni, podle uživatelem nastavených pravidel, nalezené arbitráže sami vyobchodovat. Uživatelé si mohou vybrat z 8 burz, ale v budoucích verzích mají vývojáři v plánu přidat desítky dalších. Aplikace je v současnosti (rok 2021) stále v beta verzi, a proto

je zdarma se všemi funkcionalitami a nástroji. Nevýhodou této aplikace je její stagnující vývoj, který je opožděn přibližně o rok od původního plánu. [35]

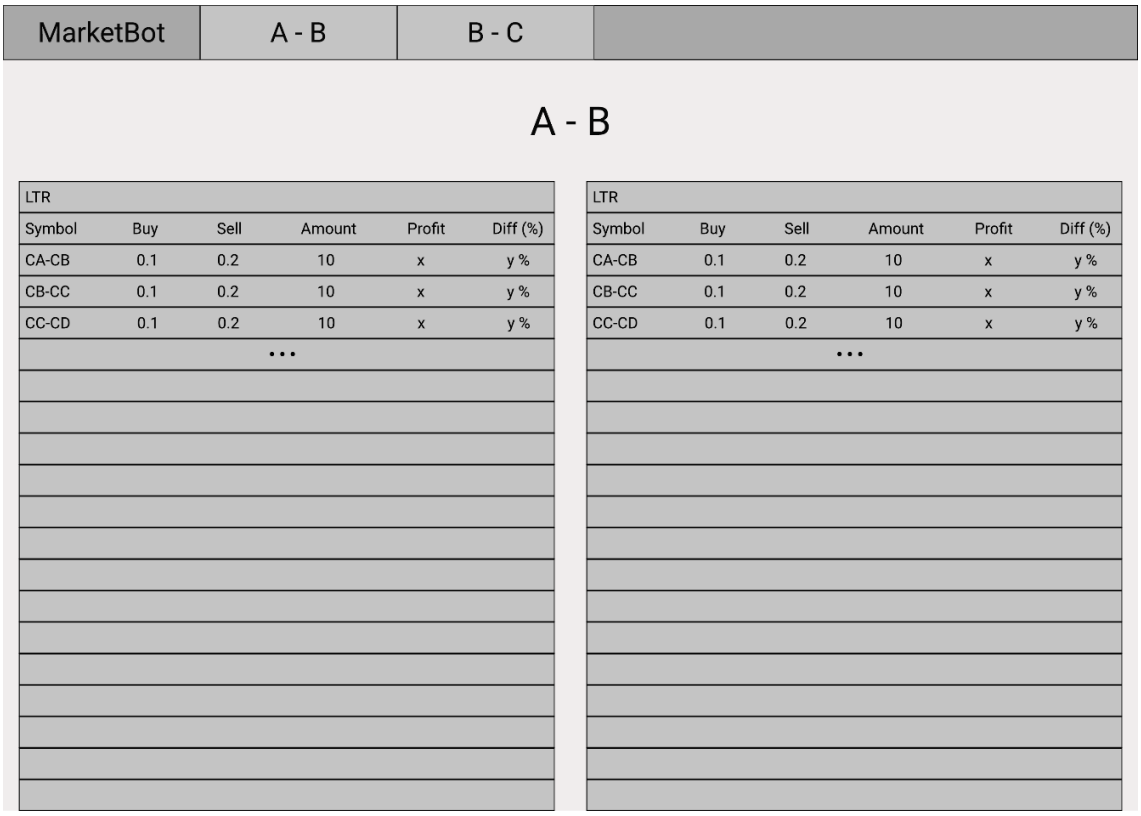
4.3 Návrh uživatelského rozhraní

Návrh uživatelského rozhraní aplikace cílil na to, aby aplikace byla přehledná, jednoduchá na používání a nenacházely se na ní prvky, jež by zbytečně odváděly pozornost uživatelů. V horní části obrazovky se nachází menu s tlačítky. Každé tlačítko obsahuje název dvou burz oddělený pomlčkou (obdélníky s textem “A-B” a B-C). Klikem na vybrané tlačítko je možné přejít na vybraný výpis porovnáních těchto dvou burz. Pod menu uprostřed se nachází nadpis obsahující text označující zobrazovaný pár burz.

Zbytek celé stránky zabírají dva graficky stejné obdélníkové bloky. Oba bloky jsou vyplněný řádky. První řádek obsahuje “LTR” (Left To Right) a “RTL” (Right to Left), což označuje směr směny, aby uživateli bylo jasné, kde provést případný nákup a prodej. Pro zjednodušení bude autor označovat burzu, na které se nakupuje, respektive prodává jako “zdrojová”, respektive “cílová”. Druhý řádek je hlavička popisující jednotlivé buňky řádku. Buňky v prvním sloupečku s textem “Symbol” obsahují směnovaný měnový pár. Sloupeček “Buy” zobrazuje nejnižší cenu, za kolik je možné na zdrojové burze nakoupit. Hodnoty ve sloupci “Sell” naopak ukazují nevyšší prodejní cenu na cílové burze. Maximální množství měny je možné nakoupit, aby se stále směna stále vyplatila, se nachází ve sloupci ”Amount”. Poslední dva sloupce zobrazují výhodnost směny, a to v jednotkách ve sloupci ”Profit” a v procentech ve sloupci ”Diff (%)”.

Aplikace bude vyžadovat zobrazování dlouhých čísel kvůli přesnosti, a proto se nebude řádně zobrazovat na menších displejích.

Wireframe 1: Wireframe uživatelského rozhraní



Zdroj: Autor

4.4 Implementace

V této části se autor zaměřuje na popsání fungování backendu a frontendu. Většina kapitol pojednává o konkrétní třídě a k ní navazující problematice, ale některé se zaměřují na principy či problémy přesahující jednu třídu. Backend a frontend jsou každá samostatná aplikace, takže slovo “aplikace” v následujících odstavcích vždy referuje na aplikaci podle dané kapitoly.

4.4.1 Backend

4.4.1.1 Princip

Ústředními částmi backendu jsou dvě třídy dědící z BackgroundService, jejíž architektura je inspirována problémem producenta a spotřebitele. Instance třídy MarketDataProvider v běžící aplikaci získává data z burz a ukládá je k objektům reprezentující burzy. Druhou zmiňovanou třídou je MarketComparator. Tato třída vezme

získaná data ze všech burz a vytvoří mezi nimi páry. Ty následně porovná na a vypočítá nejlepší možnou směnu. V poslední fázi předá tyto data pomocí SignalR na frontend.

4.4.1.2 Třída Market

Abstraktní třída Market slouží jako rodičovská třída, ze které musí dědit všechny třídy reprezentující burzy. (Kód 1) Je napsána tak, aby byla co nejobecnější, tak aby její potomci nebyli omezování ve svém fungování, ale zároveň se zabránilo opakování kódu. Obsahuje obecné informace o burzách jako jméno a výši poplatků, ale také atributy PairsSource třídy SourceInformation a DataSource třídy ApiSourceInformation. Tyto třídy se využívají na popsání postupu získávání měnových párů a směnných kurzů z burz. V poslední řadě tato třída obsahuje slovník, do kterého pak MarketDataProvider zapisuje získané měnové páry a MarketComparator je odsud načítá pro další zpracování. Klíčem v tomto slovníku je string obsahující název měnového páru a slovník vrací pro existující klíče instance třídy CurrencyPair.

Kód 1: Atributy třídy Market

```
public abstract class Market
{
    public enum FeeType {Maker, Taker};

    public string Name { get; set; }

    public SourceInformation PairsSource { get; set; }

    public ApiSourceInformation DataSource { get; set; }

    public Dictionary<string, CurrencyPair> Pairs { get; set; }

    public Dictionary<FeeType, decimal> Fees { get; set; }
```

Zdroj: Autor

4.4.1.3 Třída SourceInformation

Tato třída je základem pro všechny třídy, které popisují práci se zdroji měnových párů a požadavků na směnu z burz. Nachází se v ní pouze jeden atribut s názvem Method. Ten je typu ProcessPairs, což je delegát (Kód 2), který definuje funkci, která zpracovává data měnových párů z burz. Třídy, vycházející z této třídy, přistupují k rozdílným zdrojů, tak tato metoda musí být robustní.

Kód 2: Třída SourceInformation

```
public delegate void ProcessPairs(JToken pairs);

abstract public class SourceInformation
{
    public ProcessPairs Method { get; set; }
}
```

Zdroj: Autor

4.4.1.4 Třída WebSocketApi a ostatní potomci třídy SourceInformation

WebSocketApi je největší třída v aplikaci dědící ze SourceInformation. Její instance slouží k definování, jak se budou získaná data z burz zpracovány. Tato třída konkrétně pracuje s WebSockety, a tudíž vyžaduje mnoho atributů k vytvoření spojení. K napojení na většinu WebSocket API burz stačí provedení subscribe operace na WebSocket, ale některé vyžadují i přihlášení do zdrojového kanálu. Zároveň se API burz značně liší, a proto tato třída též hojně využívá delegatů.

Delegáti DoSubscribe a SubscribeMessage se volají na začátku připojovacího procesu, kde řeší vytvoření připojení, respektive vygenerování připojovací zprávy. K odstranění nepotřebných zpráv jsou aplikováni delegáti SkipMessage a IsSubscribedMessage. Poslední delegát v této třídě "DoSubscribe" má za úkol WebSocketu oznámit, na kterém kanálu aplikace chce naslouchat. (Kód 3)

Ostatní potomci rodičovské WebSocketAPI jsou třídy RestApi, která získává data z REST API burz a StaticSourceInformation, jenž získává data ze souboru s daty ve formátu CSV či JSON.

Kód 3: Atributy třídy WebSocketApi

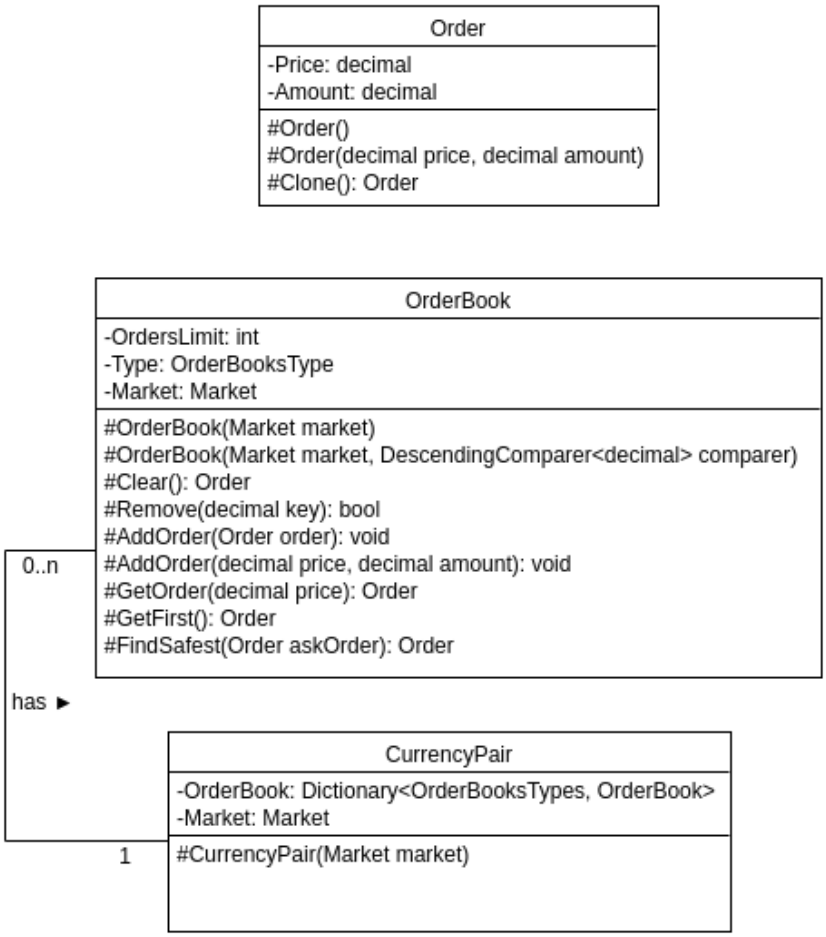
```
public bool NeedsToSubscribed { get; set; } = false;
public bool CallRepeatedly { get; set; } = true;

public SubscribeMessage SubscribeMessage { get; set; }
public SkipMessage SkipMessage { get; set; }
public IsSubscribedMessage IsSubscribedMessage { get; set; }
public AddChannel AddChannel { get; set; }
public DoSubscribe DoSubscribe { get; set; }
public Dictionary<int, string> Channels { get; set; }
```

Zdroj: Autor

4.4.1.5 Třídy Order, OrderBook a Currency

Tyto tři třídy slouží k ukládání dat nabídek a poptávek na burzách. (Obrázek 1) Ve třídě Order se zaznamenává Price (cena) a Amount (množství) jedné poptávky či nabídky. Mezi OrderBook a Order není vazba, protože OrderBook je potomek třídy SortedList, a tudíž v sobě ukládá pouze číselné hodnoty atributů Order. OrderBook implementuje několik metod, které v případě potřeby vygenerují instanci Order z dat uložených v SortedList. Další metody přidávají do třídy základní operace jako odstranění položky či kompletní vymazání SortedList. Pokud bude OrderBook pracovat jako úložiště nabídek, tak se musí jeho řazení změnit na sestupné oproti nativnímu vzestupnému. Toho se docílí při volání konstruktoru daného OrderBook tím, že se mu předá jako druhý parametr DescendingComparer.

Obrázek 1: Diagram tříd z namespace MarketBot.Models.Markets.Data



Zdroj: Autor

4.4.1.6 Třída HitBTC a ostatní potomci třídy Market

Jedinou metodou ve třídě HitBTC je konstruktor. V tom se vytvářejí instance potomků třídy SourceInformation a definují delegáti pro patřičné zdroje dat. Ačkoliv třída WebSocketApi je potomkem třídy Market a dědí tedy z ní atributy PairsSource a DataSource, musí být tyto atributy zde znovu nadefinovány se slovíčkem "new". Protože aplikace ve své implementaci ve velké míře využívá delegáty, tak se pro PairSource a DataSource může využít stejná třída, i když oba dva zdroje zpracovávají data různým způsobem. (Kód 4)

Kód ostatních tříd odvozených od třídy Market vypadá velmi podobně, ale liší v použitých zdrojích dat a v jejich definicích.

Kód 4: Část kódu třídy HitBTC

```
public class HitBTC : Market
{
    public new WebSocketApi PairsSource { get; set; }

    public new WebSocketApi DataSource { get; set; }

    public HitBTC() : base()
    {
        Name = "HitBTC";

        PairsSource = new WebSocketApi();
        PairsSource.Uri = "wss://api.hitbtc.com/api/2/ws";
        PairsSource.CallRepeatedly = false;
        PairsSource.NeedsToSubscribed = true;
        PairsSource.DoSubscribe =
        async (ClientWebSocket clientWebSocket) => {
            byte[] buffer = new byte[524288];

            buffer = Encoding.ASCII.GetBytes(
                PairsSource.SubscribeMessage("")
            );

            await clientWebSocket.SendAsync(
                new ArraySegment<byte>(buffer, 0, buffer.Length),
                WebSocketMessageType.Text, true,
                CancellationToken.None
            );
        };
        PairsSource.SubscribeMessage = (string pair) => {
            return String.Format(
"{{\"method\":\"getSymbols\",\"params\":{{\"f\":\"f\"}}}}"
            );
        };
        PairsSource.Method = (JToken pairs) => {
            foreach (JToken pair in pairs["result"]) {
```

```
                Pairs.TryAdd(
                    pair["baseCurrency"].ToString() + "-" +
                    pair["quoteCurrency"].ToString(),
                    new CurrencyPair(this)
                );
            }
        };
```

Zdroj: Autor

4.4.1.7 Třída MarketProvider

V této třídě se provádí inicializace všech instancí třídy Market a jejich uložení do Listu, který je pak používaný jako zdroj dat pro většinu výpočtů. Po počáteční inicializaci se zavolá metoda CallMarkets (Kód 5), ve které se proiterují všechny instance Market a zavolá se služba, jenž získá všechny dostupné měnové páry pro danou burzu a uloží je. Tato akce by měla proběhnout paralelně pro všechny Market najednou místo standardního sekvenčního postupu.

Kód 5: Metoda CallMarkets třídy MarketProvider

```
private IEnumerable<Task> CallMarkets()
{
    return Markets.Select(async (market) => {
        await SourceCallerProvider
            .GetSourceCallerBySourceInformationType(
                (market as dynamic).PairsSource
            )
            .Call(market, (market as dynamic).PairsSource);
    });
}
```

Zdroj: Autor

4.4.1.8 Interface ISourceCaller

Interface ISourceCaller je interface, který musí implementovat všechny třídy z namespace MarketBot.Services.Callers kromě SourceCalllerProvider, jenž přistupují k nějakému zdroji dat. Požaduje implementaci pouze jedné metody, která vrací Task, protože se s ní bude pracovat asynchronně. (Kód 6)

Kód 6: Interface ISourceCaller

```
public interface ISourceCaller {
    Task Call(Market market, SourceInformation sourceInformation); }
```

Zdroj: Autor

4.4.1.9 Třída RestCaller a ostatní třídy implementující ISourceCaller

Třída RestCaller a ostatní třídy implementující ISourceCaller slouží jako oddělení mezi definicí zdroje dat (SourceInformation) a nějakého klienta, který provádí připojení na daný zdroj. Metoda Call (Kód 7) sice přijímá jako první parametr proměnnou market, kterou nikde nevyužívá, ale tato proměnná je do této metody předává z důvodů budoucích rozšíření.

Kód 7: Metoda Call třídy RestCaller

```
public async Task Call(
    Market market,
    SourceInformation sourceInformation
) {
    RestApi api = sourceInformation as RestApi;
    HttpClient client = new HttpClient();
    string stringResult = "";
    JToken result = null;

    stringResult = await client.GetStringAsync(api.BuildUri());

    result = JsonParser.ParseToJToken(stringResult);

    await Task.Run(() => api.Method(result));
}
```

Zdroj: Autor

4.4.1.10 Třída SourceCallerProvider

Protože se v aplikaci využívají 4 třídy implementující ISourceCaller, tak pro ulehčení práce s nimi vznikl SourceCallerProvider. Obsahuje pouze jednu další metodu kromě konstruktoru, která vrátí patřičný "Caller" podle typu SourceInformation. (Kód 8) Fungování této třídy je inspirována návrhovým vzorem Facade.

Kód 8: Metoda GetSourceCallerBySourceInformationType třídy SourceCallerProvider

```
public ISourceCaller GetSourceCallerBySourceInformationType(
    SourceInformation sourceInformation
) {
    switch (sourceInformation.GetType().ToString()) {
        case
"MarketBot.Models.SourceInformations.WebSocketApi":
            return WebSocketCaller;
        case
"MarketBot.Models.SourceInformations.RestApi":
            return RestCaller;
        case
"MarketBot.Models.SourceInformations.StaticSourceInformation":
            return StaticSourceCaller;
```

```
        case
"MarketBot.Models.SourceInformations.WebSourceInformation":
            return WebSourceCaller;
    }

    return null;
}
```

Zdroj: Autor

4.4.1.11 Třída MarketDataProvider

Třída MarketDataProvider je svou strukturou a fungování velmi podobná třídě MarketProvider, ale s tím rozdílem, že je odvozená od třídy BackgroundService. Třídy, jež jsou potomky BackgroundService, mají tu vlastnost, že běží na pozadí aplikace, a tudíž v nich mohou běžet nekonečné načítací smyčky. V tomto případě se provede paralelně připojení na všechny dostupné burzy a probíhá z nich načítání nabídek a poptávek až do ukončení program. (Kód 9)

Kód 9: Metody třídy MarketDataProvider

```
protected override async Task ExecuteAsync(
    CancellationToken cancellationToken
) {
    await MarketProvider.SetMarkets();
    await Task.WhenAll(CallMarkets());
}

private IEnumerable<Task> CallMarkets()
{
    return MarketProvider.Markets.Select(async (market) => {
        await SourceCallerProvider
            .GetSourceCallerBySourceInformationType(
                (market as dynamic).DataSource
            )
            .Call(
                market,
                (market as dynamic).DataSource
            );
    });
}
```

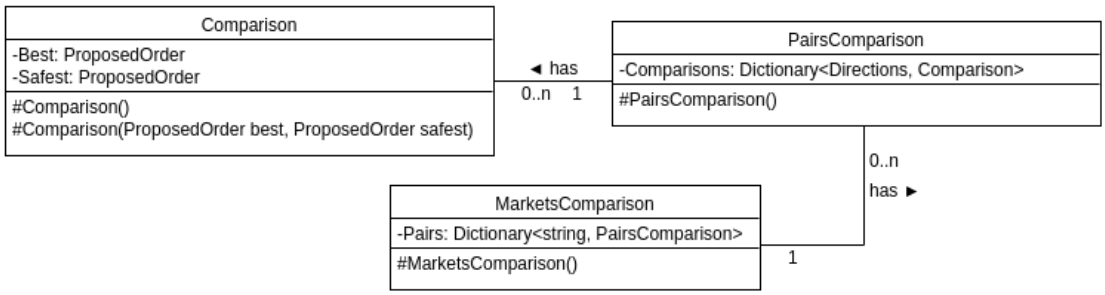
Zdroj: Autor

4.4.1.12 Třídy z namespace MarketBot.Models.Comparisons

Jedná se o třídy Comparison, PairsComparison a MarketsComparison (Obrázek 2), do kterých se ukládají vypočítané porovnání cen měn mezi burzami. Jsou tvořeny ve třídě

MarketComparator a jejich existence je omezená do doby, než jsou nahrazeny nově nalezenými porovnáními cen měn.

Obrázek 2: Diagram tříd z namespace MarketBot.Models.Comparisons



Zdroj: Autor

4.4.1.13 Třída ProposedOrder

Třída ProposedOrder je pouze data transfer object, ve kterém se ukládá vypočítaná nejvýhodnější směna pro daný měnový pár. Instance třídy ProposedOrder slouží jen k ulehčení práce při předávání výsledků na frontend.

4.4.1.14 Třída MarketComparator

MarketComparator je třída, ve které se provádí finální výpočty a předání dat na fronend. Stejně jako třída MarketDataProvider je tato třída odvozená od BackgroundService třídy, takže její běh probíhá na pozadí aplikace. Centrálním bodem této třídy je metoda ExecuteAsync, jenž obsahuje smyčku, ve které se provádí kód až do ukončení programu. Po spuštění programu trvá několik sekund, než se načte dostatek nabídek a poptávek na jednotlivých burzách. Algoritmus se v takové situaci uspí na určitý počet sekund a pokusí se operaci zopakovat. Pokud jsou požadavky splněny, algoritmus pokročí do další fáze, ve které vygeneruje všechny možné porovnání pro všechny burzy a ty následně, prostřednictvím Hubu, předá na frontend. (Kód 10)

Kód 10: Metoda ExecuteAsync třídy MarketComparator

```
protected override async Task ExecuteAsync(
    CancellationToken cancellationToken
) {
    while (!cancellationToken.IsCancellationRequested) {
        if (MarketProvider.Markets.Count < 2) {
            await Task.Delay(
                TimeSpan.FromSeconds(15),
```

```
                cancellationToken
            );
        }

        for (
            int i = 0;
            i < MarketProvider.Markets.Count;
            i++
        ) {
            for (
                int j = i + 1;
                j < MarketProvider.Markets.Count;
                j++
            ) {
                if (j == MarketProvider.Markets.Count) {
                    break;
                }

                Market market = MarketProvider.Markets[i];
                Market innerMarket = MarketProvider.Markets[j];

                if (market.Pairs.Count < 50 ||
                    innerMarket.Pairs.Count < 50
                ) {
                    await Task.Delay(
                        TimeSpan.FromSeconds(15),
                        cancellationToken
                    );

                    break;
                }

                MarketsComparison comparison = await Task.Run(
                    () => CreateMarketsComparison(
                        market,
                        innerMarket
                    )
                );

                await MarketsComparisonsHubContext.Clients.All.
                    SendAsync(
                        "ReceiveComparison",
                        market.Name + "-" + innerMarket.Name,
                        JsonParser.ParseToJToken(
                            comparison.Pairs
                        ).ToString()
                    );
            }
        }
    }
}
```

Zdroj: Autor

4.4.1.15 Třída MarketsComparisonsHub

Pro předání dat na frontend se využívá, jak už bylo dříve zmíněno, knihovna SignalR. Třída MarketsComparisons je potomkem třídy Hub ze SignalR. (Kód 11) Tělo této třídy je prázdné, protože se v aplikaci využívají pouze nativní metody třídy Hub a tato třída tedy slouží jen pro vytvoření endpointu v Startup.cs souboru.

Kód 11: Třída MarketsComparisonsHub

```
public class MarketsComparisonsHub : Hub {}
```

Zdroj: Autor

4.4.1.16 Služby

Některé třídy využité v aplikaci (Kód 12) pracují jako služby. To znamená, že se při startu aplikace vytvoří po jedné instanci těchto tříd, ke kterým je pak možné pomocí DI přistoupit z jiné služby či DI podporujících objektů.

Kód 12: Služby využívané v aplikaci

```
services.AddSingleton<WebSocketCaller>();
services.AddSingleton<RestCaller>();
services.AddSingleton<StaticSourceCaller>();
services.AddSingleton<WebSourceCaller>();

services.AddSingleton<SourceCallerProvider>();

services.AddSingleton<MarketProvider>();

services.AddHostedService<MarketDataProvider>();
services.AddHostedService<MarketComparator>();
```

Zdroj: Autor

4.4.1.17 Ostatní nezmíněné třídy a vlastnosti

Aplikace obsahuje několik pomocných tříd, které pomáhají s formátováním textu či parsování dat z a do JSON. Jednou z nezmíněných tříd je SnapshotServerController. Ten se při prvním požadavku uživatele na webové rozhraní pokusí najít předrenderované soubory s frontend a pokud je najde, tak je vrátí uživateli. Tím se docílí vlastnosti zvané Server-Side Rendering a uživateli se tedy zobrazí aplikace rychleji.

4.4.1.18 Refaktorizace

Počáteční verze aplikace využívala pouze jedné třídy odvozené z BackgroundService, ve které se prováděly všechny potřebné operace. To se během testů projevilo jako neefektivní a pomalé. Došlo tedy k rozdělení této třídy na dvě na sobě nezávislé třídy.

Až do poloviny vývoje aplikace třída Market obsahovala vše potřebné pro vytvoření spojení se zdroji dat. Kód třídy Market se tak pro některé burzy s komplikovanějšími API stával dosti nepřehledný, a proto byla třída Market přepsána, tak aby byla obecnější a využívala delegáty.

4.4.1.19 Problémy

Vývoj backendu byl komplikován faktem, že kryptoměnové burzy stále mění své API. Aby aplikace i nadále mohla s takovou burzou pracovat, bylo vždy nutné provést odpovídající úpravy, což bylo časově náročné. Burza nazývající se OKEx byla vyjmuta z používaných burz, protože nebylo možné změny v jejím API zapracovat do aplikace.

4.4.2 Frontend

4.4.2.1 Princip

Při svém startu, aplikace vytvoří a nastaví Redux store (dále jen store) se všemi potřebnými náležitostmi. Následuje inicializace ReactJS komponenty MarketsComparisons, jenž vytvoří připojení na SignalR Hub (dále jen Hub) a začne zapisovat získané údaje do store. Při svém vykreslování zobrazí získaná data. Zároveň si ze store načte páry burz komponenta Header a vypíše je jako tlačítka menu.

4.4.2.2 Inicializace aplikace v index.tsx

Tento soubor se spustí jako první a probíhá v něm prvotní inicializace a získávání potřebných dat. Vytváří se zde objekt History, který obsahuje historii procházení stránek a připravuje se zde počáteční stav store. Oba dva tyto objekty se následně předají k finální konfiguraci store. Po vytvoření store, proběhne render aplikace. (Kód 13) Aplikace dále využívá ConnectedRouter, který usnadňuje tvoření linků, přechodů na jiné komponenty a zachovává funkčnost tlačítka “Zpět”.

Kód 13: Kořen ReactJS aplikace

```
render(
  <Provider store={ store }>
    <ConnectedRouter history={ history as any }>
      <App />
    </ConnectedRouter>
  </Provider>,
  rootElement
);
```

Zdroj: Autor

4.4.2.3 Konfigurace Redux store

Konfigurace store probíhá ve funkci configureStore v souboru configureStore.ts. Tato funkce při svém běhu vytvoří store podle počátečního state, do kterého připojí všechny reducery, action creatory, middlewary a případné enhancery. Definice některých z těchto elementů pro tuto aplikaci se nachází v MarketsComparisons.ts.

State aplikace je definován množinou interfaců, které společně mapují data získaná z backendu, tak aby se s nimi dalo pracovat jako s objekty. (Kód 14)

Kód 14: Interface IMarketsComparisonsState a IMarketsComparison

```
export interface IMarketsComparisonsState {
  connection: HubConnection | null;
  marketsComparisons: IMarketsComparison[];
}

export interface IMarketsComparison {
  Name: string;
  Pairs: IPairsComparison[];
}
```

Zdroj: Autor

Funkce, které nějakým způsobem ovlivňují store, se nazývají “action creators”. V této aplikaci se nacházejí dvě. První z nich se jménem “initialize” (Kód 15) vytvoří připojení na Hub, dispatchne action, která zapíše do store, že je připojení inicializováno a následně podobným způsobem čte data z Hub a opět je dispatchuje do store. Poslední funkce je “destroy”, která se zavolá při ukončení aplikace, aby se přerušilo spojení s Hubem.

Kód 15: Action creator funkce initialize

```
initialize: (): IAppThunkAction<KnownAction> =>
  async (dispatch, getState) => {
```

```
  const connection = new HubConnectionBuilder()
    .withUrl('/markets-comparisons-hub')
    .build();

  connection.on('ReceiveComparison', (name, pairs) => {
    dispatch({
      type: 'RECEIVE_COMPARISON',
      marketsComparison: {
        Name: name,
        Pairs: JSON.parse(pairs)
      }
    })
  });

  await connection.start()
    .then(() => dispatch({
      type: 'INITIALIZED',
      connection
    }))
    .catch(err => console.log(err));
},
```

Zdroj: Autor

Reducer přijímá dva typy actions, podle kterých určuje, jak ovlivnit data ve store. Akce typu “INITIALIZED” sebou nese objekt reprezentující připojení na Hub. (Kód 16) Při obdržení akce typu “RECEIVE_COMPARISON”, reducer ověří, zda se ve store nachází dané marketsComparison, a pokud ne, tak jej do store přidá.

Kód 16: Část kódu reduceru

```
export const reducer: Reducer<IMarketsComparisonsState> =
  (state: IMarketsComparisonsState, incomingAction: Action) => {
  const action = incomingAction as KnownAction;

  switch (action.type) {
    case 'INITIALIZED':
      return {
        connection: action.connection,
        marketsComparisons: state.marketsComparisons
      };
```

Zdroj: Autor

4.4.2.4 Komponenta App

Jedná se o první komponentu, která se začne renderovat po spuštění aplikace. V této aplikaci slouží spíše jako wrapper pro komponenty s vyšší funkcionalitou. Na poslední řádce se nachází kód (Kód 17), který je potřeba pro hot reload komponent, a to zapříčiní, že při vývoji nebude nutné aktualizovat stránku, aby se projevily změny.

Kód 17: Kód pro inicializaci hot reloadu komponent

```
export default hot(module)(App);
```

Zdroj: Autor

4.4.2.5 Komponenta Header

Tato komponenta je napojená na store, a proto má přístup k datům v něm uloženým, ale také ke všem actions creators. (Kód 18) Úkolem této komponenty je vykreslit menu v horní části obrazovky, jenž obsahuje tlačítka na přecházení na jednotlivé porovnání burz. Když se Header renderuje, tak si vytáhne data všech porovnání burz ze store, proiteruje je a vygeneruje z nich potřebná tlačítka.

Kód 18: Napojení na store

```
export default connect(
  (state: IApplicationState) => state.marketsComparisons,
  MarketsComparisonsStore.actionCreators
)(Header as any);
```

Zdroj: Autor

4.4.2.6 Komponenta Main

V komponentě Main se renderuje celé tělo aplikace. Komponenta Switch dokáže naslouchat na změny URL, a podle toho provést redirect na jinou stránku (Kód 19) či vykreslit jinou komponentu.

Kód 19: Komponenta Main

```
export default class Main extends React.Component<{}, {}> {
  public render() {
    return <MainElem role="main">
      <Switch>
          <Redirect exact from="/" to="/markets-comparisons" />
          <Route path='/markets-comparisons/:name?'
          component={ MarketsComparisons }
          />
      </Switch>
    </MainElem>
  }
}
```

Zdroj: Autor

4.4.2.7 Komponenta MarketsComparisons

Komponenta MarketsComparisons je komponenta, která startuje a ukončuje celý proces získávání dat z backendu a následné vypisování uživateli. Ke správnému fungování musí props této komponenty obsahovat RouteComponentsProps. (Kód 20) Tento objekt obsahuje parametry současné Route path, a tak je možné identifikovat správný výpis k zobrazení.

Kód 20: Props předané do MarketsComparisons

```
type MarketsComparisonsProps =
  MarketsComparisonsStore.IMarketsComparisonsState
  & typeof MarketsComparisonsStore.actionCreators
  & RouteComponentProps<{ name: string }>;
```

Zdroj: Autor

Metoda render vykresluje oba dva dolní bloky kompletně se všemi informaci, a proto musel být kód rozdělen do několika metod a dalších pomocných objektů. Implementace využívá faktu, že ReactJS dokáže ukládat komponenty do proměnných, a tím odstínit komplexní struktury. (Kód 21)

Kód 21: Část metody renderMarketsComparison

```
const tableBody = TableBuilder
  .renderBody(marketComparison.Pairs);

return <FlexDiv>
  {["LTR", "RTL"].map(direction =>
    <div key={direction.toLocaleLowerCase()}>
      <ComparisonsTable>
        {TableBuilder.renderHead(direction)}

        <tbody>
          {tableBody[direction]}
        </tbody>
      </ComparisonsTable>
    </div>
  )}
</FlexDiv>
```

Zdroj: Autor

4.4.2.8 Třída TableBuilder

Pomocná třída, která se stará renderování dat směn do tabulek. Třída TableBuilder není ReactJS komponenta, ale její metody generují JSX.Element, který je možné následně

vykreslit v patřičné komponentě. Kód některých metod se může jevit jako velmi nepřehledný (Kód 22). To je způsobené tím, že vykreslovaná data obsahují několik úrovní zanoření.

Kód 22: Metoda renderBody

```
public static renderBody(pairs: object): object {
  const tableBody = {"LTR": [], "RTL": []};

  Extensions.toArray(pairs).map((pairKeyVal: any) =>
    Extensions.toArray(pairKeyVal.value.Comparisons)
      .filter(
        (comparisonKeyVal: any) =>
        comparisonKeyVal.value !== null
      )
      .map(
        (comparisonKeyVal: any) =>
        tableBody[comparisonKeyVal.key].push(
          TableBuilder.renderComparison(
            pairKeyVal.key,
            comparisonKeyVal.value
          )
        )
      )
  );

  return tableBody;
}
```

Zdroj: Autor

4.4.2.9 Vzhled aplikace

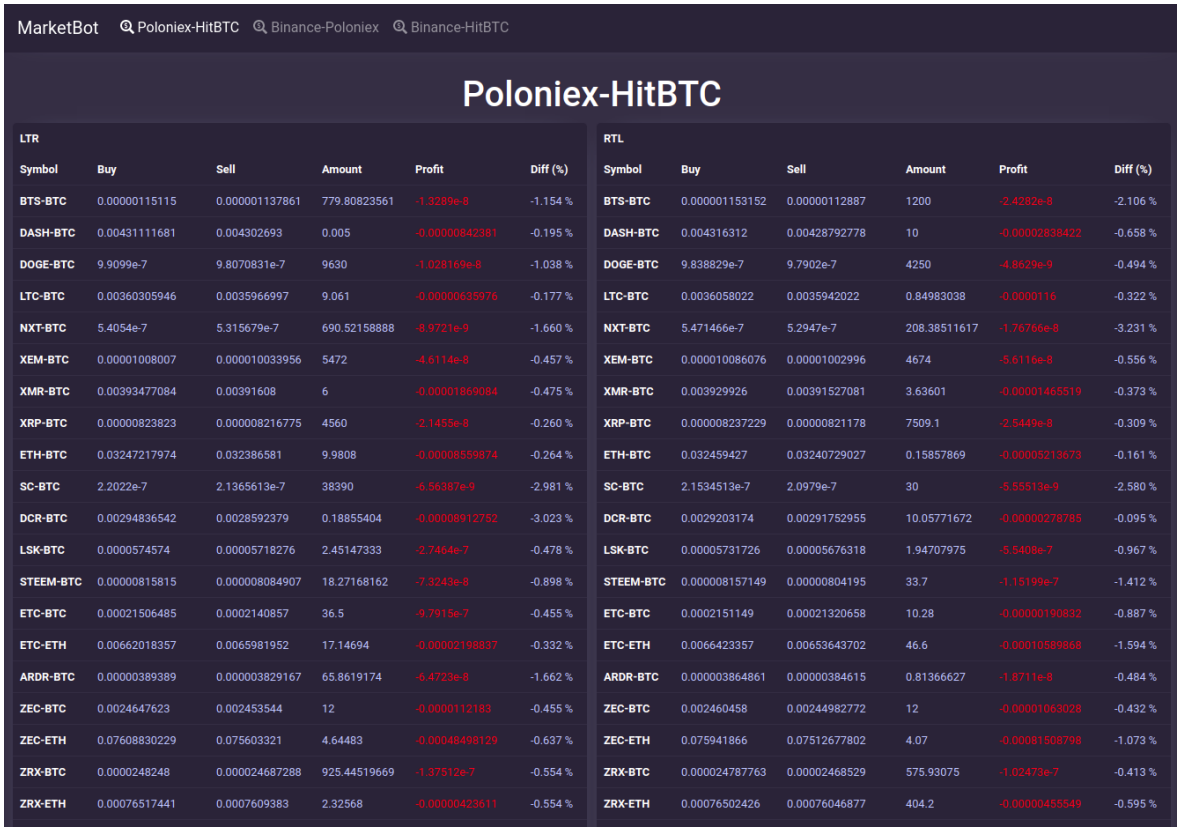
Vzhled aplikace je postaven na Bootstrap 4. Bootstrap frontendová knihovna, která obsahuje předdefinované grafické elementy. Aplikace tyto komponenty využívá za pomocí knihoven react-bootstrap a styled-component. (Kód 23) Styly (CSS) jsou v tomto případě uloženy společně v JS. Výsledný vzhled byl upraven do tmavé fialové barvy pro lepší čitelnost.

Kód 23: Ukázka využití styled-components s react-bootstrap komponentou

```
const CustomNavbar = styled(Navbar)`
  background-color: #2A2438; box-shadow: 0px 0px 30px -25px #BBC8FF;
`
```

Zdroj: Autor

Obrázek 3: Výsledný vzhled rozhraní



Zdroj: Autor

4.4.2.10 Další vlastnosti aplikace

Jak už bylo dříve zmíněno, frontend je samostatná aplikace, ale v produkčním buildu a základním vývojem nastavení, je frontend inicializován backendem, takže při změně v backendu, dojde k restartu i frontendu. To je nežádoucí chování, jelikož to prodlužuje dobu opětovného startu aplikace. Aby se tomuto zabránilo, je nutné nastavit proxy mezi frontendovým vývojovým serverem a backend aplikací. (Kód 24)

Kód 24: Nastavení proxy

```
const proxy = require('http-proxy-middleware')

module.exports = function(app) {
  app.use(proxy('/markets-comparisons-hub', {
    target: 'ws://localhost:5001',
    secure: false,
    ws: true
  }))
```

Zdroj: Autor

Protože je aplikace založená na CRA, obsahuje service worker, jenž v případě výpadku internetového připojení zobrazí uživateli data z cache prohlížeče.

4.4.2.11 Problémy

Během vývoje se vyskytlo několik problémů. Hlavním z problémů byla nefungující proxy mezi frontend development serverem a backendem, takže se při vývoji na backendu musel restartovat i frontend development server. Problém se vyřešil aktualizací aplikace na verzi v té době v development větvi projektu.

Další problém, se kterým se autor potýkal během vývoje, bylo to, že TypeScript ve verzi 4.5+ a styled-components se transpilovaly přes 20 minut. V době nebylo možné snížit ani povýšit verzi TypeScriptu, která by obsahovala požadované funkcionality. Situace se vyřešila tím, že autor počkal s vývojem na vydání novější verze TypeScriptu.

4.5 DevOps

4.5.1 Docker

Aplikace ke svému chodu v development módu vyžaduje mít instalovaný ASP.NET Core, patřičné knihovny k němu a také NodeJS s potřebnými knihovnami. To samé potřebuje k vytvoření produkčního build aplikace. Instalace všech těchto závislostí na produkční server značně komplikuje správu a údržbu zvláště za předpokladu, že na ten samý server se nainstalují další aplikace se svými závislostmi. Tato práce proto obsahuje dva Dockerfile obsahující Docker image pro build a pro chod aplikace. Image pro build aplikace obsahuje kompletní .NET Core SDK a navíc NodeJS. Během buildu produkční verze aplikace se vygenerují a zkompilují všechny potřebné náležitosti za pomocí toho image. (Kód 25) Samotná aplikace je na serveru spuštěna v conteineru z druhého image, jenž obsahuje pouze runtime pro .NET Core. (Kód 26) Ten je značně menší a optimalizován pro produkční prostředí.

Kód 25: Dockerfile pro build image

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1
RUN apt-get update -yq && apt-get upgrade -yq
RUN curl -sL https://deb.nodesource.com/setup_13.x | bash -
RUN apt-get install -yq nodejs build-essential
WORKDIR /app
```

Zdroj: Autor

Kód 26: Dockerfile pro runtime image

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1

WORKDIR /app

ENTRYPOINT [
    "dotnet",
    "bin/Release/netcoreapp3.1/publish/MarketBot.dll"
]
```

Zdroj: Autor

4.5.2 Ansible

Jak už bylo dříve zmíněno, vytvoření produkčního build vyžaduje provedení několika kroků, jako například build Docker image, startování/vypínání conteinerů či volání dotnet příkazů. Pro zefektivnění toho procesu, byl do aplikace přidán Ansible playbook, který se následně spustí na produkčním serveru a provede všechny požadované kroky, včetně spuštění aplikace, bez nutnosti cokoliv jiného vykonávat.

4.5.3 GitLab a CI

Kód celé aplikace byl od počátku vývoje verzován za pomoci GitLabu. Po pushnutí commitu do repozitory se spustí GitLab CI Pipeline, která se pokusí aplikaci vybuildit a spustit. Tím se ověří v omezené míře, že nově přidaný kód není závadný.

4.5.4 Nasazení

Pro testovací nasazení aplikace byl využit mini počítač, na němž se odladil proces nasazení a generování TSL certifikátu pro testovací doménu. Finální aplikace byla nasazena pomocí Ansible na server, jenž disponuje 16 jádry o taktu 2 GHz a 32 GB RAM, což by mělo být pro chod aplikace více než dostačující. Autor zvolil pro nasazení tento server, protože jiný neměl. Aby bylo možné se na web dostat z internetu, bylo nutné na serveru spustit reverzní proxy, která přesměrovává HTTP požadavky do docker containeru s aplikací. K tomuto účelu posloužil Nginx Docker container. Finální krokem bylo vygenerování Let's Encrypt certifikátu pro doménu.

4.5.5 Testování

Aplikace byla v průběhu vývoje testována, ať už jako celek, tak i její jednotlivé části. Po nasazení na produkční server se občas stávalo, že nastala situace, při které

frontend přestal zobrazovat nová data. Po bližší inspekci autor zjistil to, že dochází k přerušení komunikace se SignalR Hubem na straně frontendu. Nepodařilo se najít příčinu, a tak byl do frontendové aplikace přidán kód, který jednou za několik minut aktualizuje stránku, což daný problém pro běžného uživatele vyřešilo.

5 Výsledky a diskuse

5.1 Zhodnocení autorem

Vývoj aplikace, která je rozdělená do dvou částí na backend a frontend sice zjednodušuje dohledávání bugů a do jisté míry je možné tyto aplikace vyvíjet samostatně, ale zároveň to přidává velké množství požadavků na znalosti vývojáře, protože musí umět pracovat, jak s frontendovými, tak i backendovými technologiemi.

Autor práce má zkušenosti s webovým frameworkem Symfony na PHP, který funguje velmi podobně jako ASP.NET Core, takže vývoj backend aplikace probíhal v celku bez větších zdržení. Jediné, co dělalo problémy, byla práce pomocí nativní knihovny pro WebSockety, protože vyžaduje práci s bufferem a stávalo se, že se do něj přijatá zpráva celá nevešla, takže bylo nutné ještě ověřovat integritu zpráv.

Frontendová část, ačkoliv neobsahuje tolik kódu, vyžadovala o dost více práce než backend. Frontendové technologie, jako ReactJS či Redux fungují diametrálně jinak, než na co byl autor zvyklý z backendu. Autor se domnívá, že využití Reduxu v této aplikaci bylo možná až nadbytečné, protože kód aplikace není zas tak rozsáhlý na frontendu. Redux totiž vyžaduje naprogramování velké množství kódu a přizpůsobení fungování celé aplikace.

Zajímavostí je, že se někdy podaří najít arbitráž na burze Poloniex dříve, než se nabídka či poptávka, kterých se to týká, zobrazí na burze. Je to pravděpodobně způsobené tím, že frontend Poloniexu musí vykreslovat mnohem více věcí, a proto je někdy pomalejší než tato aplikace.

5.2 Návrhy na vylepšení a nové funkce

Na frontend se předává pomocí SignalR Hubu vždy celé porovnání všech párů pro daný burzovní pár, což je velmi neefektivní a značně to zpomaluje aplikaci. Situace by se dala vyřešit tím, že by se předávaly pouze nově vypočítané směny. V současné implementaci toto nelze provést a vyžadovalo by to větší refaktorizaci kódu. Aplikace je postavená na starší verzi .NET Core a i frontendové knihovny už mají novější verze (2021), takže aktualizace by aplikaci prospěla. V současné chvíli je aplikace jednoúčelová, ale autor by jí rád rozšířil. Opětovně by se přidalo hledání trojúhelníkových arbitráží s tím, že by se zapracovalo na optimalizaci. Další funkcionalitou by mohlo být měření metrik jako výskyt arbitráží na určitém měnovém páru.

6 Závěr

Dílčím cílem práce bylo popsat postupy implementace, využité přístupy a technologie, což bylo splněno primárně v kapitole obsahující teoretická východiska. Tyto východiska byly rozděleny do kategorií „Obchodní“ a „Softwarové“, protože řešený problém vyžaduje nejen znalost tvorby webových aplikací, ale také to jak fungují burzy.

Hlavním cílem práce bylo navrhnout a implementovat aplikaci, která získává real-time data z burz a hledá mezi nimi arbitráže. Toho cíle bylo dosáhnuto v praktické části, kde byla nejdříve provedena analýza požadavků a proveditelnosti. Následovala analýza existujících řešení, návrh uživatelského rozhraní a implementační kapitola. V kapitole „Implementace“ se autor zabývá nejdříve backendovou částí aplikace. Popisuje zde jednotlivé třídy a důležité části kódu, ale také postupy vývoje a vyskytnuté problémy. Ve frontendové části autor prochází aplikaci po jednotlivých komponentách v pořadí, ve kterém jsou inicializovány při spuštění, a jejich implementace je autorem okomentována. Postup a způsob nasazení aplikace byly okomentovány v kapitole „DevOps“.

Získané poznatky z vývoje a návrhy na vylepšení byly projednány v kapitole „Výsledky a diskuse“, kde autor zároveň zhodnotil tuto práci.

Výsledkem je aplikace, která uživatelům nabízí velmi rychlý zdroj informací o možných arbitrážích na kryptoměnových burzách. Aplikace je multiplatformní a je jí snadné nasadit prakticky na jakýkoliv linuxový stroj s Dockerem.

7 Seznam použitých zdrojů

1 SHRIVASTAVA, Gulshan, SHARMA, Kavita, LE, Dac-Nhuong. *Cryptocurrencies and Blockchain Technology Applications.* Spojené státy: John Wiley & Sons, 2020, 336 s. ISBN 978-1642821246.

2 PEILLARD, Benjamin. *How do Cryptocurrencies and the Crypto Market Work?* [online]. 2019 [cit. 2021-03-02]. Dostupné z: https://medium.com/hashingsystems/how-do-cryptocurrencies-and-the-crypto-market-work-8298de896848

3 RABIEJ, Marcin. *Cryptocurrency arbitrage strategies — part I* [online]. 2018 [cit. 2019-12-21]. Dostupné z: https://medium.com/coinmonks/cryptocurrency-arbitrage-strategies-part-i-20e9dd327919.

4 Microsoft Docs. *C# documentation* [online]. 2019 [cit. 2019-12-23] Dostupné z: https://docs.microsoft.com/en-gb/dotnet/csharp/.

5 ALBAHARI, Joseph, ALBAHARI, Ben. *C# 7.0 in a Nutshell.* Spojené státy: O'Reilly Media, 2017, 1056 s. ISBN 978-14-9198-765-0.

6 PRICE, Mark J. *C# 8.0 and .NET Core 3.0: modern cross-platform development : build applications with C#, .NET Core, Entity Framework Core, ASP.NET Core, and ML.NET using Visual Studio Code*. Fourth edition. Velká Británie: Packt Publishing, 2019, 818 s. ISBN 978-178-8471-572.

7 NAGEL, Christian. *Professional C# 7 and .NET Core 2.0.* Spojené státy: John Wiley & Sons, 2018, 1440 s. ISBN 978-11-1944-927-0.

8 Microsoft Docs. *Introduction to ASP.NET Core | Microsoft Docs* [online]. 2020 [cit. 2021-02-06]. Dostupné z: https://docs.microsoft.com/en-gb/aspnet/core/introduction-to-aspnet-core

9 CHAUGULE, Menka. *Kestrel Web Server & Reverse Proxy in ASP.NET Core* [online]. 2020 [cit. 2021-02-06]. Dostupné z: https://medium.com/@menkachaugule/kestrel-web-server-reverse-proxy-in-asp-net-core-ecd34b98d6ca

10 SANDERSON, Steven. *Pro ASP.NET MVC Framework: Expert's voice in .NET*. Spojené státy: Apress, 2009, 550 s. ISBN 978-1-4302-1008-5.

11 Entity Framework Core. *Entity Framework Core Tutorials* [online]. 2020 [cit. 2021-02-06]. Dostupné z: https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx

12 LOMBARDI, Andrew. *WebSocket : Lightweight Client-Server Communications* [online]. Spojené státy: O'Reilly Media, 2015, 144 s. [cit. 2021-02-06]. ISBN 978-1-4493-6925-5. Dostupné z: https://www.lehmanns.de/shop/mathematik-informatik/33296205-9781449369255-websocket

13 BOTTIAU, David. *Advanced realtime streaming with SignalR in .NET Core* [online]. 2020 [cit. 2021-02-06]. Dostupné z: https://medium.com/@dbottiau/advanced-realtime-streaming-with-signalr-in-net-core-2e38fce26fbb

14 HÁMORI, Ferenc. *The History of React.js on a Timeline* [online]. 2018 [cit. 2021-02-07]. Dostupné z: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/

15 ARCHER, Ralph. *ReactJS: For Web App Development*. Spojené státy: CreateSpace Independent Publishing Platform, 2015, 94 s. ISBN 978-1519791689.

16 React – A JavaScript library for building user interfaces. *Introducing Hooks – React* [online]. 2018 [cit. 2021-02-07]. Dostupné z: https://reactjs.org/docs/hooks-intro.html

17 GARREAU, Marc, FAUROT, Will. *Redux in Action*. Spojené státy: Manning Publications, 2018, 312 s. ISBN 978-1617294976.

18 Redux. *Redux Fundamentals, Part 1: Redux Overview* [online]. [cit. 2021-02-07]. Dostupné z: https://redux.js.org/tutorials/fundamentals/part-1-overview

19 FREEMAN, Adam. *Essential TypeScript: From Beginner to Pro*. Spojené státy: Apress, 2019, 546 s. ISBN 9781484249789.

20 HIWARALE, Uday. *A quick introduction to "Type Declaration" files and adding type support to your JavaScript packages* [online]. 2020 [cit. 2021-02-07]. Dostupné z: https://medium.com/jspoint/typescript-type-declaration-files-4b29077c43

21 POULTON, Nigel. *Docker Deep Dive*. Velká Británie: Packt Publishing, 2020, 249 s. ISBN 978-1800565135.

22 JANETAKIS, Nick. *Understanding How the Docker Daemon and Docker CLI Work Together* [online]. 2017 [cit. 2021-02-09]. Dostupné z:

https://nickjanetakis.com/blog/understanding-how-the-docker-daemon-and-docker-cli-work-together

23 Docker Documentation. *Docker overview* [online]. 2021 [cit. 2021-02-09]. Dostupné z: https://docs.docker.com/get-started/overview/

24 SOPPELSA, Fabrizio, KAEWKASI, Chanwit. *Native Docker Clustering with Swarm*. Velká Británie: Packt Publishing, 2016, 280 s. ISBN 978-1786467607.

25 KEDAR, Shahar. *5 Reasons We Love Using Ansible for Continuous Delivery* [online]. 2014 [cit. 2021-02-09]. Dostupné z: https://www.bigpanda.io/blog/5-reasons-we-love-using-ansible-for-continuous-delivery/

26 HOCHSTEIN, Lorin. *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. Spojené státy: O'Reilly Media, 2014, 334 s. ISBN 978-1491916148.

27 HOGBIN WESTBY, Emma Jane. *Git for Teams*. Spojené státy: O'Reilly Media, 2015, 355 s. ISBN 978-1491911204.

28 Git. *Getting Started - A Short History of Git* [online]. [cit. 2021-02-09]. Dostupné z: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

29 EVERTSE, Joost. *Mastering GitLab 12: Implement DevOps culture and repository management solutions*. Velká Británie: Packt Publishing, 2019, 608 s. ISBN 978-1789534061.

30 *History of GitLab* [online]. [cit. 2021-02-09]. Dostupné z: https://about.gitlab.com/company/history/

31 RAZA, Syed. *Complete DevOps Gitlab and Kubernetes*. Spojené státy: Stone River eLearning, 2020, 584 s. ISBN 978-1838828042.

32 SACOLICK, Isaac. *What is CI/CD? Continuous integration and continuous delivery explained* [online]. 2020 [cit. 2021-02-09]. Dostupné z: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

33 Let's Encrypt - Free SSL/TLS Certificates. *How It Works* [online]. 2019 [cit. 2021-02-09]. Dostupné z: https://letsencrypt.org/how-it-works/

34 АЛЕКСАНДРОВИЧ, Малимонов Денис. *Intra-Exchange-Crypto-Arbitrage* [online]. 2020 [cit. 2021-02-16]. Dostupné z: https://github.com/tg-bomze/Intra-Exchange-Crypto-Arbitrage

35 *OneExBit* [online]. 2018 [cit. 2021-02-16]. Dostupné z: https://oneexbit.com/

8 Přílohy

Příloha 1 – CD se zdrojovým kódem aplikace a soubory pro build a nasazení aplikace