Aplikace pro monitorování trhů s kryptoměnami

Bc. Daniel Večeř

Diplomová práce
2021

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav elektroniky a měření

Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Daniel Večeř**
Osobní číslo:	**A19408**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Bezpečnostní technologie, systémy a management**
Forma studia:	**Prezenční**
Téma práce:	**Aplikace pro monitorování trhů s kryptoměnami**
Téma práce anglicky:	**A Cryptocurrency Markets Monitoring Application**

Zásady pro vypracování

1. Popište současný stav technologií pro vývoj a zabezpečení webových aplikací.
2. Zaměřte se především na frameworky .NET a ASP.NET.
3. Navrhněte aplikaci s využitím popsaných technologií.
4. Navrhněte způsob zabezpečení komunikace mezi klientem a serverem.
5. Realizujte vývoj navržené aplikace a popište její klíčové části.
6. Demonstrujte výsledky.

Forma zpracování diplomové práce: **Tištěná/elektronická**

Seznam doporučené literatury:

1. PRINCE, Mark. 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. 4. Birmingham: Pack Publishing, 2019. ISBN 978-1788478120.
2. ASP.NET | Open-source web framework for .NET. .NET | Free. Cross-platform. Open Source. [online]. Dostupné z: https://dotnet.microsoft.com/apps/aspnet
3. NAGEL, Christian. Professional C# 7 and .NET Core 2.0. 11th edition. Indianapolis: Wrox, a Wiley Brand, 2018. ISBN 978-1119449270.
4. ALBAHARI, Joseph a Ben ALBAHARI. C# 7.0 in a nutshell. 7th edition. Sebastopol: O´Reilly, 2018. ISBN 978-1491987650.
5. V. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. Bringing the web up to speed with WebAssembly. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY: ACM, 2017, s. 185-200. ISBN 978-1-4503-4988-8.

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **15. ledna 2021**
Termín odevzdání diplomové práce: **17. května 2021**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

Ing. Milan Navrátil, Ph.D. v.r.
ředitel ústavu

Ve Zlíně dne 15. ledna 2021

Jméno, příjmení: Daniel Večeř
Název diplomové práce: Aplikace pro monitorování trhů s kryptoměnami
Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 20.5. 2021

Daniel Večeř v.r.
podpis studenta

ABSTRAKT

Hlavním cílem této diplomové práce je popis problematiky vývoje moderních webových aplikací s využitím frameworku Blazor WebAssembly, který je relativně novou součástí frameworku .NET. Teoretická část práce se věnuje především frameworkům .NET, ASP.NET Core, Blazor a dalším klíčovým technologiím z oblasti vývoje webových aplikací. Součástí teoretické části práce je i stručný úvod do principů fungování kryptoměn. Praktická část navazuje na informace poskytnuté v teoretické části a demonstruje použití popsaných technologií při vývoji webové aplikace, jejímž účelem je umožnit uživatelům monitorování trhů s kryptoměnami v reálném čase. V úvodu praktické části jsou nejprve definovány požadavky na aplikaci samotnou, následně je proveden návrh základní struktury aplikace a detailní popis její realizace, který zahrnuje ukázky kódu. Speciální pozornost je věnována realizaci zabezpečení aplikace. Finální kapitola praktické části se pak věnuje demonstraci funkcionality vytvořené aplikace.

Klíčová slova: .NET, Blazor WebAssembly, ASP.NET Core, MongoDB, C#, kryptoměny

ABSTRACT

The primary goal of this master's thesis is to describe the practice of developing modern web applications using the Blazor WebAssembly framework, which is a relatively new part of the .NET framework. The theoretical part of the thesis describes the .NET, ASP.NET Core and Blazor frameworks, as well as other key technologies commonly used for developing web applications. A brief description of the basic principles of cryptocurrencies is also included. The practical part of this thesis then uses the information provided in the theoretical part to show the process of creating a web application based on the described technologies, the purpose of which is to allow users to monitor cryptocurrency markets in real time. It begins by stating a list of requirements for the developed application, before moving on to describing the design of the application and the realization itself, including various code samples. Special attention is given to describing the process of securing the application. The final chapter is devoted to demonstrating the functionality of the created application.

Keywords: .NET, Blazor WebAssembly, ASP.NET Core, MongoDB, C#, cryptocurrencies

Tímto chci poděkovat vedoucímu práce Ing. Eriku Královi, Ph.D. za odborné rady, organizaci a konzultaci diplomové práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Tato diplomová práce se zabývá tématem vývoje a zabezpečení moderních webových aplikací postavených na platformě .NET a příbuzných technologiích, a to v kontextu návrhu a vývoje webové aplikace zaměřené na monitorování trhů s kryptoměnami.

Teoretická část diplomové práce je zaměřena hlavně na analýzu a popis jednotlivých technologií pro vývoj webových služeb, které jsou v současné době k dispozici – především pak framework .NET v jeho nejnovější verzi. Velká pozornost je také samozřejmě věnována dalším technologiím, které spadají pod tento framework, či jsou nějakým způsobem příbuzné. Zde se jedná především o programovací jazyk C#, ASP.NET Core (součást frameworku .NET, která je specificky zaměřena na webový vývoj), Blazor (framework, který umožňuje vývoj webových aplikací s využitím C# a HTML) a MongoDB (dokumentový databázový systém řadící se mezi takzvané NoSQL databáze). V menší míře se teoretická část rovněž věnuje problematice kryptoměn jako takových a vysvětlení principů toho, jak fungují a jakým způsobem je v současnosti možné s nimi obchodovat.

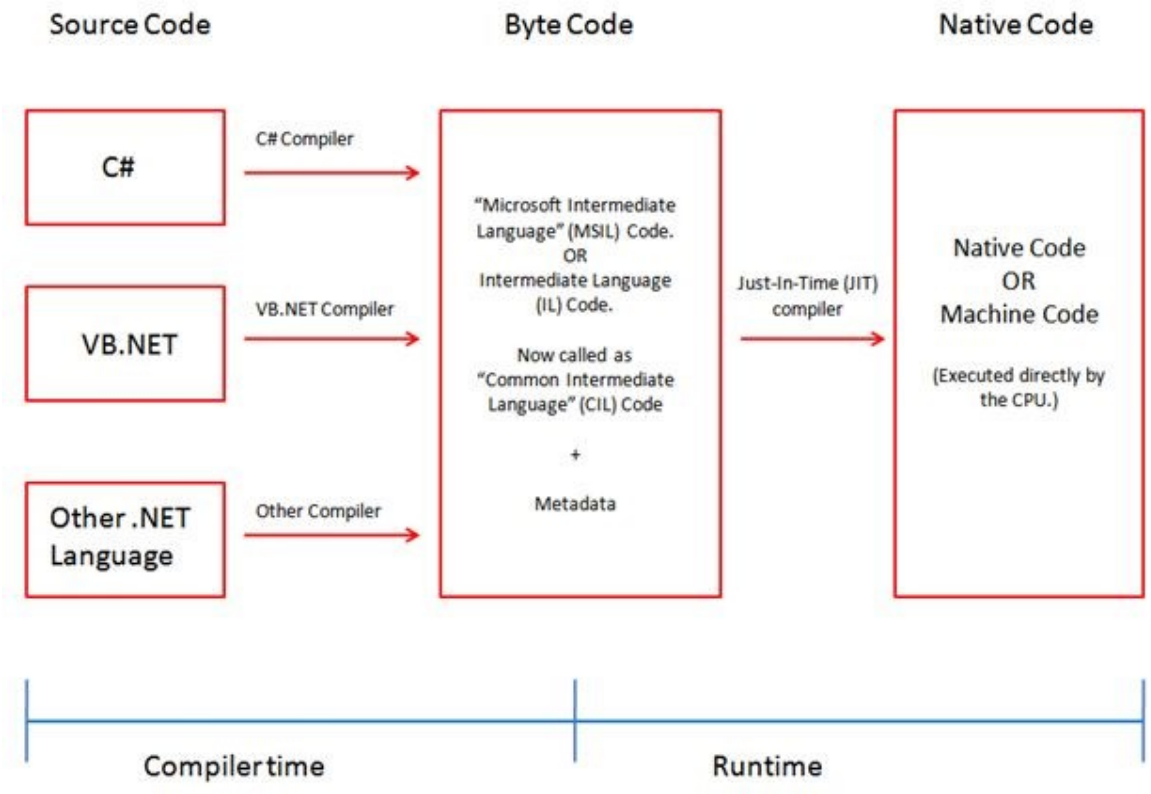
Praktická část této práce se pak skládá z návrhu aplikace pro monitorování kryptoměnových trhů a popisu postupu realizace vývoje této aplikace s využitím technologií a nástrojů popsaných již dříve v teoretické části práce. Speciální pozornost je věnována realizaci zabezpečení aplikace ve všech jejich částech. Především se jedná o zabezpečení komunikace mezi webovým klientem a serverem a bezpečné ukládání dat v databázi. Závěr praktické části pak sestává z demonstrace výstupní aplikace a jejích možností.

I. TEORETICKÁ ČÁST

1 TECHNOLOGIE PRO VÝVOJ WEBOVÝCH APLIKACÍ

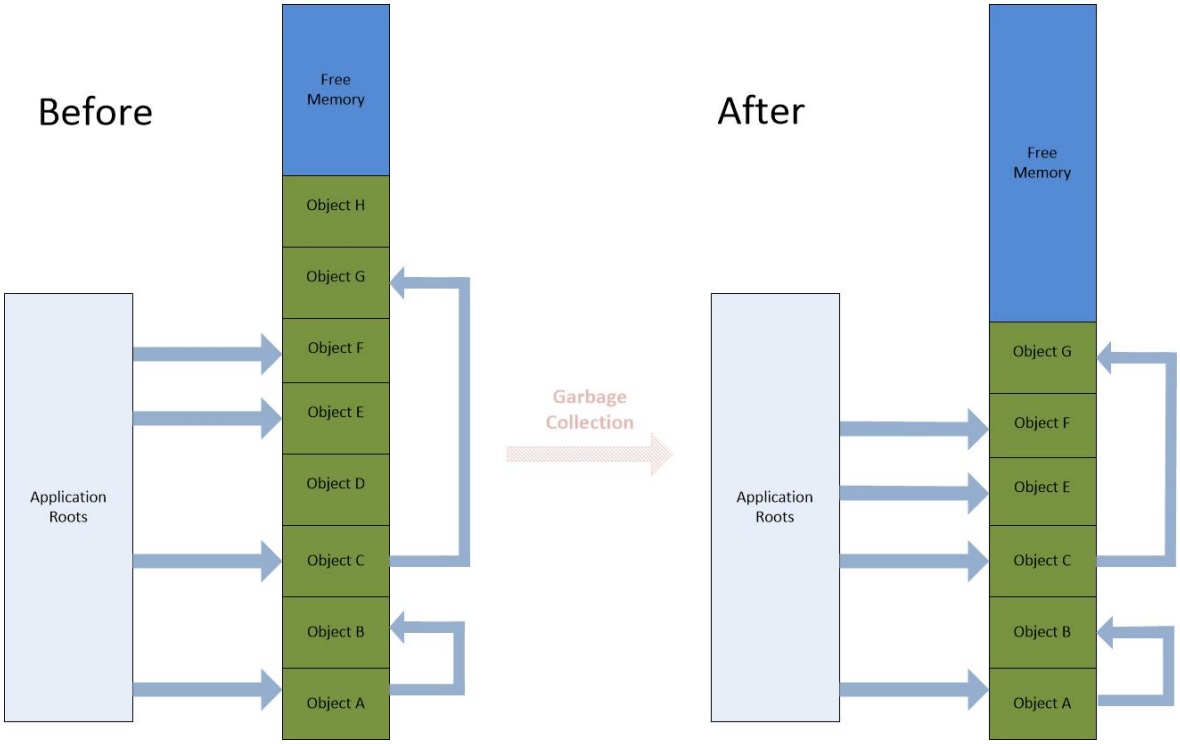
1.1 Framework .NET

.NET framework je platforma pro vývoj software vytvořená společností Microsoft. První verze byla publikována v roce 2002. V této době byl framework určen čistě pro vývoj nativních aplikací pro operační systém Microsoft Windows. Od té doby Microsoft neustále pracuje na nových verzích, které přinášejí stále další a další funkce. [1]



Obrázek 1: Schéma postupu kompilace kódu [2]

Jednou z hlavních vlastností frameworku .NET je podpora více programovacích jazyků. V současnosti se jedná především o jazyky C#, VB.NET a F#. Na obrázku 1 lze vidět postup převodu z některého ze zmíněných jazyků na strojový kód. Nezávisle na tom, v kterém programovacím jazyku programátor píše kód, dochází při procesu kompilace k převedení zápisu v konkrétním jazyce na zápis v CIL (Common Intermediate Language). Kód ve formátu CIL je potom s využitím CLR (Common Language Runtime) převeden na strojový kód, který již lze vykonávat na specifické hardwarové platformě. [1]



Obrázek 2: Zjednodušená vizualizace principu Garbage Collection v .NET [3]

Další důležitou vlastností je automatická správa paměti, čímž se programy napsané v některém z .NET kompatibilních jazyků liší od programů vytvořených například v jazyce C. Programátor tedy nemá pod kontrolou alokaci paměti, ale přenechává tuto starost takzvanému Garbage Collectoru. Ten v prostředí .NET zjednodušeně funguje tak, že s využitím samostatného vlákna odlišného od hlavního vlákna dané aplikace periodicky prochází všechny vytvořené objekty a kontroluje, jestli je již možné je smazat a uvolnit tak místo v paměti. Tento postup je znázorněn na obrázku 2. Objekty existující v paměti, které ještě není možné smazat, jsou postupně přesouvány mezi třemi generacemi. Tím se minimalizuje nutnost procházet pokaždé všechny objekty v paměti – tím, že dojde k rozdělení, je umožněno častěji kontrolovat objekty, u kterých se dá předpokládat, že jejich životnost nebude tak dlouhá, zatímco objekty s delší předpokládanou dobou životnosti lze odložit do některé z generací, kde ke kontrole nedochází tak často. Nejčastěji ke kontrole dochází v generaci 0, zatímco generace 2 obsahuje objekty s nejdelší předpokládanou dobou životnosti. Využití GC samozřejmě velmi urychluje vývoj aplikací a minimalizuje možnost vzniku úniků paměti. Na druhou stranu ovšem použití automatického GC nikdy nemůže být tak optimální, jako manuální alokace paměti a následné uvolnění. [1]

Tabulka 1: Přehled historických verzí původního frameworku .NET [4]

Rok	Verze .NET	Vylepšení a nové vlastnosti
2002	1.0	První vydání, managed code, CLR, použití .DLL jako knihoven tříd
2003	1.1	Podpora mobilních zařízení pro ASP.NET, ADO.NET pro databáze Oracle a OBDC, podpora IPv6
2005	2.0	CLR 2.0, vylepšení pro ASP.NET a ADO.NET, generické typy, parciální třídy, anonymní metody, nullable typy, podpora pro 64bitové aplikace
2006	3.0	Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF)
2007	3.5	Podpora AJAX, Language Integrated Query (LINQ), stromy výrazů, Hash sety, podpora P2P připojení, DateTimeOffset
2010	4.0	CLR 4.0, Task Parallel Library (TPL), Dynamic Language Runtime (DLR)
2012	4.5	Podpora async a await, rozšíření WPF, WCF a ASP.NET, podpora polí větších než 2 GB
2013	4.5.1	Vylepšení výkonu a možností debugování, podpora automatického přesměrování bindingů, rozšířená podpora pro Windows Store aplikace
2014	4.5.2	Rozšíření možností ASP.NET API, podpora systémového DPI pro Windows Forms, Event Tracing for Windows
2015	4.6	Nový 64bitový JIT kompilátor, vylepšení ASP.NET, optimalizace funkce Garbage Collectoru
2017	4.7	Podpora pro vysoké DPI ve Windows Forms, podpora dotykového ovládání ve WPF pro Windows 10, rozšíření kryptografických a bezpečnostních možností frameworku
2019	4.8	Vylepšení funkce JIT kompilátoru, aktualizace ZLib

V roce 2016 Microsoft začal paralelně s hlavní verzí .NET frameworku vyvíjet i takzvaný .NET Core. Jedná se o multiplatformní verzi klasického .NET frameworku, která kromě mnoha dalších vylepšení přináší především možnost spuštění aplikací používajících .NET Core jak na systémech s OS Windows, tak i na Mac OS X a na velké části Linuxových distribucí. V pozdějších verzích navíc přibyla i podpora pro procesorovou architekturu ARM – ze začátku pouze pro 32bitové ARM procesory, v pozdějších verzích pak i pro 64bitové procesory. Velkou výhodou oproti původnímu frameworku je také to, že celý projekt .NET Core je open source – zdrojové kódy jsou volně přístupné na platformě GitHub, kde se do vývoje mohou kromě zaměstnanců společnosti Microsoft zapojit i vývojáři z komunity. [1]

Tabulka 2: Přehled historických verzí .NET Core [5]

Rok	Verze .NET Core	Vylepšení a nové vlastnosti
2016	1.0	První vydání, multiplatformní podpora, open source, ASP.NET Core, Xamarin.Forms, UWP
2016	1.1	Podpora dalších distribucí Linuxu, vylepšený server Kestrel, podpora Microsoft Azure cloudu, Entity Framework Core
2017	2.0	.NET Standard 2.0, podpora dalších distribucí Linuxu
2018	2.1	Zlepšení výkonu, oprava chyb, podpora dalších distribucí Linuxu, včetně distribucí na platformě ARM32 a Alpine
2018	2.2	Podpora Windows na ARM32, Azure Active Directory
2019	3.0	Velké zvýšení výkonu, podpora WPF, WinForms a WCF (pouze pro Windows), snížení využití operační paměti, vylepšení Garbage Collectoru, oficiální podpora pro Raspberry Pi a ostatní ARM64 zařízení
2020	3.1	Oprava chyb a další vylepšení výkonu, dlouhodobá podpora po dobu tří let

V roce 2020 Microsoft oznámil vznik .NET 5, který se má stát hlavní implementací .NET a ukončit tak větvení na původní .NET framework a .NET Core. .NET 5 obsahuje naprostou většinu toho, co již uměly oba předchozí frameworky, s výjimkou technologií Web Forms (jako náhrada jsou doporučeny technologie Blazor či Razor Pages) a Windows Workflow (doporoučenou náhradou je open source řešení CoreWF). [6]

Obrázek 3: Přehled struktury komponent .NET 5 [6]

Jak lze vidět na obrázku 3, v oblasti vývoje nativních desktopových aplikací .NET 5 podporuje frameworky WPF, Windows Forms a UWP, pro vývoj webových aplikací je pak dostupný framework ASP.NET. Samozřejmostí je také možnost vývoje cloudových služeb pro Microsoft Azure a mobilních aplikací založených na frameworku Xamarin. Z oblasti herního vývoje je pak podporován velmi populární engine Unity, ve kterém probíhá veškeré skriptování herní logiky přímo v jazyce C#. Pro aplikace z oblasti Internet of Things (IoT) je stejně jako v .NET Core 3.0 dostupná podpora pro zařízení využívající procesory ARM32 a ARM64. Kromě toho pak .NET 5 obsahuje i technologie použitelné pro vývoj v oblasti umělé inteligence.

Kromě standardních vylepšení, jako je opětovné navýšení rychlosti a snížení nároků na využití operační paměti byla oznámena celá řada nových funkcí. Mezi nejvýznamnější patří především příslib interoperability s knihovnami napsanými v jazycích Java, Objective-C a Swift, a to na všech platformách, které tyto jazyky podporují. Rovněž byla oznámena chystaná podpora statické kompilace – v současné době .NET používá takzvanou kompilaci just-in-time (JIT), zatímco chystaná kompilace typu ahead-of-time (AOT) umožňuje vytvářet ještě výkonnější a kompaktnější kód, než tomu bylo doposud. [6]

1.2 Jazyk C#

C# je moderní, objektově orientovaný a typově bezpečný programovací jazyk vyvinutý společností Microsoft. Ve spojení s frameworkem .NET jej lze využít k vývoji mnoha typů aplikací. Syntaxe jazyka je silně podobná syntaxi programovacích jazyků C, C++ a Java. Na rozdíl od jazyků C a C++ se ovšem jedná o vysokoúrovňový programovací jazyk, čímž se více podobá již zmíněnému jazyku Java. První verze byla společností Microsoft uvolněna v roce 2002, od té doby stále probíhá vývoj – v roce 2021 je nejaktuálnější verzí C# 9.0. [7]

Tabulka 3: Přehled historických verzí jazyka C# [8]

Rok	Verze C#	Vylepšení a nové vlastnosti
2002	1.0	První vydání, základní vlastnosti
2005	2.0	Generické typy, parciální třídy, anonymní metody, iterátory, nullable typy, statické třídy, privátní settery
2007	3.0	Implicitně typované proměnné (klíčové slovo var), automatické properties, anonymní typy, extenzní metody, lambda výrazy, stromy výrazů, parciální metody
2010	4.0	Dynamický binding, volitelné a pojmenované argumenty metod
2012	5.0	Klíčová slova async a await pro práci s asynchronními metodami
2015	6.0	Klíčové slovo nameof, podpora použití await uvnitř bloku catch, interpolace stringů, filtr výjimek
2017	7.0	Out proměnné, třída Tuple, Pattern Matching, lokální funkce, možnost definování main metody jako async
2019	8.0	Readonly modifikátor, statické lokální funkce, nová struktura příkazu switch s využitím pattern matching, async streamy
2020	9.0	Typ Record, statické anonymní funkce, vylepšení pattern matchingu, top level statements, init-only setters

Mezi hlavní výhody použití jazyka C# pro vývoj aplikací bezesporu patří relativní rychlost vývoje oproti nízko úrovňovým jazykům, jako jsou například jazyky C nebo C++. Tím, že C# většinou poskytuje vyšší míru abstrakce, tak umožňuje vývojářům nezabývat se všemi konkrétními detaily dané implementace. Typickým příkladem je systém správy paměti – zatímco v jazyce C je nutno, aby alokaci a dealokaci paměti řešil přímo programátor, jazyk C# využívá automatický systém správy paměti skrze GC (jak již bylo zmíněno v kapitole věnující se frameworku .NET). [7]

S tím souvisí i další výhoda – v jazyce C# není potřeba při práci s referenčními typy používat ukazatele. V některých vysoce specializovaných případech je však práce s ukazateli stále optimálnější, či dokonce nutná – právě pro tyto případy C# práci s ukazateli stále podporuje, i když pro většinu aplikací vyvíjených v C# je tato funkcionalita již zbytečná. [9]

```
class Program
{
	static void Main(string[] args)
	{
		Console.WriteLine("Hello World!");
	}
}
```

Obrázek 4: Ukázka základní syntaxe v jazyku C#

Podobnost syntaxe s jazyky C a Java (viditelná na obrázku 4) lze rovněž považovat za výhodu – významně se tak snižuje čas, který programátor znalý některého z těchto jazyků potřebuje k pochopení alespoň základních programů napsaných v C#. Nicméně na tuto podobnost samozřejmě nelze spoléhat vždy – po mnoha letech vývoje do jazyka C# přibylo mnoho funkcí, které v ostatních jazycích vůbec nejsou, případně jsou implementovány zcela jiným způsobem. [9]

Mezi významné vlastnosti, které přibyly ve verzi jazyka C# 3.0, patří podpora pro technologii LINQ. Ta umožňuje dynamicky sestavovat dotazy, které jsou svou strukturou značně podobné dotazovacímu jazyku SQL. Výhodou oproti jazyku SQL ovšem je, že LINQ lze použít jak pro dotazování nad databází, tak nad strukturami existujícími v paměti – lze tak definovat široké množství libovolných dotazů nad poli, listy, slovníky a dalšími typy kolekcí, které jazyk C# podporuje. [9]

Verze jazyka 5.0 poté přišla s podporou klíčových slov async a await. C# už dříve umožňoval práci s asynchronními metodami, nicméně až s touto aktualizací se jejich implementace dala považovat za skutečně rychlou, snadnou a elegantní. V praxi použití

vypadá tak, že se modifikátorem async označí definice signatury metody, která má být vykonávána asynchronně (návratová hodnota takovéto metody by však měla vždy být typu Task). Při volání takovéto metody z jiné části kódu se pak před samotné zavolání vloží klíčové slovo await, které značí, že kód dané metody se nebude vykonávat dál, dokud nedojde k získání návratové hodnoty z metody volané pomocí await. Při tomto však nedochází k blokování celého vlákna, tak jak by tomu bylo u klasické synchronní metody. To je samozřejmě velmi praktické v případě vývoje aplikací, kde není žádoucí zablokovat celou aplikaci při čekání na vykonání nějaké metody. [4]

```
var orientation = direction switch
{
      Directions.Up => Orientation.North,
      Directions.Right => Orientation.East,
      Directions.Down => Orientation.South,
      Directions.Left => Orientation.West,
      _ => throw new NotImplementedException(),
};
```

Obrázek 5: Syntaxe příkazu switch v C# 8.0

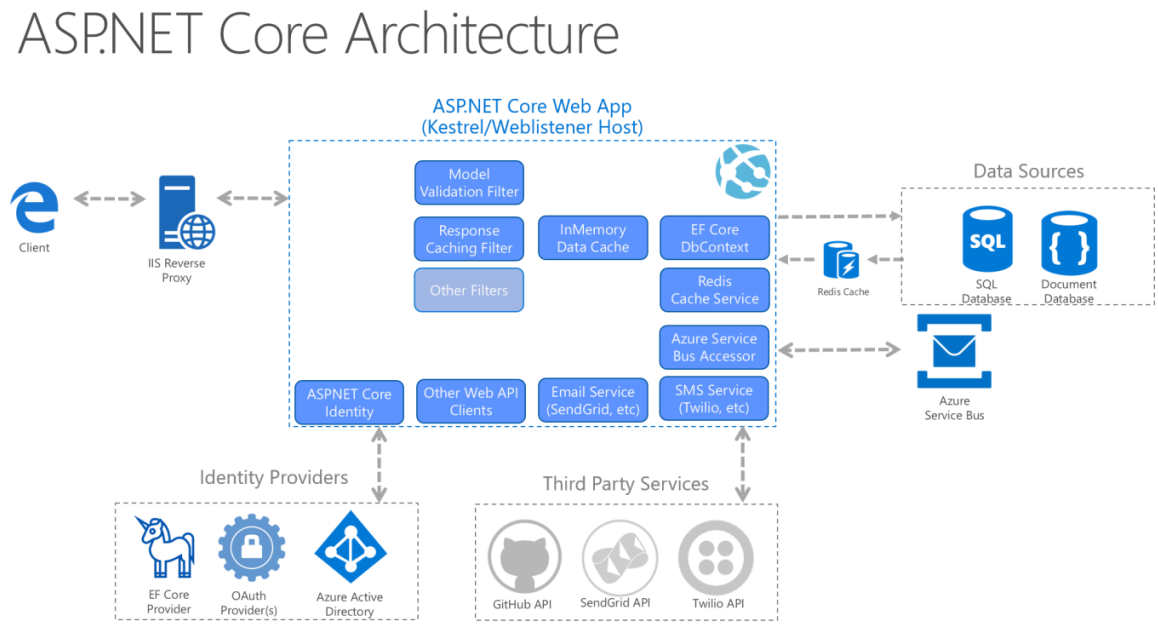
S každou verzí C# dochází k přidání nové funkcionality, v pozdějších verzích se pak jedná především o nové způsoby zápisu syntaxe, jejichž cílem je ušetřit práci programátorům. Typickým příkladem je nová syntaxe příkazu switch specifikovaná ve verzi jazyka 8.0, která využívá operátoru lambda (=>) a výrazů. Tento styl zápisu je zobrazen na obrázku 5. Takto zapsaný příkaz se samozřejmě chová zcela identicky jako klasický zápis, nicméně je v tomto případě potřeba napsat mnohem méně kódu než v případě použití klasického switche. [4]

Ačkoliv se v první řadě jedná o jazyk zaměřený na imperativní, objektově orientované programování, tak především v poslední době do jazyka pronikají vlastnosti typické pro funkcionální programovací jazyky. Jedná se o funkce jako je podpora anonymních delegátů, lambda výrazů a extenzních metod. Verze jazyka 7.0 poté přidala podporu datového typu tuple a porovnávání vzorů (pattern matching). Možnosti použití porovnávání vzorů byly v následujících vydaných verzích jazyka velmi rozšiřovány. [10]

1.3 Framework ASP.NET Core

ASP.NET Core je framework pro vývoj webových aplikací. Je součástí frameworku .NET (od verze .NET Core 1.0 až do současné verze .NET 5) a stejně jako on byl vyvinut společností Microsoft, nicméně se jedná o open source řešení, do kterého mohou prostřednictvím platformy GitHub přispívat i třetí strany. Jedná se o nástupce původní

technologie ASP.NET, od které se odlišuje několika klíčovými vlastnostmi. Jak již bylo zmíněno, celý projekt Core je open source. Stejně jako .NET Core a .NET 5, tak i ASP.NET Core je multiplatformní – webové služby vyvinuté na základu ASP.NET Core je tedy možno provozovat na velkém množství zařízení i operačních systémů – podporuje operační systémy Windows, Linux a Mac OS X, stejně jako procesorové architektury x86-64, ARM32 a ARM64. [11]



Obrázek 6: Architektura ASP.NET Core [12]

Výhodou oproti klasickému frameworku ASP.NET je, že celá architektura (znázorněna na obrázku 6) je nově plně modulární a distribuovaná pomocí NuGet balíčků. Praktický dopad je tedy takový, že vývojář do projektu nainstaluje pouze ty balíčky, které jsou skutečně potřebné – tím se podstatně snižuje jak celková velikost výsledné aplikace, tak i její nároky na využití paměti a procesoru. [11]

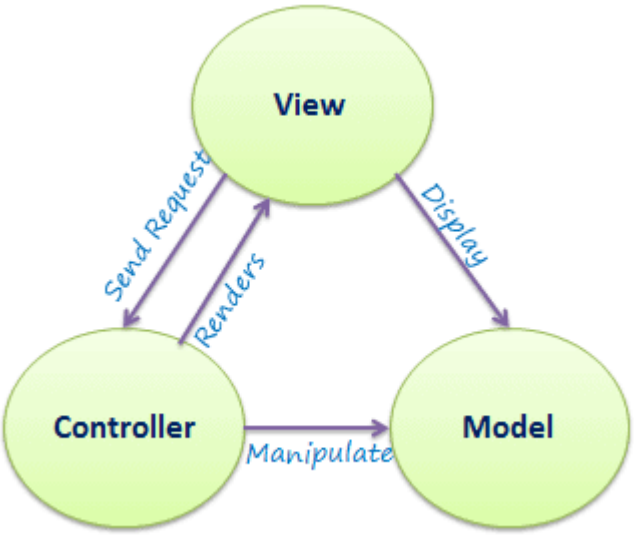
Mezi další výhody patří celková rychlost oproti ASP.NET – došlo k odstranění všech závislostí na starší knihovně System.Web.dll, které zbytečně zpomalovaly systém a omezovaly možnosti multiplatformního nasazení. ASP.NET Core také obsahuje zabudovaný IoC (Inversion of Control) kontejner, který umožňuje využití automatického vkládání závislostí. [11]

Webové služby vyvinuté s použitím ASP.NET Core lze hostovat mnoha způsoby – zůstává klasická možnost použít Windows a IIS, nicméně vzhledem k multiplatformní kompatibilitě celého řešení lze použít i webový server Apache, případně Kestrel (integrovaný webový

server, který je přímo součástí ASP.NET Core a lze jej použít na všech podporovaných operačních systémech). [11]

1.3.1 Framework ASP.NET Core MVC

MVC je softwarová architektura, která danou aplikaci rozděluje na tři samostatné podmnožiny – view, model a controller. Výhodou použití tohoto návrhového vzoru je jasné vymezení a rozdělení toho, čím se daná část aplikace má zabývat. Ačkoliv byl model MVC popsán již v roce 1970 pro využití v platformě Smalltalk, tak v posledních letech opět nabyl na popularitě a je použit v mnoha frameworcích, která se zabývají právě tvorbou webových aplikací – příkladem je jak ASP.NET Core MVC, tak například Spring Framework pro jazyk Java, či Ruby on Rails pro jazyk Ruby. [13]



Obrázek 7: Schéma návrhového vzoru MVC [13]

Na obrázku 7 lze vidět jednotlivé komponenty MVC včetně popisu jejich vzájemných interakcí. Model představuje data, se kterými daná aplikace pracuje. Při striktním dodržování zásad MVC by se model měl skutečně skládat pouze z dat aplikace, nicméně v kontextu ASP.NET Core se za součást modelu velmi často považuje i validační a business logika aplikace. Typicky je to i z důvodu omezení duplicity kódu – framework obsahuje mnoho anotačních atributů, které mohou sloužit současně pro validaci dat i definování struktury entit uložených v databázi. [13]

View by měl reprezentovat uživatelské rozhraní webové aplikace. Jeho odpovědností by tedy mělo být pouze přijímat příkazy uživatele a na základě těchto příkazů poté vykreslovat provedené změny. View by měl obsahovat co nejmenší množství jakékoliv logiky, kromě té, která souvisí čistě se vzhledem uživatelského rozhraní – příkladem může být například

zvýraznění některé části rozhraní po kliknutí myší. V prostředí ASP.NET bývá časté využití technologie Razor, která umožňuje generoval HTML soubory s využitím kódu psaného v jazyce C#, či v některém z dalších .NET kompatibilních jazyků. [1]

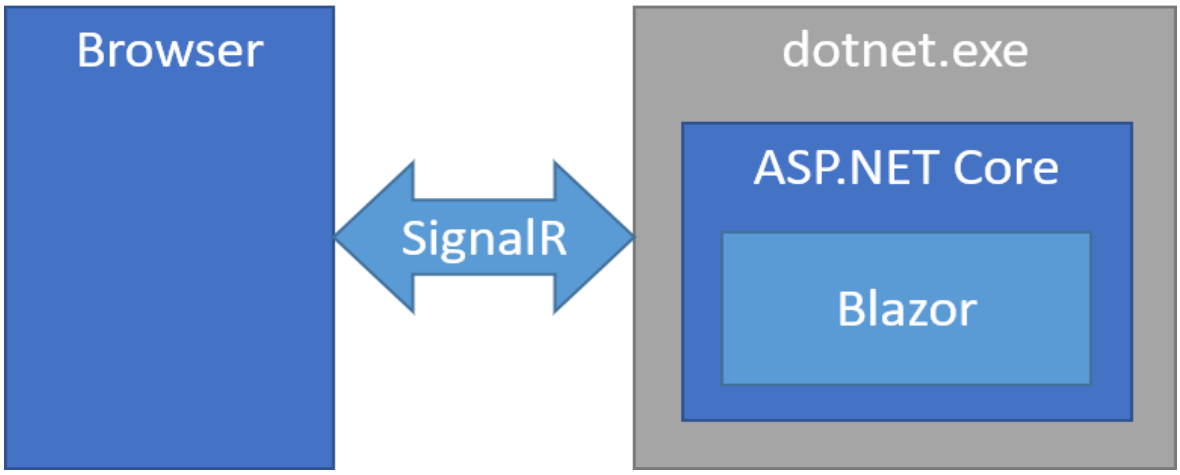
Controller ošetřuje interakci uživatelů se systémem, práci s modelem a výběr daného view, který má být zobrazen uživateli. Controller přebírá požadavky buď z daného view, nebo přímo přes API a na základě nich vykonává nějakou akci. Výsledek této akce je potom navrácen uživateli. Konkrétní implementace controllerů by však měla být poměrně minimální – například detaily business logiky by měla ošetřovat samostatná vrstva, na kterou se pak controllery mohou odkazovat. [1]

Kromě komplexních webových aplikací skládajících se ze všech tří vrstev MVC je ve frameworku ASP.NET Core možné vytvářet i takzvané Web API. Jedná se v podstatě o webové servery, které neobsahují komponentu view – neexistuje tak žádné grafické uživatelské rozhraní, ale pouze API, které poskytuje nějaká data. Pro výměnu dat se typicky používá HTTP komunikace ve spojení s některým z populárních formátů pro serializaci dat – v současné době se typicky jedná buď o XML nebo JSON. ASP.NET Core nicméně umožňuje vytváření vlastních formátů pro komunikaci. [1]

1.3.2 Framework ASP.NET Core Blazor

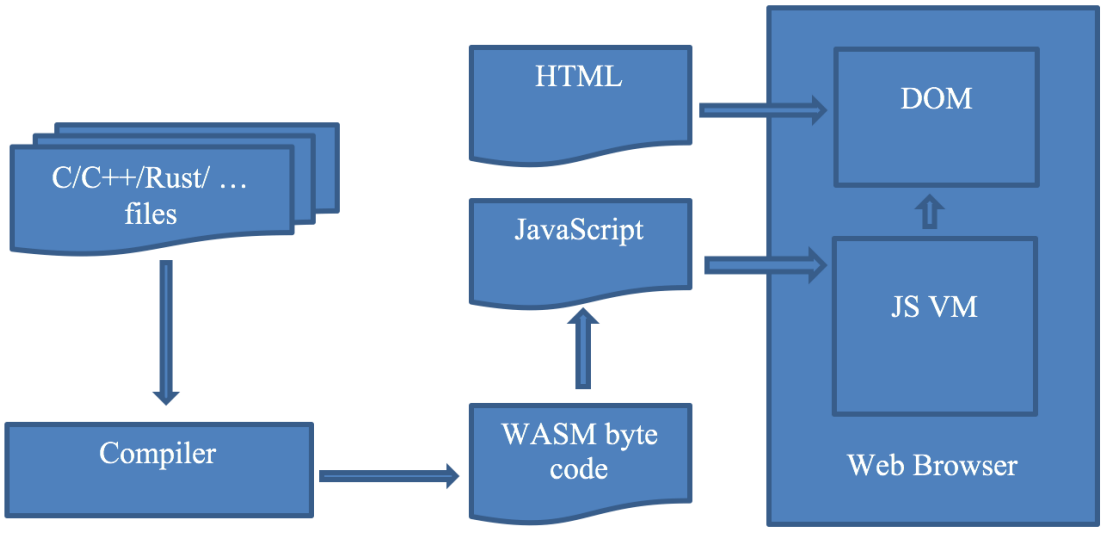
Blazor je webový framework, který je součástí komplexního frameworku ASP.NET Core. Je zaměřen na vývoj interaktivních klientských webových aplikací s využitím technologie .NET. Jednou z hlavních vlastností je možnost používání kódu napsaného v jazyce C# pro definování interakční logiky klientské části aplikace – místo typického kódu napsaného v JavaScriptu tedy lze použít C#. Současně je ale zachována i interoperabilita s JavaScriptem – z .NET metod používaných v Blazoru lze tedy volat již existující funkce napsané v JavaScriptu. .NET a C# jsou obecně považovány za výkonnější než JavaScript. V prostředí webového prohlížeče však samozřejmě .NET kód nemůže být tak efektivní, jako v jiných oblastech použití. [14]

Tím, že Blazor používá C# vzniká ještě jedna výhoda, a to že lze využívat téměř všechny již existující knihovny napsané v jazyce C#, kterých je obrovské množství. Současně lze také využívat ty samé knihovny v klientské i serverové části aplikace, čímž dochází k výrazné redukci duplicity kódu. [14]



Obrázek 8: Schéma Blazor Server [14]

V současné době jsou k dispozici dva typy Blazor aplikací. Prvním typem je takzvaný Blazor Server, který se vyznačuje tím, že veškerý kód napsaný v jazyce C# se vykonává na serveru samotném. Klientský webový prohlížeč pouze aktualizuje zobrazené uživatelské rozhraní s využitím technologie SignalR. Ta umožňuje otevřít spojení mezi klientem a serverem, přes které potom server posílá povely pro aktualizaci vizuálního rozhraní na základě kódu vykonaného na straně serveru, tak jak je to zobrazeno na obrázku 8. Nevýhodou tohoto řešení je samozřejmě vysoká zátěž serveru při připojení velkého množství klientů. [14]



Obrázek 9: Schéma standardu WebAssembly [15]

Druhým typem je pak Blazor WebAssembly. Jak je již z názvu patrné, využívá se zde standardu WebAssembly, který definuje přenositelný formát pro binární kód. Veškerý kód napsaný v jazyce C# se tedy vykonává přímo v prohlížeči na klientském počítači, stejně jako by tomu bylo při použití JavaScriptu, což podstatně snižuje zátěž serveru. Na obrázku 9 je

zobrazen postup odeslání kódu v některém z podporovaných jazyků do prohlížeče a jeho vykonání. Další výhodou je mnohem větší rychlost webových aplikací, jelikož není potřeba pro provedení změny na uživatelském rozhraní čekat na odpověď serveru. Nevýhodou tohoto přístupu může být, že před spuštěním webové aplikace v prohlížeči je potřeba celou její zkompilovanou podobu stáhnout ze serveru – to může být problém především u pomalých připojení a velkých aplikací. [16]

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Obrázek 10: Ukázka struktury jednoduché komponenty pro Blazor

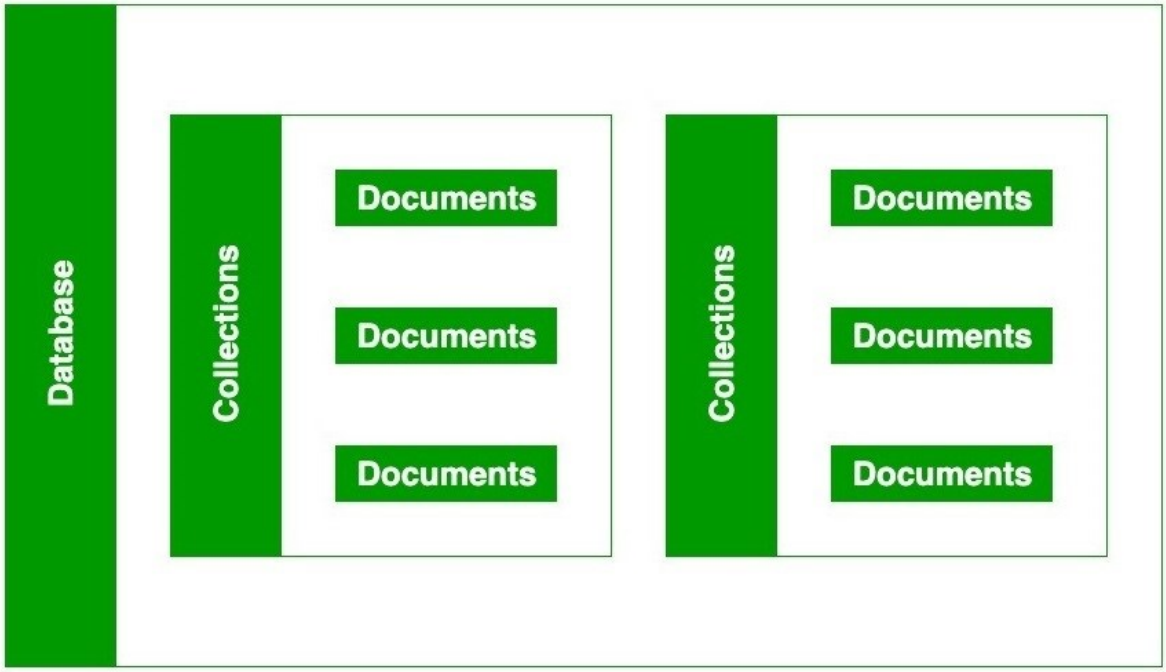
Aplikace používající Blazor jsou založené na jednotlivých komponentách. Na obrázku 10 lze vidět jednoduchou ukázku takovéto komponenty. Komponentou se v pojetí Blazoru rozumí element uživatelského rozhraní – to může být například stránka, dialogové okno nebo formulář pro zadávání dat. Komponenty jsou klasické .NET třídy, které definují logiku pro renderování grafického uživatelského rozhraní a logiku reakce na události. Stejně jako ASP.NET Core MVC, tak i Blazor z části využívá technologii Razor, která definuje syntaxi pro kombinování HTML kódu a kódu napsaného v jazyce C#. Na rozdíl od ASP.NET Core MVC ale Blazor skutečně definuje logiku klientské části aplikace, zatímco v pojetí MVC se jedná o model typu request – response. Komponenty lze do sebe zanořovat, či případně znovu použít v jiném místě aplikace. Již hotové komponenty lze také distribuovat prostřednictvím NuGet balíčků, tak jak je to zvykem u jiných .NET knihoven. [15]

V budoucí verzi (pravděpodobně se bude jednat o součást chystaného frameworku .NET 6) Microsoft plánuje rozšíření použitelnosti Blazoru i pro tvorbu nativních aplikací – mohlo by tak jít o budoucí náhradu populárních frameworků WPF a UWP, s tím rozdílem, že

v Blazoru by mělo být možné vytvářet aplikace s grafickým rozhraním i pro ostatní podporované operační systémy. [6]

1.4 Databázový systém MongoDB

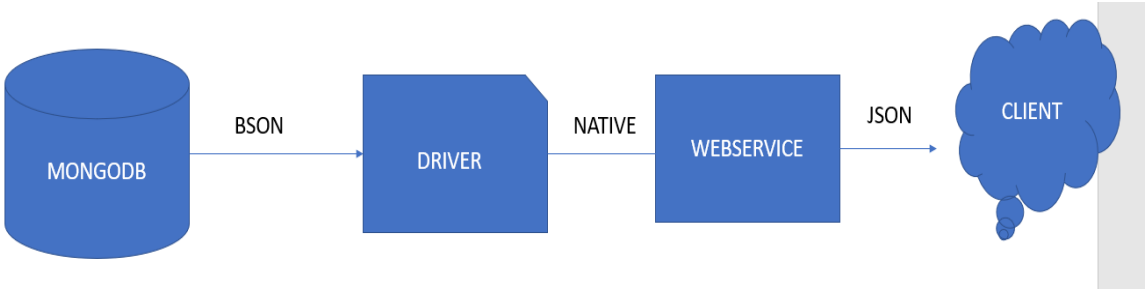
MongoDB je dokumentová databáze ze skupiny takzvaných NoSQL databází (databázové systémy, které nepoužívají dotazovací jazyk SQL a většinou nejsou relační). Databáze MongoDB vznikla v roce 2009 a v současné době je vyvíjena společností MongoDB Inc. V roce 2021 se jedná o jednu z velmi populárních alternativ k relačním SQL databázím – využívají ji například firmy Cisco, Coinbase, eBay nebo IBM. [17]



Obrázek 11: Základní struktura databáze v MongoDB [17]

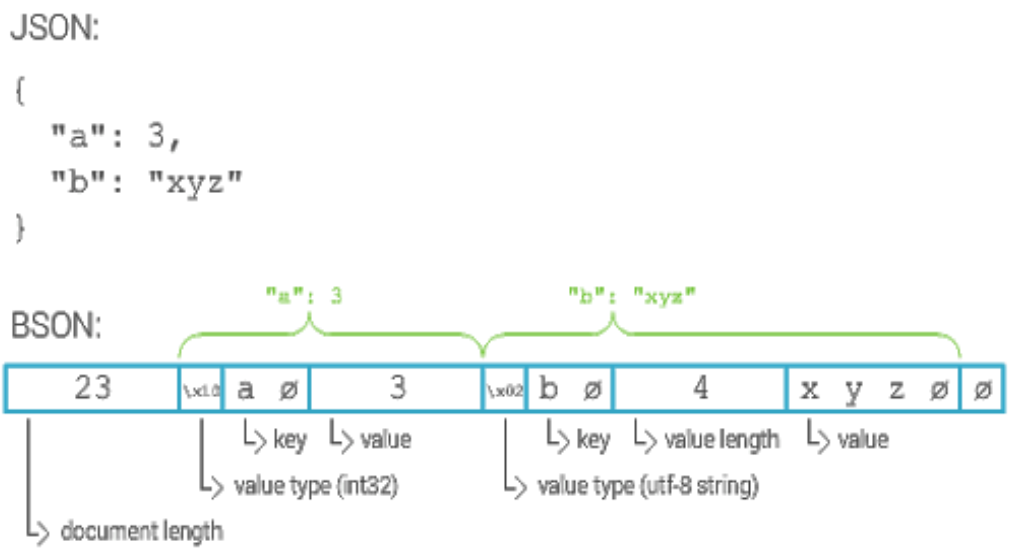
Základním principem fungování této databáze (a dokumentových databázových systémů obecně) je to, že místo klasických tabulek, které jsou používané v relačních databázích, jsou použity kolekce dokumentů. Struktura dokumentů není pevně daná, ale jedná se v podstatě o objekty typu klíč – hodnota. Jak lze vidět na obrázku 11, jedna kolekce může obsahovat velké množství strukturálně zcela odlišných dokumentů. To poskytuje velkou výhodu z hlediska vývojářské flexibility, neboť takový způsob ukládání dat mnohem více odpovídá tomu, jak se pracuje se třídami a objekty v moderních, objektově orientovaných jazycích – MongoDB mimo jiné podporuje dědičnost. Strukturu dokumentu navíc není potřeba předem definovat, tak jak je to typické u relačních modelů a tabulek, ale serializace objektů na dokumenty probíhá zcela dynamicky, což vede k velké úspoře vývojářského času. V rámci

jednoho dokumentu lze velmi snadno ukládat komplexní hierarchické struktury, vnořená pole a další komplexní typy objektů. [17]



Obrázek 12: Schéma použití MongoDB jako součást webové služby [18]

Pro ukládání dokumentů se používá formát BSON (Binary JSON). Jedná se o binární způsob zápisu objektů, které jsou svou strukturou víceméně identické jako populární formát pro serializaci objektů JSON (JavaScript Object Notation). V typickém použití databázového systému MongoDB jako úložiště pro webovou službu (znázorněno na obrázku 12) tedy dochází k tomu, že rozhraní webové služby přijímá data ve standardním formátu JSON, následně je zpracuje v souladu se svou vnitřní logikou a při uložení objektů do databáze pak dochází ke konverzi na formát BSON. To typicky zajišťuje takzvaný driver – knihovna napsaná v některém z kompatibilních jazyků, která zajišťuje efektivní komunikaci mezi webovou službou a databází samotnou. [18]



Obrázek 13: Porovnání formátu JSON a BSON [19]

Hlavní výhodou využití formátu BSON k ukládání je minimalizace velikosti takto uložených dat, což je navíc ještě umocněno tím, že v pozdějších verzích MongoDB využívá

automatickou kompresi u často se opakujících dokumentů se stejnou strukturou. Formát je rovněž optimalizovaný i pro rychlé provádění databázových skenů – ve velmi specifických případech to ovšem může vést k tomu, že data zapsaná ve formátu BSON jsou o něco větší než původní JSON podoba, neboť BSON obsahuje i metadata o obsažených datech, která slouží právě ke zrychlení databázových skenů. Kompletní schéma struktury BSON objektu lze vidět na obrázku 13. [18]

Tabulka 4: Přehled historických verzí MongoDB [20]

Rok	Verze MongoDB	Vylepšení a nové vlastnosti
2009	1.0	První vydání
2009	1.2	Zrychlení tvorby indexů, ukládání JavaScriptových funkcí, možnost vytvoření více indexů pro jednu kolekci
2010	1.4	Možnost vytvářet indexy na pozadí, optimalizace správy paměti, vylepšení detekce regulárních výrazů
2010	1.6	Sharding, replica sety, podpora IPv6
2011	1.8	Journaling, sparse indexy
2011	2.0	Příkaz compact, změna defaultní velikosti zásobníku, polygonové vyhledávání, multi-location dokumenty
2012	2.2	Framework pro agregace, TTL kolekce
2013	2.4	Hashované indexy, textové vyhledávání, vylepšení zabezpečení, vylepšená podpora geospaciálních dat
2014	2.6	Nový protokol pro zapisovací operace, zlepšení agregačních funkcí a textového vyhledávání, optimalizace dotazů
2015	3.0	WiredTiger engine pro ukládání, SCRAM-SHA-1 autentizace, MongoDB Ops Manager, zlepšení výstupu funkce explain

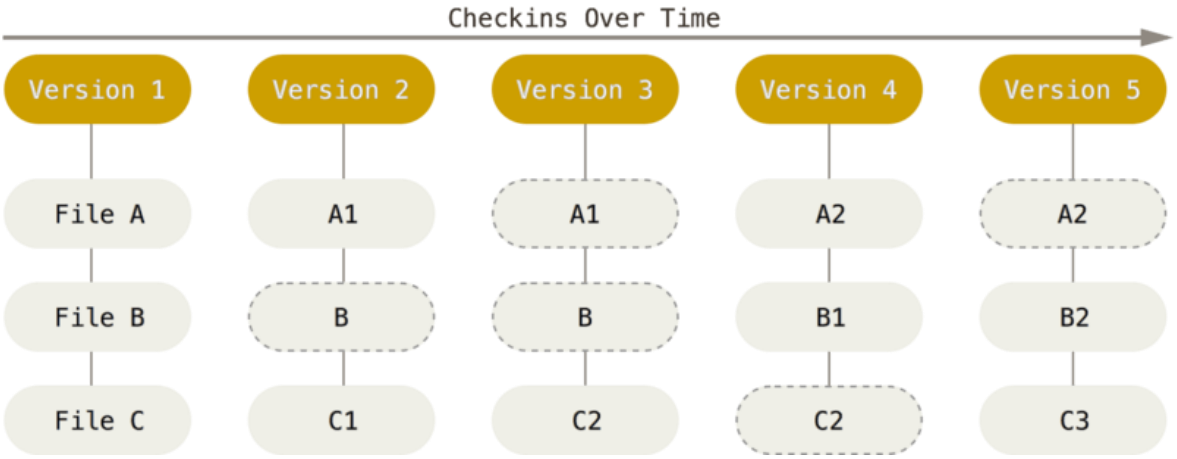
2015	3.2	WiredTiger engine jako defaultní ukládací engine, zlepšení funkce replikace
2016	3.4	Pohledy, kolace,
2017	3.6	Sharded clusters, MongoDB Compass, komprese OP_MSG
2018	4.0	Transakce složené z více dokumentů, nové typy agregací, autentizace SCRAM-SHA-256, odstranění MMAPv1
2019	4.2	Distribuované transakce, wildcard indexy, oficiální podpora pro Linux na ARM64
2020	4.4	Zajištěné čtení, definice sdílených klíčů, zrcadlené čtení, detailnější statistiky indexů

MongoDB server je dostupný v několika edicích, přičemž komunitní edice je poskytována zcela bez poplatků i pro využití v komerčních projektech. Jedná se o open source projekt, jehož zdrojové kódy jsou dostupné na platformě GitHub. Oficiálně jsou podporovány systémy Microsoft Windows, Linux a Mac OS X, pro některé distribuce Linuxu jsou pak podporovány i procesory architektury ARM64. Vzhledem k tomu, že zdrojový kód serveru je volně dostupný, tak je možné provést vlastní kompilaci pro cílovou platformu, pokud není oficiálně podporována. [17]

1.5 Verzovací systém Git

Git je softwarový systém pro správu verzí, který se nejčastěji používá pro koordinovanou spolupráci více vývojářů v rámci vývoje softwarových produktů. Byl vytvořen v roce 2005 autorem operačního systému Linux Linusem Torvaldsem, a to za účelem verzování zdrojového kódu Linuxového kernelu. Git je zcela open source a je možné jej bezplatně používat – je distribuován pod licencí GNU GPL 2. [21]

Git se od ostatních známých systémů pro verzování liší především svým přístupem k datům, která spravuje. Pro ostatní systémy je typické, že data ukládají jako seznam změn provedených na jednotlivých hlídaných souborech. Tento systém se typicky označuje jako delta-based version control. [21]



Obrázek 14: Systém ukládání dat v Gitu [21]

Oproti tomu Git s daty pracuje jako s kolekcí snapshotů souborového systému, jak lze vidět na obrázku 14. Při každém uložení pak dochází k tomu, že Git uloží snapshot stavu všech hlídaných souborů – aby však nebylo zbytečně plýtváno, tak v případě, že se některý ze souborů nijak nezměnil od posledního stavu, Git pro daný soubor uloží pouze odkaz na předchozí stav, který již je uložený v některém z minulých snapshotů. Ve výsledku lze tedy data uložená v Gitu považovat za seznam postupně prováděných snapshotů, z nichž každý vždy popisuje stav celého souborového systému. [21]

Jednou z výhod Gitu oproti mnoha ostatním systémům pro verzování je to, že jej lze používat i zcela lokálně, bez nutnosti připojení na vzdálený server. Celá historie daného repositáře je vždy uložena přímo v lokální databázi, není tedy potřeba při procházení změn stahovat nějaká další data. V případě, že daný vývojář nemá momentálně přístup k síti, tak stále může pracovat téměř bez omezení, stejně jako by pracoval, když k síti připojený je. Po připojení pak může všechny změny nahrát na vzdálený server. [21]

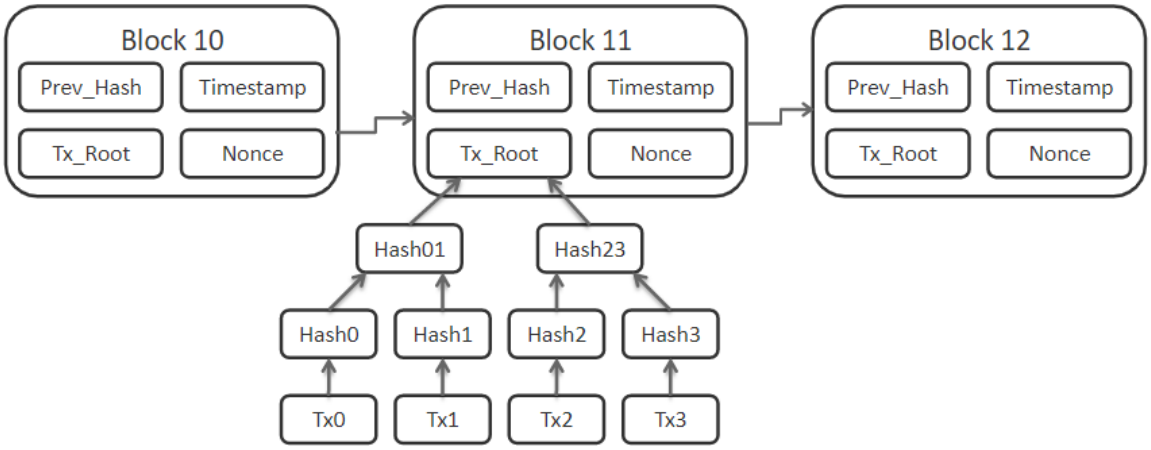
Pro veškerá data uložená v Gitu se používají kontrolní součty, což znamená, že není možné provést jakoukoliv změnu v obsahu souborů bez toho, aby systém zaregistroval, že došlo ke změně. Pro provádění kontrolních součtů se využívá hashovací algoritmus SHA-1, který vytváří hashe dlouhé 40 znaků pro libovolně veliký soubor. [21]

Soubory v Gitu mohou nabývat celkem tří hlavních stavů. Prvním stavem je stav modified – zde se jedná o soubory, které byly nějakým způsobem změněny, ale nebylo ještě provedeno jejich uložení (commit) do lokální databáze Gitu. Druhým možným stavem je stav staged – jedná se o způsob označení modifikovaného souboru, který lze použít pro jeho oddělení od ostatních modifikovaných souborů. Třetím stavem je pak stav committed – jedná se o soubory, pro které již byly zaznamenané změny uloženy do databáze Gitu. [21]

2 ZÁKLADNÍ PRINCIPY KRYPTOMĚN

Jelikož se práce zabývá vývojem aplikace použitelné pro monitorování kryptoměnových trhů, je vhodné definovat, co to vlastně kryptoměny jsou. Jako kryptoměny jsou označovány digitální měny založené na principech kryptografie. Tyto kryptografické principy typicky zaručují základní funkcionalitu pro provádění transakcí v dané měně, kontrolu nad vytvářením nových jednotek měny a verifikaci vlastnictví určitého množství kryptoměny. Většina dnes známých kryptoměn nemá žádnou fyzickou podobu – jedná se pouze o digitální data. V současné době je většina kryptoměn decentralizovaná, což znamená, že neexistuje žádná centrální instituce, která by měla kontrolu nad měnou, tak jako je tomu u klasických měn, kde v roli hlavní autority vystupuje většinou centrální banka. První plně decentralizovanou kryptoměnou na světě se v roce 2009 stala měna s názvem Bitcoin, kterou vytvořil anonymní vývojář vystupující pod jménem Satoshi Nakamoto. [22]

2.1 Blockchain



Obrázek 15: Blockchain kryptoměny Bitcoin [23]

Blockchain je seznam záznamů (v kontextu blockchainu nazývané jako bloky), které jsou vzájemně zřetězeny, tak jak je to znázorněno na obrázku 15. Každý z bloků obsahuje unikátní kryptografický hash, který se odkazuje na předchozí blok v tomto řetězu. Kromě toho pak ještě obsahuje aktuální časové razítko a data o provedených transakcích. Tato unikátní struktura zajišťuje, že není možné změnit jeden jediný blok v celém řetězu bez toho, aby došlo k narušení celé struktury blockchainu a tím pádem i k odhalení této změny. Připojení nového bloku do blockchainu je typicky možné jen pokud dojde k takzvanému konsensu – většina sítě dané kryptoměny se shodne, že údaje obsažené v novém bloku jsou platné. Při vytvoření nového bloku obdrží všichni členové sítě novou verzi blockchainu

s připojeným novým blokem. Dochází pak k porovnávání s původní verzí blockchainu – pokud jsou nalezeny nějaké další změny kromě nového bloku, tak nemůže být nová verze blockchainu přijata. [24]

2.2 Decentralizace

Jak již bylo zmíněno, většina kryptoměn se snaží být plně decentralizovaná – neexistuje tedy žádná centrální autorita, která by měla pravomoc řídit měnový kurz, emitaci nových jednotek měny či jakkoliv jinak ovlivňovat směřování dané kryptoměny. Hodnota dané měny je tedy vždy určena pouze poptávkou a nabídkou. Integrita kryptoměny musí vždy být zajištěna její architekturou – je potřeba, aby použité algoritmy zajišťovaly, že veškeré provedené transakce jsou vždy korektní a bezpečné. Stejně je tomu u procesu emitace nových jednotek měny – vzhledem k centralizaci musí být algoritmy zajištěno, že si každý uživatel nemůže vytvořit libovolné množství dané měny, ale musí být aplikována jistá pravidla, která platí pro všechny. [24]

2.3 Generace kryptoměn

Kryptoměny se obecně dají rozdělit do několika takzvaných generací, v závislosti na tom, jaké možnosti jejich technologie poskytuje a jaké si kladou cíle. První takzvanou generací, kam spadá například dnes obecně nejznámější kryptoměna Bitcoin, je skupina kryptoměn, jejichž hlavním cílem je vytvořit pouhou funkční alternativu k standardním měnám. Tyto kryptoměny lze tedy využít především pro výměnu za nějaké zboží, služby či jinou měnu, případně jako investiční nástroj sloužící k navýšení či alespoň uchování hodnoty, jako je tomu například u zlata. [25]

Jako druhá generace se pak typicky označují kryptoměny, které poskytují veškerou funkcionalitu kryptoměn první generace, ale současně s tím se snaží nabídnout i něco navíc. Typickým a nejznámějším zástupcem je kryptoměna Ethereum, která jako jedna z prvních přišla s konceptem chytrých kontraktů. Blockchain dané kryptoměny tak lze využít k uzavření kontraktu mezi dvěma stranami. Jako třetí strana, která dohlíží na dodržení kontraktu, pak vystupuje pouze algoritmus dané kryptoměny, který při naplnění stanovených podmínek zajistí naplnění kontraktu. [25]

Třetí generace kryptoměn se pak zaměřuje především na řešení problému škálovatelnosti. V současné době je síť Bitcoinu téměř neustále přetížena, a v důsledku toho jsou tak transakce pomalé, případně jsou účtovány obrovské poplatky za provedení jakékoliv

transakce. V současné době existuje mnoho kryptoměn, které tento problém nějakým způsobem řeší, zůstává však otázkou, které z těchto řešení nakonec převládne. Příkladem takovéto měny může být například kryptoměna Cardano, která kromě toho, že poskytuje vlastnosti první generace (možnost obchodování, uchování hodnoty) a druhé generace (chytré kontrakty), tak nabízí i řešení problému škálovatelnosti. [25]

2.4 Proof of Work a Proof of Stake

Jak již bylo zmíněno v kapitole zabývající se blockchainem, pro rozšiřování blockchainu o nový blok je důležitý koncept konsensu. U tradičních kryptoměn založených na myšlence Proof of Work (důkaz práce) typicky dochází k tomu, co se označuje za těžbu kryptoměn – všechny počítače připojené do kryptoměnové sítě neustále provádějí kryptografické operace, dokud jeden z nich nenajde platný hash. Za nalezení platného hashe je typicky vítězi připsáno jisté množství dané kryptoměny. Ostatní účastníci těžby žádnou odměnu nezískají, ale celý proces se následně opakuje znovu a znovu, čímž získávají další šanci. Toto sebou nese několik negativ – většina kryptoměn je postavena tak, že čím více jednotek měny již bylo vytěženo, tím větší výpočetní výkon je potřeba k validaci nového bloku. Neúměrně tak narůstají požadavky na výpočetní výkon celé sítě – to může způsobovat celou řadu problémů v mnoha sektorech, ať už se jedná o nedostupnost grafických karet, obrovské zatížení elektrické sítě nebo dopad na životní prostředí. Dalším negativem pak je, že v případě, že by se někomu povedlo získat pod svou kontrolu více než 51 % výpočetního výkonu sítě, byl by schopný provést útok na síť, který by nenávratně poškodil validitu celého blockchainu. [26]

Koncept Proof of Stake navrhuje řešení těchto nedostatků. Těžaři měny jsou v tomto případě nahrazeni takzvanými validátory. Ti fungují na principu toho, že musí vlastnit určité množství dané měny, které potom uzamknou. Toto vlastnictví jim potom na základě konkrétního algoritmu měny dává právo být při rozšiřování blockchainu o nové bloky odměněni. Detaily daného odměňovací algoritmu se liší, v současné době existuje relativně velké množství kryptoměn, které používají nějakou formu Proof of Stake. Největší nevýhodou je pak to, že existuje možnost útoku na síť v případě, že by útočník vlastnil více než 51 % celkového množství měny. S jistotou lze ovšem tvrdit, že získat takové množství některé měny by pro útočníka mělo být tak nákladné, že by při provedení útoku a následné ztrátě důvěry v měnu poškodil především sám sebe. [26]

2.5 Možnosti obchodování kryptoměn

Stejně jako je klasické měny možné obchodovat s využitím mezinárodního obchodního systému Forex, tak i kryptoměny lze obchodovat s využitím takzvaných cryptocurrency exchanges, což jsou v podstatě burzy pro kryptoměny. Typicky taková platforma svým uživatelům poskytuje možnost obchodovat vzájemně kryptoměny v pevně stanovených párech – například pár s označením BTC/ETH umožňuje nakupovat kryptoměnu Ethereum za Bitcoin, případně obráceně. Většina velkých kryptoměnových burz poskytuje svým zákazníkům možnost nakupovat kryptoměny i za klasické měny, a to v závislosti na tom, v jakých zemích daná společnost operuje – typická je možnost nákupu a prodeje za americké dolary (takový pár se pak označuje například jako BTC/USD) a eura, některé burzy ovšem podporují i jiné lokální měny. U centralizovaných systémů většinou dochází k tomu, že kryptoměna, kterou daný uživatel nakoupí, je uložena v peněžence, kterou má ve vlastnictví burza – to může být v jistých případech riskantní, jelikož pak uživatel nemá plný přístup ke své kryptoměně a spoléhá se na to, že se společnost provozující systém zachová vždy korektně. Většina burz však poskytuje uživatelům možnost kdykoliv si převést nakoupené kryptoměny do své vlastní peněženky. To je sice výhodné z pohledu toho, že uživatel tak získává absolutní kontrolu nad svými prostředky uloženými v dané kryptoměně, nicméně pak musí důkladně dbát na zabezpečení takto uložených prostředků. [27]

Vzhledem k tomu, že jedním ze základních cílů velké části kryptoměn je dosažení absolutní decentralizace, tak se centralizované burzy setkávají s častou kritikou. V současné době již existuje celá řada plně decentralizovaných kryptoměnových burz využívajících P2P sítě k umožnění obchodování. Ačkoliv je decentralizace v jistých směrech výhodou, může být i značnou nevýhodou – absolutní absence jakékoliv regulace může nahrávat vzniku podvodů. Rovněž je na takovýchto burzách velmi problematické dosáhnout výměny kryptoměn za klasické měny (ačkoliv to není zcela nemožné) – v současné době musí většina centralizovaných burz splňovat zákony země, kde sídlí, aby jim bylo umožněno nakládat s fiat měnami. Jelikož u decentralizovaných burz neexistuje žádná oficiální entita vystupující v roli zprostředkovatele obchodu, je pro ně rovněž nemožné splnit zákonné podmínky ve většině zemí. [27]

II. PRAKTICKÁ ČÁST

3 NÁVRH APLIKACE

3.1 Požadavky na aplikaci

Požadavky na výslednou aplikaci lze pro přehlednost rozdělit na požadavky technické, které se týkají použitých technologií a vývojových metod, požadavky funkční, které popisují, jakou funkcionalitu musí aplikace poskytovat uživatelům a požadavky na zabezpečení aplikace.

3.1.1 Technické požadavky

Technické požadavky pro aplikaci jsou následující:

- Aplikace je vyvinuta na nejnovější verzi frameworku .NET (v současné době .NET 5)
- Pro klientskou část aplikace je použita technologie Blazor WebAssembly
- Serverová i klientská část aplikace jsou multiplatformní
- Při vývoji aplikace je použit verzovací systém Git
- Zdrojový kód aplikace je dostupný skrze platformu GitHub
- Zdrojový kód aplikace je psán v souladu s obecnými pravidly pro použité programovací jazyky a softwarová architektura je navržena s využitím doporučovaných návrhových vzorů

3.1.2 Funkční požadavky

Funkční požadavky pro aplikaci jsou následující:

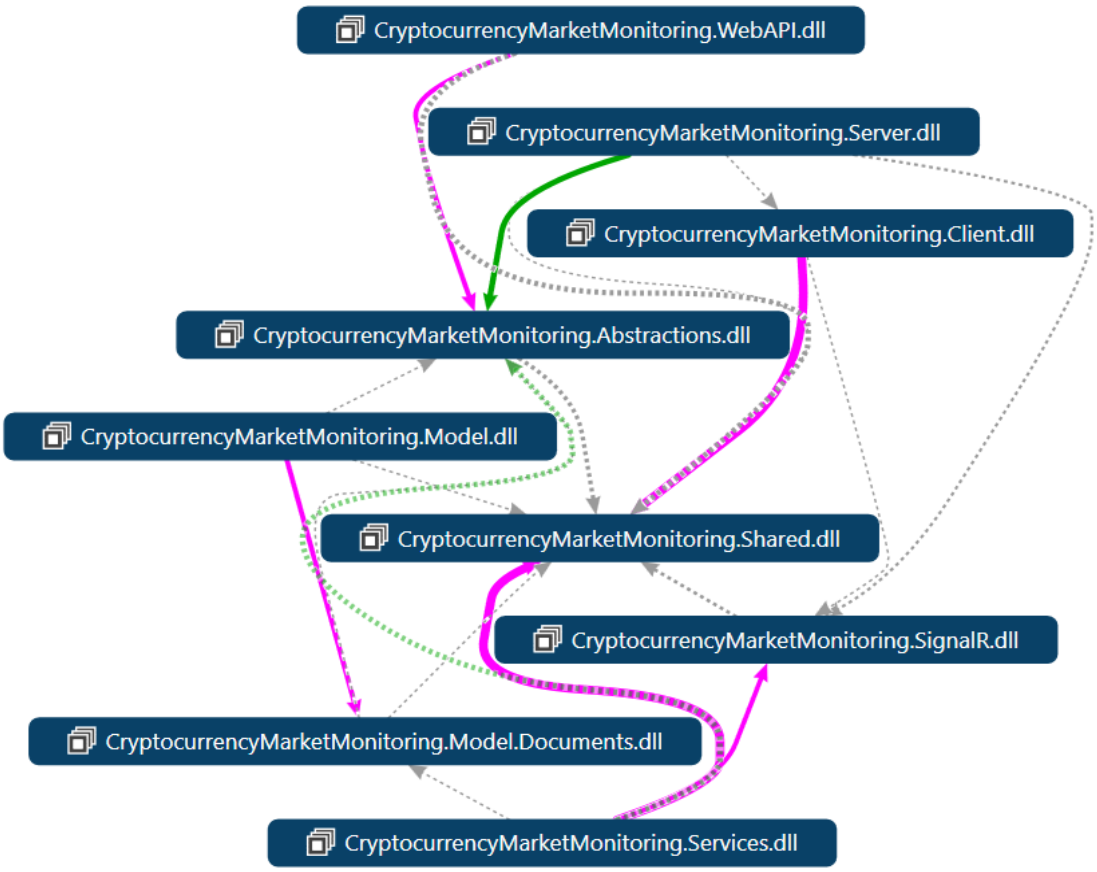
- Aplikace umožňuje monitorování aktuální ceny vybraných kryptoměn
- Aplikace umožňuje zobrazení grafu vývoje ceny kryptoměn, včetně historických dat
- Aplikace umožňuje registraci a přihlášení uživatelů
- Aplikace umožňuje u vybraných kryptoměn zobrazit doplňkové informace poskytované burzou
- Aplikace poskytuje přehledné uživatelské rozhraní
- Aplikace umožňuje uživatelům vytvořit si seznam kryptoměn a zobrazit jejich aktuální hodnoty v jednom pohledu

- Grafové komponenty použité v aplikace jsou interaktivní – je možné filtrovat v nich podle času, případně modifikovat zobrazená data
- Graf poskytuje uživatelům možnost vykreslit indikátory pro technickou analýzu, případně zobrazit křivku trendového vývoje
- Grafy s vývojem ceny kryptoměn je možné exportovat v některém z obecně použitelných formátů, například PDF, JPG, PNG

3.1.3 Požadavky na zabezpečení

- Veškeré uživatelské údaje jsou uloženy v bezpečné formě (především hesla musí být hashovaná a databáze obsahující hashe hesel musí být dostatečně zabezpečena)
- Aplikace používá šifrovanou komunikaci všude tam, kde je to možné, s využitím protokolů HTTPS a WSS
- Autentizace a autorizace uživatelů v aplikaci je provedena formou standardizovaného a bezpečného řešení
- Aplikace umožňuje registrovaným uživatelům zabezpečit svůj účet s využitím vícefázové autentizace
- Aplikace je vytvořena v souladu s obecně platnými požadavky na zabezpečení moderních webových aplikací

3.2 Struktura kódu aplikace



Obrázek 16: Graf referencí jednotlivých projektů v aplikaci

Pro přehlednost je celý kód aplikace rozdělen do několika samostatných projektů, kdy každý z nich řeší nějakou dílčí část logiky celé výsledné aplikace. Projekty se na sebe podle potřeby vzájemně odkazují pomocí takzvaných referencí, jak lze vidět na obrázku 16. Graf referencí je v přehlednější podobě dostupný také jako příloha P I.

CryptocurrencyMarketMonitoring.Abstractions

V tomto projektu se nacházejí především rozhraní (interface) tříd, které se v rámci serverové části aplikace používají s využitím Dependency Injection, případně některé další abstraktní třídy používané napříč celou aplikací.

CryptocurrencyMarketMonitoring.Client

Tento projekt obsahuje klientskou část aplikace, která je založená na technologii Blazor WebAssembly. V základu se tedy jedná o konzolovou aplikaci pro framework .NET 5, která obsahuje jednotlivé Blazor komponenty. Tyto komponenty se skládají ze souborů obsahujících HTML, C# a CSS.

CryptocurrencyMarketMonitoring.Server

Jedná se o hlavní projekt serverové části aplikaci, který obsahuje přímo spustitelnou „konzolovou“ aplikaci, která následně zajišťuje spuštění a hostování hlavní služby. Rovněž je zde řešeno veškeré základní nastavení týkající se použitého frameworku ASP.NET Core.

CryptocurrencyMarketMonitoring.Shared

Výhodou toho, že jak klientská, tak serverová část aplikace je psána v jazyce C#, je to, že je díky tomu možné sdílet velké množství kódu. To se může hodit například u definice tříd používaných pro přenos objektů přes API při komunikaci mezi serverem a klientem – definovat strukturu objektů pak stačí pouze jednou.

CryptocurrencyMarketMonitoring.WebAPI

Tento projekt obsahuje veškeré controllery, které poskytují API serverové aplikace. API je založené na standardu REST a využívá tedy metody HTTP GET, POST, PUT a DELETE, v kombinaci s přenosem dat ve formátu JSON.

CryptocurrencyMarketMonitoring.Services

Obsahem tohoto projektu jsou implementace jednotlivých služeb, uvnitř kterých je definovaná business logika celé aplikace. Pokud tedy klient používá API, tak jednotlivé controllery přijmou požadavek, ale zpracování a další interní logika je vykonávána právě ve vrstvě Services.

CryptocurrencyMarketMonitoring.Model

Tato část obsahuje vše, co se týká práce s databází systému – jedná se především o implementaci rozhraní mezi C# kódem a databázovým systémem MongoDB, s využitím nativního driveru, který je distribuován formou NuGet balíčku.

CryptocurrencyMarketMonitoring.Model.Documents

Tento projekt obsahuje definice datových tříd, které slouží jako reprezentace dokumentů, které lze následně ukládat do databázového systému MongoDB.

CryptocurrencyMarketMonitoring.SignalR

SignalR je technologie společnosti Microsoft postavená na bázi technologie WebSocket, která umožňuje efektivní obousměrnou komunikaci mezi klientem a serverem v reálném čase. Součástí tohoto projektu je definice jednotlivých hubů, které popisují metody, pomocí kterých server přijímá a odesílá data tímto kanálem.

4 REALIZACE APLIKACE

4.1 Základní architektura serverové části

Serverová část aplikace je založena na technologii ASP.NET Core 5.0 – jedná se tedy o webovou službu hostovanou s využitím integrovaného serveru Kestrel. Vstupním bodem aplikace je třída Program, konkrétně pak její metoda Main, tak jak je tomu obecně zvykem.

```
public class Program
{
   public static async Task Main(string[] args)
   {
      await CreateHostBuilder(args).Build().RunAsync();
   }

   public static IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
          .UseServiceProviderFactory(new AutofacServiceProviderFactory())
          .ConfigureWebHostDefaults(webBuilder =>
          {
             webBuilder.UseStartup<Startup>();
           });
}
```

Obrázek 17: Kód ve třídě Program

Jednou z vlastností novějších verzí frameworku .NET je podpora pro asynchronní Main metodu – jak lze vidět na obrázku 17, signatura metody Main obsahuje slova async a Task. Díky tomu je v těle metody možné volat asynchronně implementované metody s využitím operátoru await. V tomto konkrétním případě se jedná o metodu RunAsync, která inicializuje spuštění webového serveru.

```
public class Startup : DIContainerConfigurator
{
   public Startup(IHostEnvironment env) : base(env)
   {
      var builder = new ConfigurationBuilder()
       .SetBasePath(env.ContentRootPath)
       .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
       .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
       .AddEnvironmentVariables();

       Configuration = builder.Build();
   }
```

Obrázek 18: Konstruktor třídy Startup

Třída Program se dále pomocí metody UseStartup odkazuje na třídu Startup. Jedná se o standardní způsob konfigurace ASP.NET serveru – tato třída může obsahovat celou řadu metod, jejichž úkolem je nastavit různé parametry týkající se fungování serverové služby.

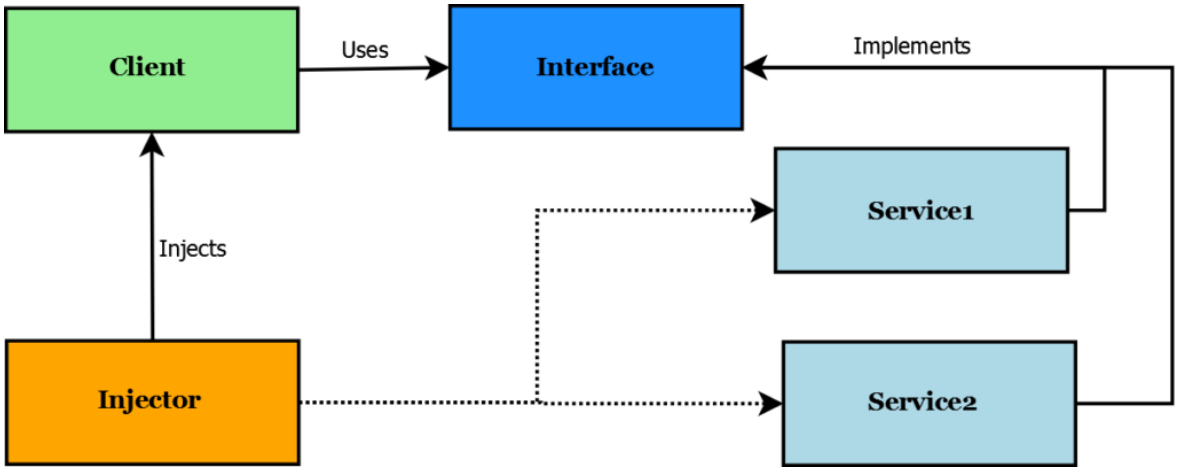
```
public override void ConfigureServices(IServiceCollection services)
{

   services.AddControllersWithViews().AddJsonOptions(opts =>
   {
      opts.JsonSerializerOptions.PropertyNamingPolicy = null);
   });
   services.AddRazorPages();
   services.AddSignalR().AddJsonProtocol(options =>
   {
      options.PayloadSerializerOptions.PropertyNamingPolicy = null;
   });


  services.AddControllerModules(Configuration, ContentRootPath);
  services.ConfigureBinanceClientSettings();

  services.AddResponseCompression(opts =>
  {
     opts.MimeTypes = ResponseCompressionDefaults.MimeTypes
     .Concat(new[] { "application/octet-stream" });
  });
```

Obrázek 19: Konfigurační metoda ConfigureServices ve třídě Startup

Na obrázku 19 lze vidět metodu ConfigureServices, která patří rovněž pod třídu Startup, jejímž obsahem je nastavení jednotlivých částí webového serveru. Právě v této metodě tak dochází k například zavedení API controllerů, stránek s využitím syntaxe Razor nebo technologie SignalR, která s využitím websocketů umožňuje real-time komunikaci mezi klientem a serverem v obousměrném komunikačním kanálu.

4.1.1 Inversion of Control Container

Jako Dependency Injection (DI) se označuje programovací technika, při které je hlavním cílem minimalizovat vazby mezi jednotlivými třídami. Prakticky to většinou znamená, že pokud daná třída využívá ve své logice nějaké další třídy, tak by neměla být zodpovědná za vytváření instancí těchto tříd, ale měla by očekávat, že jí budou přiděleny – typicky se tak děje předáním těchto instancí jako parametrů v konstruktoru.



Obrázek 20: Obecné schéma Dependency Injection [28]

Typicky se rovněž místo odkazování na konkrétní třídu používají rozhraní (interface), která popisují chování dané třídy (jaké jsou veřejně dostupné metody a vlastnosti) bez toho, aby existovala jakákoliv návaznost na detaily konkrétní implementace. Tento způsob využití rozhraní lze vidět na obrázku 20.

S tímto úzce souvisí i princip Inversion of Control (IoC). Při použití tohoto návrhové vzoru dochází k invertování toku řízení programu. Programátor tak nedefinuje, ve kterém bodě programu dojde k vytvoření instance konkrétní třídy, ale předává kontrolu jiné komponentě, jejímž úkolem je automaticky vytvářet instance tříd a předávat je tam, kde jsou potřeba.

Ve frameworku .NET existuje celá řada populárních knihoven a frameworků, které řeší problematiku DI a IoC. V novějších verzích frameworku dokonce existuje zabudovaná podpora DI – pro velkou část projektů je toto řešení zcela dostačující, nicméně ještě nenabízí všechny funkce, které poskytují některé z dedikovaných knihoven. Pro účely tohoto projektu je použita knihovna Autofac – jedná se o jeden z velmi populárních IoC kontejnerů pro framework .NET.

```
public class AutofacModule : Module
{
   protected override void Load(ContainerBuilder builder)
   {
      builder.RegisterType<CryptocurrencyOverviewService>()
             .As<ICryptocurrencyOverviewService>();

      builder.RegisterType<CryptocurrencyOverviewManager>()
             .As<ICryptocurrencyOverviewManager>()
             .As<IHostedService>().SingleInstance();

      builder.RegisterType<CoinGeckoClient>().As<ICoinGeckoClient>();
      builder.RegisterType<ChartDataService>().As<IChartDataService>();
      builder.RegisterType<UserService>().As<IUserService>();

      builder.RegisterType<BinanceClient>()
             .As<IBinanceClient>()
             .SingleInstance();
   }
}
```

Obrázek 21: Ukázka registrace komponent s využitím AutofacModule

K registraci jednotlivých komponent lze použít třídu Module. S využitím přetížené metody Load lze definovat, které třídy implementují které rozhraní. To je v podstatě vše, co je v tomto bodě potřeba nastavit – Autofac na základě nastavení uvnitř těchto modulů sestaví vše potřebné. Kromě standardních situací, kdy jedna třída odpovídá vždy jednomu rozhraní, lze nakonfigurovat i mnoho dalších rozdílných situací – pokud například jedno generické rozhraní implementuje více tříd, je možné provést mapování podle klíče. Stejně tak je možné definovat, jestli má při každém požadavku o získání instance být vytvořena instance nová, nebo má být napříč celou aplikací sdílena pouze jedna instance.

```
public UserController(IUserService userService)
{
   _userService = userService;
}

[HttpPost("login")]
public IActionResult Login(LoginDto login)
{
   var response = _userService.Login(login);
   return Ok(response);
}
```

Obrázek 22: Ukázka automatického doplnění instance třídy do konstruktoru

Na obrázku 22 lze vidět příklad automatického doplnění instance třídy implementující požadované rozhraní. Kdykoliv tedy dojde k vytvoření nové instance třídy UserController, tak Autofac automaticky vyhodnotí, že rozhraní IUserService je implementováno třídou UserService, vytvoří příslušnou instanci a tu předá do konstruktoru. Třída UserController poté může používat metody z rozhraní IUserService – například metodu Login.

```
public static void RegisterAutofacModules(this ContainerBuilder builder,
                                           IConfigurationRoot configuration,
                                           string contentRootPath)
{
   var assemblies = new List<Assembly>();

   var entryAssembly = Assembly.GetEntryAssembly();
   assemblies.Add(entryAssembly);
   configuration.GetSection("Modules:Assembly")
      .GetChildren()
      .ToList()
      .ForEach(section =>
      {
         var path = Path.Combine(contentRootPath, $"{section.Value}");
         if (File.Exists(path))
         {
            var assembly = AssemblyLoadContext.Default
                              .LoadFromAssemblyPath(path);

            assemblies.Add(assembly);
          }
      });
      builder.RegisterAssemblyModules(assemblies.ToArray());
}
```

Obrázek 23: Základní konfigurace načítání assembly modulů pro Autofac

Každý z projektů může mít svou vlastní třídu AutofacModule, ve které dochází k registraci komponent, jejichž implementace se nachází uvnitř daného projektu. Aby však ve výsledku Autofac mohl fungovat, je potřeba zavést moduly ze všech projektů – to se děje opět ve třídě Startup, konkrétně v metodě RegisterAutofacModules, která na základě konfiguračního souboru prochází všechny assembly moduly a provádí jejich registraci.

4.2 Komunikace přes SignalR

SignalR je součást frameworku .NET, která umožňuje oboustrannou komunikaci mezi klientem a serverem v reálném čase. Jedná se v podstatě o knihovnu, která pouze obaluje základní funkcionalitu technologie WebSocket takovým způsobem, aby ji šlo pohodlným

způsobem používat v prostředí aplikací napsaných pro framework .NET. Tuto technologii je výhodné používat zejména tam, kde by klasická komunikaci s využitím REST API nedávala smysl – jedná se typicky o webové aplikace, u kterých je potřeba zasílat data s vysokou frekvencí – příkladem tematicky se týkajícím této práce může být například potřeba zobrazovat aktuální cenu některé kryptoměny s tím, že aktualizovaná data přicházejí každou sekundu. Výhodou použití SignalR jakožto nadstavby nad WebSockety může být především jednodušší implementace a podpora automatické serializace a deserializace objektů na JSON a zpět při přenosu. Další výhodou pak je, že všechny huby mohou být na serverové straně řešení silně typované, čímž se omezí počet „magických" textových konstant definovaných uvnitř kódu.

```
public class OverviewUpdateHub : Hub<IOverviewUpdateClient>
{
   public async Task SendUpdateAsync(IEnumerable<OverviewUpdateDto> updates)
   {
      await Clients.Others.ReceiveUpdate(updates);
   }
}
```

Obrázek 24: Ukázka základního SignalR hubu

Základním stavebním kamenem knihovny SignalR jsou takzvané buby. Jedná se o třídy, ve kterých jsou definovány všechny metody, pomocí nichž lze zprostředkovávat komunikaci mezi klientem a serverem. Na obrázku lze vidět ukázku třídy OverviewUpdateHub, pomocí které lze odesílat aktualizace stavů směrem ke klientovi.

```
app.UseEndpoints(endpoints =>
{
   endpoints.MapHub<OverviewUpdateHub>("/overview-update");
});
```

Obrázek 25: Ukázka zavedení hubu ve třídě Startup

Všechny takto vytvořené huby je potřeba zavést v metodě Configure třídy Startup, stejně jako je tomu například u API controllerů. K tomu slouží extenzní metoda MapHub, která je přímou součástí frameworku. To je vše, co je potřeba provést na straně serveru – data lze pak kdykoliv odesílat nebo naopak přijímat s využitím nadefinovaných metod. Ke všem takto registrovaným hubům je kdykoliv možné přistupovat prostřednictvím DI s využitím rozhraní IHubContext.

```
_updateHubConnection = new HubConnectionBuilder().AddJsonProtocol(options =>
{
   options.PayloadSerializerOptions.PropertyNamingPolicy = null;
}).WithUrl(NavigationManager.ToAbsoluteUri("/overview-update")).Build();

_updateHubConnection.On<IEnumerable<OverviewUpdateDto>>("ReceiveUpdate",
(updates) =>
{
   DoGridUpdate(updates);
});

await _updateHubConnection.StartAsync();
```

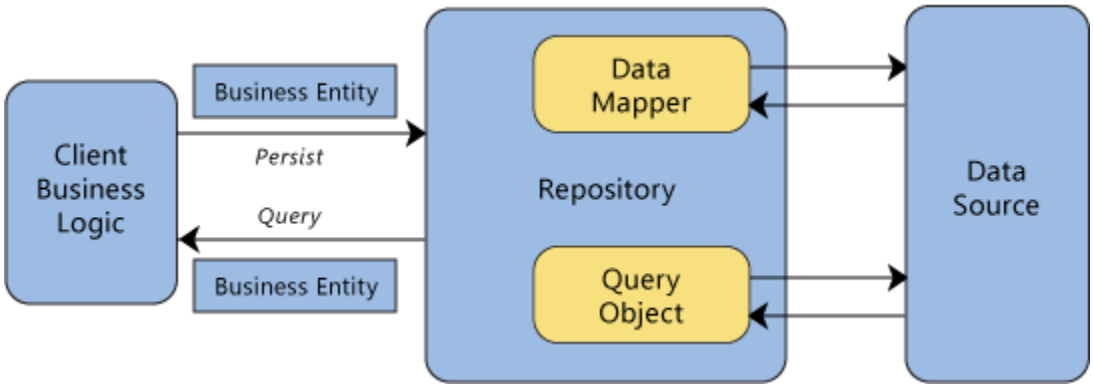
Obrázek 26: Ukázka inicializace připojení skrze hub na klientské straně

Na klientské straně lze pro inicializaci připojení použít systémovou třídu HubConnectionBuilder, která slouží jako továrna pro instance tříd typu HubConnection. Na takto získané instanci lze potom nadefinovat, která akce se má vyvolat, pokud dojde k přijmutí dat přes tento hub. Na závěr je ještě potřeba spustit metodu StartAsync, která inicializuje spojení. HubConnection potom vyčkává, než začnou přicházet nějaká data.

4.3 Databázová vrstva

Nejnižší vrstva komunikace s databází, která mimo jiné zajišťuje mapování z dokumentů na C# třídy a možnost přistupovat k databázi bez použití nativního dotazovací jazyka, je implementována skrze již zmíněný nativní C# driver pro MongoDB. I tak je nicméně vhodné použít další vrstvu abstrakce v rámci zajištění systémového přístupu k databázi napříč celou aplikací.

4.3.1 Repozitář



Obrázek 27: Schéma generického repozitáře [29]

Právě za tímto účelem je vhodné použít návrhový vzor repozitář, jehož smyslem je zapouzdření logiky pro přístup k nějakému zdroji dat tak, aby nezávislé na implementačních detailech nižší vrstvy bylo vždy možné používat obecné metody, které definuje právě repozitář. Umístění repozitáře v rámci architektury lze vidět na obrázku 27.

```
interface IMongoRepository<TDocument> where TDocument : IMongoDocumentBase
{
   IMongoCollection<TDocument> Collection { get; }
   FilterDefinitionBuilder<TDocument> Filter { get; }
   IndexKeysDefinitionBuilder<TDocument> Index { get; }
   ProjectionDefinitionBuilder<TDocument> Project { get; }
   UpdateDefinitionBuilder<TDocument> Updater { get; }

   bool Any(Expression<Func<TDocument, bool>> filter);
   Task<bool> AnyAsync();
   void Delete(Expression<Func<TDocument, bool>> filter);
   void Delete(TDocument entity);
   void Delete(ObjectId id);
```

Obrázek 28: Ukázka kódu rozhraní pro generický repozitář

V repozitáři je možné definovat širokou škálu metod pro všechny podporované databázové operace, a to jak v synchronním, tak asynchronním režimu. Práce s takovýmto repozitářem pak může připomínat používání extenzních metod z technologie LINQ při práci s libovolným typem kolekce.

4.3.2 Definice databázových dokumentů

```
public interface IMongoDocumentBase
{
   [BsonId]
   public ObjectId Id { get; set; }
}
```

Obrázek 29: Ukázka kódu bázové třídy pro dokument v MongoDB

Pro definici tříd, které odpovídají dokumentům, které jsou uloženy v databázi Mongo, je vhodné definovat si bázovou třídu, ze které pak všechny ostatní mohou dědit. V případě této aplikace se jedná o třídu MongoDocumentBase s rozhraním IMongoDocumentBase, které je zase naopak vhodné použít při definici generiky repozitáře. Jako základní databázové id Mongo používá hodnoty typu ObjectId. Jedná se o unikátní číslo složené z 12 bytů, kdy první 4 byty udávají časové razítko vzniku objektu, dalších 5 bytů je náhodných a poslední 3 byty vyjadřují číslo automaticky inkrementované databází. Jako unikátní databázové id záznamu lze použít i odlišné datové typy – například integer, nicméně obecně je

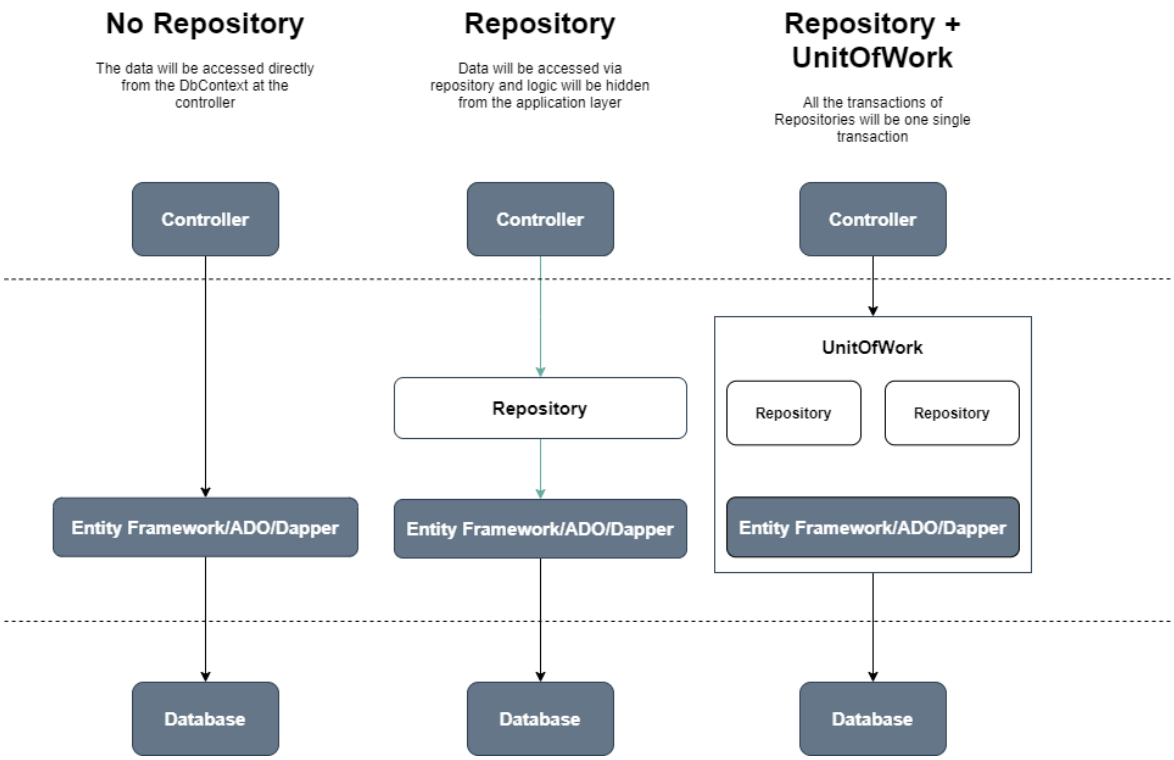
doporučováno používat právě typ ObjectId. V některých případech to ovšem může mít opodstatnění – velikost 12 bytů pro každé použité id může být příliš velká, pokud se jedná o kolekci, kam se ukládají relativně malé dokumenty ve velkém počtu.

```
public class User : MongoDocumentBase
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Username { get; set; }
    public string Email { get; set; }
    public string PasswordHash { get; set; }
}
```

Obrázek 30: Konkrétní třída odvozená z bázové třídy MongoDocumentBase

Z bázové třídy lze vytvářet odvozené třídy – příkladem může být na obrázku znázorněná třída User, která reprezentuje datový model pro ukládání uživatelů do databáze. Každý uživatel tak kromě specifických údajů, jako je jméno či email, bude mít ještě vlastnost Id, kterou zdědil právě z třídy MongoDocumentBase.

4.3.3 Unit of Work



Obrázek 31: Schéma využití návrhového vzoru Unit of Work a repozitáře [30]

Návrhový vzor Unit of Work se u SQL databází typicky využívá pro sjednocení více databázových operací v rámci jedné transakce – jak lze vidět na schématu na obrázku 31,

typická situace může být taková, že jedna instance Unit of Work pracuje se dvěma repozitáři, kdy každý z nich modifikuje v databázi jinou entitu. Smyslem třídy Unit of Work je potom uložit všechny provedené změny nad libovolným počtem entit v rámci jedné transakce.

Databázový systém MongoDB samozřejmě nepracuje s entitami a tabulkami, tak jako je tomu u SQL databází s využitím Entity Frameworku. Použití vzoru Unit of Work tak nemusí zprvu dávat příliš smysl, i tak je nicméně možné jej aplikovat – mimo jiné lze díky tomu docílit toho, že v případě výměny databázového systému za jiný nebude potřeba přepisovat vyšší logiku aplikace, ale jen implementaci, která se nachází pod úrovní Unit of Work. V novějších verzích databázového systému MongoDB jsou pak navíc podporovány i transakční operace, což tento návrhový vzor opět činí relevantním.

```
public abstract class UnitOfWorkMongoBase : IDisposable
{
   public UnitOfWorkMongoBase(ILoggerFactory loggerFactory,
                              IMongoRepositoryLocator locator)
   {
      _logger = loggerFactory.CreateLogger<UnitOfWorkMongoBase>();
       Locator = locator;
   }

   protected IMongoRepositoryLocator Locator { get; private set; }
```

Obrázek 32: Konstruktor třídy UnitOfWorkMongoBase

V rámci toho, aby se v jednotlivých konkrétních implementacích pro dané entity neopakoval neustále identický kód, je vhodné vytvořit bázovou abstraktní třídu, ze které pak mohou konkrétní implementace vycházet. To v tomto případě obstarává třída UnitOfWorkMongoBase, která obsahuje generické metody, pomocí kterých je možné provádět všechny standardní databázové CRUD (create, read, update and delete – tedy vytváření nových záznamů, čtení, aktualizace existujících záznamů a mazání). Tyto metody jsou však napsány relativně obecně tak, aby se daly použít pro široké množství specifických databázových operací – čtení například může být prováděno jako vyhledání konkrétního záznamu podle id, nebo vyhledávání množiny záznamů podle zadaných filtrů. Unit of Work se tedy vždy odkazuje na repozitář, který poskytuje celou sadu těchto specifických metod.

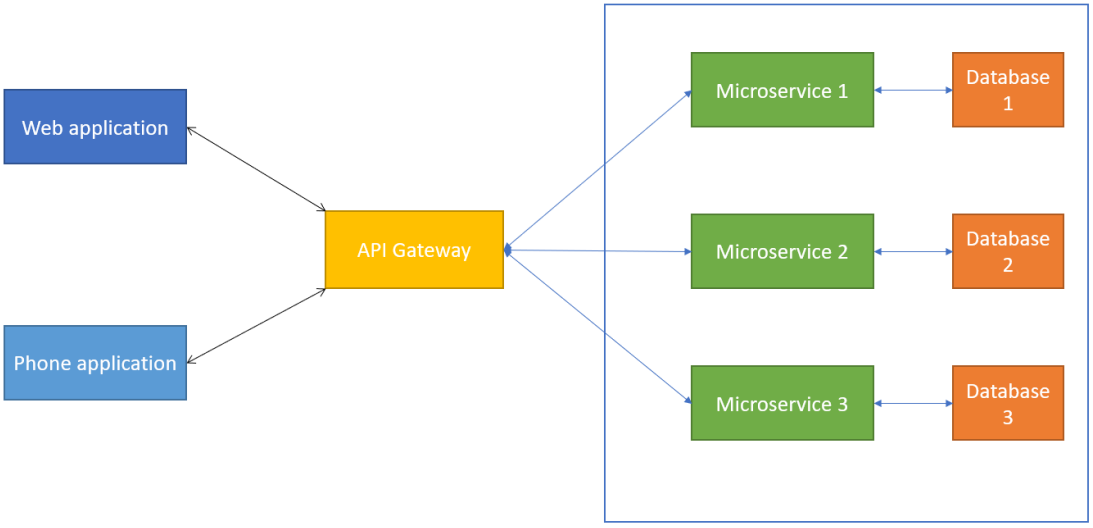
```
Task<TEntity> ExecuteCommandAsync<TEntity>(Func<IMongoRepositoryLocator,
                                           Task<TEntity>> command)
                                           where TEntity : class,
                                                           IMongoDocumentBase
{
   try
   {
      return await command.Invoke(Locator);
   }
   catch (MongoDB.Bson.BsonSerializationException bsonException)
   {
      _logger.LogError(bsonException.Message,
                       GetExceptionParams(bsonException));
       throw;
   }
   catch (Exception exception)
   {
      _logger.LogError(exception.Message, GetExceptionParams(exception));
      throw;
   }
}
```

Obrázek 33: Jedna z generických metod abstraktní báze Unit of Work

Na obrázku lze vidět ukázku jedné z již zmíněných generických metod. Tato konkrétní metoda vrací generický typ TEntity, což může být kterýkoliv z dokumentů – podmínkou pro tento typ je pouze to, že se musí jednat o třídu a zároveň tato třída musí implementovat rozhraní IMongoDocumentBase.

4.4 Servisní vrstva

Tato vrstva slouží jako zprostředkovatel mezi vrstvou Web API a databázovou vrstvou. Třídy patřící do této vrstvy obsahují většinu business logiky aplikace samotné. Většina tříd, které tato vrstva obsahuje se řadí mezi takzvané služby – jedná se o třídy, které v sobě zapouzdřují veškerou aplikační logiku týkající se některé z podmnožin aplikace. Typickým příkladem může být služba, která má na starost přihlašování a odhlašování uživatelů. Výhodou rozdělení na služby je jednoznačně vyšší přehlednost a udržovatelnost kódu.



Obrázek 34: Základní schéma architektury s využitím mikroslužeb [31]

V poslední době se prosazuje takzvaná architektura mikroslužeb, při které nejsou jednotlivé služby odděleny pouze logicky na úrovni tříd v kódu, ale jedná se o celé komplexní webové služby, které fungují nezávislé na sobě, jak lze vidět na obrázku 34. Případná komunikace mezi takto oddělenými službami pak může probíhat například na základě HTTP komunikace přes REST API, případně nějaké jiné formy síťové komunikace. Výhodou takovéhoto uspořádání je možnost škálování zátěže – a to v podstatě téměř neomezeně, pokud je celá architektura vytvořena takovým způsobem, že lze přidávat stále další a další mikroslužby.

4.4.1 Třída OverviewManager

Tato třída se nachází uvnitř již popsané servisní vrstvy a jejím hlavním úkolem je periodicky stahovat aktuální přehled základních údajů o kryptoměnách, cachovat je a případně je poskytovat klientské aplikaci, ať již formou požadavků zaslaných přes API, či formou rychlých updatů v reálném čase přes komunikační kanál SignalR.

```
public class OverviewManager : BackgroundService, IOverviewManager
public abstract class BackgroundService : IHostedService, IDisposable
{
    protected BackgroundService();

    public virtual void Dispose();
    public virtual Task StartAsync(CancellationToken cancellationToken);
    public virtual Task StopAsync(CancellationToken cancellationToken);
    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);
}
```

Obrázek 35: Třída OverviewManager implementující třídu BackgroundService

Třída implementuje abstraktní třídu BackgroundService a rozhraní IHostedService. Jedná se o standardní návrhový vzor, který je v novějších verzích .NET Core možné používat právě pro takovéto třídy.

```
public override async Task StartAsync(CancellationToken cancellationToken)
{
    await UpdateDataAsync(sendUpdate: false);
    _waitHandle.Set();

    _executingTask = ExecuteAsync(_stoppingCts.Token);

    return;
}
```

Obrázek 36: Metoda StartAsync třídy OverviewManager

První z metod, které je potřeba implementovat podle rozhraní IHostedService je metoda StartAsync. V této metodě je možné provést libovolnou inicializační logiku, která je potřeba pro pozdější periodické vykonávání nějaké akce. V případě této konkrétní třídy tedy dojde k úvodnímu získání potřebných dat a nastavení EventWaitHandle – tato třída je použita z toho důvodu, aby nebylo možné pokusit se o přístup k datům třídy dříve, než byla inicializace dokončena. Poté dojde k spuštění hlavního Tasku celé smyčky.

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        await UpdateDataAsync(true);

        await Task.Delay(_updateCyclePeriod, stoppingToken);
    }
}
```

Obrázek 37: Metoda ExecuteAsync třídy OverviewManager

Hlavní logika, která se periodicky vykonává, je definovaná v metodě ExecuteAsync – obsahuje cyklus while, který se ukončí pouze v případě, že předaný CancellationToken byl aktivován. Do té doby bude docházet k neustálému volání metody UpdateDataAsync, která obsahuje konkrétní logiku týkající se aktualizace dat uchovávaných v této třídě.

```
public override async Task StopAsync(CancellationToken cancellationToken)
{
   if (_executingTask == null)
       return;

   try
   {
       _stoppingCts.Cancel();
   }
   finally
   {
    await Task.WhenAny(_executingTask,
                       Task.Delay(Timeout.Infinite, cancellationToken));
   }
}
```

Obrázek 38: Metoda StopAsync třídy OverviewManager

Stejně jako je potřeba mít metodu pro start hostované služby, tak je potřeba implementovat i metodu zajišťující její správné ukončení. K tomu slouží metoda StopAsync, jejíž implementace především volá metodu Cancel na daném tokenu. Tím by mělo dojít k ukončení cyklu uvnitř metody ExecuteAsync a následnému dokončení celého spuštěného Tasku.

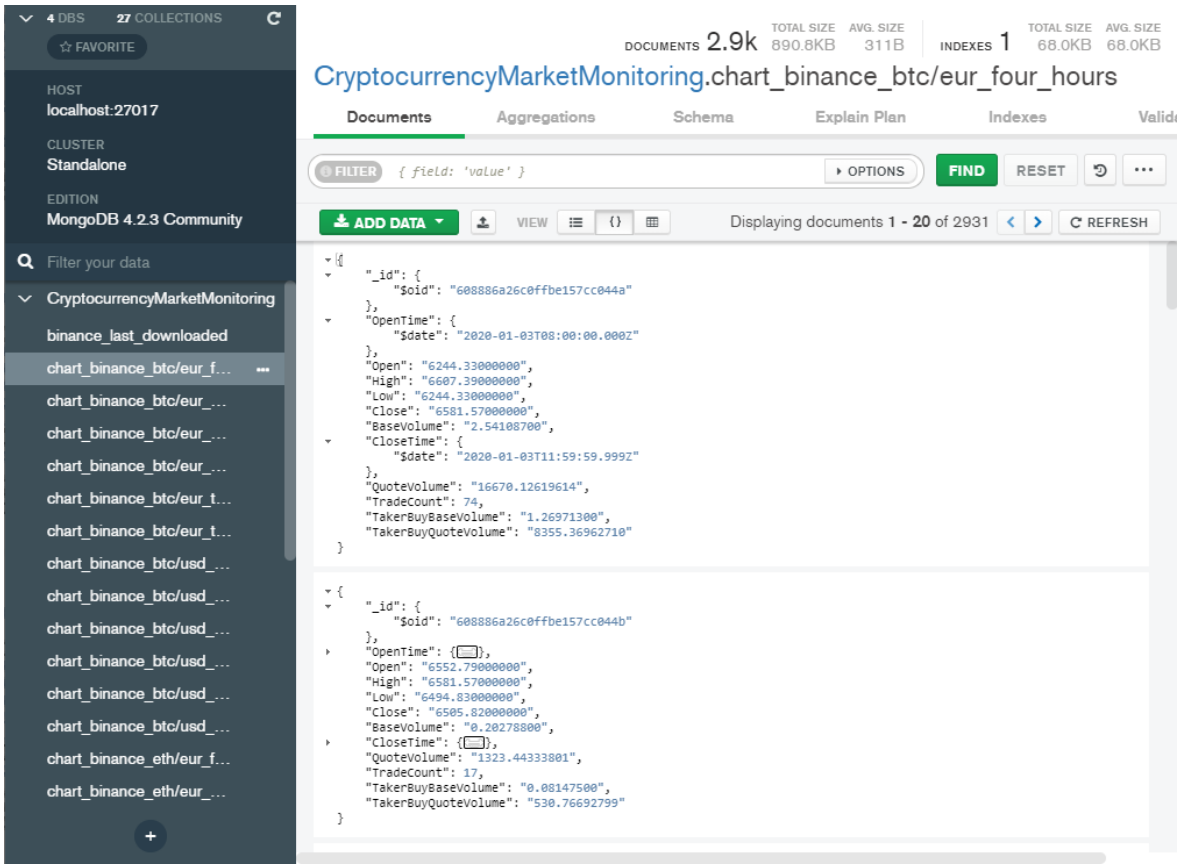
4.4.2 Třída BinanceChartDownloadManager

Tato třída slouží pro periodické stahování dat o vývoji cen kryptoměn ze serverů kryptoměnové burzy Binance. Podobně jako třída OverviewManager je implementována s využitím standardního rozhraní IHostedService Obsahuje tedy opět metody StartAsync a StopAsync, které slouží k provedení inicializace při startu serveru a následné deinicializaci.

```
private async Task DownloadDataAsync(string currency,
                                     string vsCurrency,
                                     IntervalType intervalType,
                                     CancellationToken stoppingToken)
{
   if (_lastDownloadedStamps.TryGetValue($"currency", var lastStamp))
   {
      var klines = await _binanceClient.Spot.Market
                         .GetKlinesAsync($"{currency}{binanceVsCurrency}",
                         (KlineInterval)intervalType,
                         startTime: lastStamp.Timestamp,
                         limit: 1000,
                         ct: stoppingToken);
```

Obrázek 39: Metoda pro stahování dat z Binance API

Nejdůležitější část logiky se potom odehrává v metodě DownloadDataAsync, která s využitím HTTP komunikace kontaktuje servery Binance a stahuje aktuální data. Struktura těchto dat je poměrně komplexní – nejedná se pouze o údaje o ceně v určitém čase, ale i údaje o tom, jaká byla otevírací cena, zavírací cena, počet uskutečněných obchodů v daném intervalu a další. Pro každou kryptoměnu jsou tato data dostupná ve velkém množství intervalů – může se tedy jednat například o data, kdy jsou údaje agregovány po minutách, hodinách, dnech či týdnech.



Obrázek 40: Uložení dat o ceně do jednotlivých kolekcí v databázi

Z důvodu optimalizace výkonu a logického členění dat se pak tato cenová data pro každou dvojici kryptoměn ukládají do samostatné kolekce v databázi. Na jednotlivé kolekce se pak údaje dělí i podle zvoleného časového intervalu. Na obrázku 40 lze v levé části vidět seznam jednotlivých kolekcí, zatímco v pravé části jsou zobrazeny dokumenty uložené uvnitř vybrané kolekce.

4.5 Vrstva API

Tato vrstva poskytuje rozhraní, kterým se serverová aplikace prezentuje navenek. Jedná se o standardní REST API implementované na základech HTTP komunikace. V ASP.NET

Core je zvykem, že API se skládá z libovolného množství tříd (typicky označované jako controller), kdy každá z nich obsahuje libovolné množství takzvaných endpointů.

```
[ApiController]
[Route("[controller]")]
public class OverviewController : ControllerBase
{

   public OverviewController(ILogger<OverviewController> logger,
                             IOverviewService cryptocurrencyOverviewService)
   {
      _logger = logger;
      _cryptocurrencyOverviewService = cryptocurrencyOverviewService;
   }

   [HttpGet("currencies")]
   public IActionResult GetSupportedCurrencies()
   {
      return Ok(_cryptocurrencyOverviewService.GetSupportedCurrencies());
   }

   [HttpGet("overview-all/{currency}")]
   public IActionResult GetCryptocurrencyOverviewAll(string currency = "usd")
   {
      return Ok(_cryptocurrencyOverviewService.GetOverviewAll(currency));
   }

   private readonly ILogger<OverviewController> _logger;
   private IOverviewService _cryptocurrencyOverviewService;
}
```

Obrázek 41: Ukázka implementace API controlleru

Každý jednotlivý endpoint musí mít unikátní cestu, pomocí které je potom server schopen určit, na kterou metodu má směrovat který požadavek. Standardně se využívají metody GET (pro čtení záznamů), POST (pro vytváření nových záznamů), PUT (pro aktualizaci záznamů) a DELETE (pro mazání záznamů).

```
app.UseEndpoints(endpoints =>
{
   endpoints.MapRazorPages();
   endpoints.MapControllers();
   endpoints.MapFallbackToFile("index.html");
});
```

Obrázek 42: Nastavení endpoint routingu v konfiguraci serveru

Třídy označené atributem ApiController není potřeba manuálně instancovat – framework si vytváření instancí hlídá sám. Je pouze potřeba v konfiguraci serveru zavolat metodu UseEndpoints (pro novější verze ASP.NET Core, dříve byla používána metoda UseMvc či jiné metody). Součástí volání je i metoda MapControllers, která zajistí mapování všech endpointů definovaných uvnitř tříd typu controller. Pokud je pak na server poslán HTTP požadavek s příslušnými parametry, server sám určí, jestli je potřeba vytvořit instanci nějakého controlleru a zavolat patřičnou metodu.

4.6 Klientská část aplikace

Klientská část aplikace je, jak již bylo zmíněno, napsána s využitím frameworku Blazor. Využívá se zde tedy opět programovací jazyk C#, ale ve spojení se značkovacím jazykem HTML a kaskádovými styly CSS. Spojením těchto tří prvků vznikají takzvané komponenty, ze kterých je potom složena webová aplikace. Kromě těchto komponent však lze v Blazoru vytvářet i třídy napsané čistě v jazyku C#, které pouze obstarávají nějakou aplikační logiku, bez návaznosti na vzhled uživatelského rozhraní. Samozřejmě pak lze používat i stránky obsahující pouze HTML a CSS, pokud není potřeba vykonávat v nich nějakou komplexnější logiku.

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.Services.AddSyncfusionBlazor();

        builder.RootComponents.Add<App>("#app");

        builder.Services
               .AddScoped<IAuthenticationService, AuthenticationService>()
               .AddScoped<IHttpService, HttpService>()
               .AddScoped<ILocalStorageService, LocalStorageService>();


        builder.Services.AddScoped(x =>
        {
            return new HttpClient()
            {
                BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
            };
        });

        var host = builder.Build();

        var authenticationService = host.Services
                                        .GetRequiredService<IAuthenticationService>();
        await authenticationService.Initialize();

        await host.RunAsync();
    }
}
```

Obrázek 43: Hlavní metoda Blazor aplikace

Stejně jako serverová aplikace v ASP.NET Core se i Blazor aplikace v základu chová jako konzolová aplikace – hlavní třídou celé aplikace je třída Program s metodou Main. Uvnitř těla této metody je potřeba vytvořit instanci třídy WebAssemblyHostBuilder, pomocí které se následně spustí aplikace jako taková (na úplném konci metody Main se volá metoda RunAsync). Ještě před tím je však možné provádět libovolnou konfiguraci aplikace – především se jedná o registraci tříd pro Dependency Injection. Framework podporuje DI sám o sobě, není tedy potřeba použít externí knihovny. Je však možno používat i tyto, jako například Autofac, ale vzhledem k tomu, že framework Blazor lze stále považovat za relativní novinku, tak ještě nemá tak širokou podporu, jako je tomu například u ASP.NET Core.

4.6.1 Layouty

```
@inherits LayoutComponentBase
@inject NavigationManager NavigationManager

<div class="topbar py-0">
    <NavMenu />
</div>

<div class="content">
    @Body
</div>

<footer class="page-footer footerbar font-small pt-0 fixed-bottom">
    <div class="footer-copyright text-center py-1">
        ©2021 Daniel Večeř
    </div>
</footer>
```

Obrázek 44: Definice hlavního layoutu aplikace

Kromě komponent a stránek každá Blazor aplikace obsahuje i takzvané layouty. Jedná se v podstatě o šablony, u kterých se předpokládá, že budou využívány napříč celou aplikaci. Defaultně má každá aplikace napsaná v Blazoru hlavní layout (nazvaný standardně MainLayout.razor), který slouží jako definice vzhledu hlavní stránky aplikace. Do tohoto layoutu lze potom dynamicky vkládat měnící se obsah stránky – například přes parametr @Body.

Kromě tohoto způsobu vkládání stránek lze však využít i vnořování layoutů – například do hlavního layoutu stránky lze vložit layout obsahující navigační menu aplikace. Rovněž lze definovat více layoutů pro jednotlivé části aplikace a dynamicky mezi nimi přepínat, pokud je potřeba významně měnit grafický vzhled aplikace při přechodu mezi jednotlivými částmi. Každý layout může využívat jednak globální kaskádové styly, tak své vlastní. Například pro layout MainLayout.razor lze definovat kaskádový styl MainLayout.razor.css – je nutno dodržet tuto konvenci pojmenování, aby mohl framework automaticky vytvořit provázání.

```
<div class="@NavMenuCssClass navbar-collapse navbar-dark">
    <ul class="navbar-nav mr-auto">
       <li class="nav-item">
          <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
             <span class="oi oi-pulse" aria-hidden="true"></span> Overview
          </NavLink>
       </li>
    </ul>

    @if (AuthenticationService.User == null)
    {
       <ul class="navbar-nav float-md-right">
          <li class="nav-item">
             <NavLink class="nav-link" href="login">
                 <span class="oi oi-account-login"/> Log In
             </NavLink>
           </li>
           <li class="nav-item">
             <NavLink class="nav-link" href="register">
                  <span class="oi oi-person"/> Sign Up
             </NavLink>
           </li>
       </ul>
    }
```

Obrázek 45: Ukázka části layoutu pro navigační menu aplikace

I do layoutů lze vkládat kód napsaný v jazyce C# a dynamicky tak měnit, které prvky mají být zobrazeny v jaký moment – příkladem může být například navigační menu, ve kterém lze přidávat nebo naopak odebírat položky podle toho, jestli je uživatel přihlášen. Pro zajištění toho, že dojde k překreslení uživatelského rozhraní je však typicky potřeba použít metodu StateHasChanged, která danou stránku informuje o tom, že je potřeba zpracovat změny. Pro realizaci navigace lze použít komponentu NavLink, která je přímo součástí frameworku Blazor, identicky však funguje i klasický odkaz realizovaný prostřednictvím čistého HTML.

4.6.2 Stránky

```
@page "/"
@using CryptocurrencyMarketMonitoring.Shared
@using System.Collections.ObjectModel
@using Syncfusion.Blazor.Grids
@using Syncfusion.Blazor.DropDowns;
@using System.ComponentModel
@using System.Linq;
@using Microsoft.AspNetCore.SignalR.Client
@using System.Collections.Generic;
@using Microsoft.Extensions.DependencyInjection;
@using CryptocurrencyMarketMonitoring.Client.Services;
@using Syncfusion.Blazor.Spinner;

@implements IAsyncDisposable

@inject NavigationManager NavigationManager
@inject IHttpService HttpService
```

Obrázek 46: Úvod definice stránky

Kód každé stránky musí začínat definicí cesty – k tomu slouží direktiva @page zapsaná na prvním řádku souboru. Díky tomu je pak framework schopen automaticky provádět směrování. Stejně jako v případě klasické třídy v jazyce C# pak může následovat celá řada direktiv using, pomocí kterých lze definovat, jaké knihovny a komponenty má daná stránka používat. Pomocí direktivy @implements lze implementovat polymorfismus – daná stránka například může dědit z rozhraní IAsyncDisposable.

Speciálním případem je pak direktiva @inject. Jelikož stránky v Blazoru nemají na rozdíl od klasických tříd v C# konstruktor, přes který by mohly být do instance třídy vloženy závislosti, lze toho dosáhnout právě skrze tuto direktivu. Pokud tedy v třídě Program byla registrována některá ze tříd specifikovaných direktivou @inject, dojde při vytvoření instance stránky automaticky k naplnění instance této třídy.

```
protected override async Task OnInitializedAsync()
{
   SupportedCurrencies = await HttpService
                                  .Get<List<string>>("Overview/currencies");
   SelectedCurrency = SupportedCurrencies.FirstOrDefault(x => x == "USD");
   ActiveCurrency = SelectedCurrency;
   Cryptocurrencies = await HttpService
         .Get<List<OverviewDto>>($"Overview/overview-all/{SelectedCurrency}");

   _updateHubConnection = new HubConnectionBuilder()
                           .AddJsonProtocol(options =>
                           {
                              Options
                              .PayloadSerializerOptions
                              .PropertyNamingPolicy = null;
                           })
                           .WithUrl(NavigationManager
                           .ToAbsoluteUri("/overview-update")).Build();

   _updateHubConnection
        .On<IEnumerable<OverviewUpdateDto>>("ReceiveUpdate", (updates) =>
        {
            DoGridUpdate(updates);
        });


   await _updateHubConnection.StartAsync();

   await _updateHubConnection.SendAsync("Subscribe",
                                        _updateHubConnection.ConnectionId,
                                        SelectedCurrency);

   _loading = false;
   StateHasChanged();
   await base.OnInitializedAsync();
}
```

Obrázek 47: Inicializační metoda stránky

Každá strana rovněž obsahuje metodu OnInitialized (případně asynchronní variantu OnInitializeAsync), kterou lze přetížit a implementovat tak svou vlastní logiku, která se má spustit, jakmile dojde k inicializaci dané stránky – uvnitř této metody je vhodné definovat například získání dat z API nebo inicializaci komunikace se serverem prostřednictvím socketů. V rámci tohoto inicializačního procesu lze také měnit vzhled stránky – například odložit renderování určité komponenty až na moment, kdy jsou pro ni dostupná data. K tomu

je ale potřeba opět využít metodu StateHasChanged, která dá stránce povel zahájit znovu vykreslování.

4.6.3 Formuláře a validace

Framework Blazor obsahuje velké množství předem nachystaných komponent pro nejčastěji používané uživatelské operace. Jednou z takových operací je potřeba vyplnit data do formuláře a následně je odeslat na server, často s požadavkem na validaci správnosti odeslaných dat.

```
<EditForm Model="@_login" OnValidSubmit="HandleValidSubmit">
   <DataAnnotationsValidator />
       <div class="form-group d-flex justify-content-center">
          <label class="w-50">
             Username
             <InputText @bind-Value="_login.Username" class="form-control" />
             <ValidationMessage For="@(() => _login.Username)" />
          </label>
       </div>
       <div class="form-group d-flex justify-content-center">
          <label class="w-50">
             Password
             <InputText @bind-Value="_login.Password" class="form-control" />
             <ValidationMessage For="@(() => _login.Password)" />
          </label>
       </div>
       <button class="btn btn-primary">
          @if (_loading)
          {
             <span class="spinner-border spinner-border-sm mr-1"></span>
          }
          Login
       </button>
       @if (!string.IsNullOrEmpty(_error))
       {
          <div class="alert alert-danger mt-3 mb-0">@_error</div>
       }
</EditForm>
```

Obrázek 48: Ukázka formuláře s validací ve frameworku Blazor

Komponenta InputText umožňuje obousměrný data binding na libovolnou vlastnost nebo proměnnou obsaženou v C# kódu stránky. To umožňuje poměrně efektivní a pohodlnou práci i v případě komplexních formulářů – všechna data lze například ukládat do jednoho složitého objektu a odesílat je poté na server jako celek. Komponenty

DataAnnotationsValidator a ValidationMessage pak slouží pro automatickou validaci uživatelských vstupů.

```
public class UserDto
{
    public string Id { get; set; }
    [Required]
    public string FirstName { get; set; }
    [Required]
    public string LastName { get; set; }
    [Required]
    [EmailAddress]
    public string Email { get; set; }
    public string Token { get; set; }
    [JsonIgnore]
    public string PasswordHash { get; set; }
    [RegularExpression("^(?=.*[0-9]).{8,}$")]
    [Required]
    public string Password { get; set; }
}
```

Obrázek 49: DataAnnotation atributy pro automatickou validaci

Pro správnou funkci automatické validace je nutné vlastnosti třídy, jejichž data chceme validovat, označit atributy ze jmenného prostoru DataAnnotations. Těchto atributů existuje celá řada – předpřipravené jsou například atributy pro vyžádání povinného pole, kontrolu emailové adresy nebo porovnání s regulárním výrazem. Kromě toho však lze definovat i své vlastní validační atributy – podmínkou je pouze implementování bázové třídy ValidationAttribute a přetížení jejich metod svou vlastní logikou.

4.6.4 Komplexní komponenty

Kromě používání komponent, které jsou již součástí frameworku Blazor, jako byla například komponenta EditForm pro implementaci jednoduchých formulářů, je samozřejmě možné i vytvářet vlastní komponenty. V současné době již také existuje velké množství doplňujících balíčků komponent určených přímo pro framework Blazor, jejichž cílem je nabízet funkcionalitu, která v základní podobě frameworku chybí.

```
<SfStockChart Title="Market data provided by Binance">
    <StockChartTooltipSettings Enable="true"></StockChartTooltipSettings>
    <StockChartCrosshairSettings Enable="true"></StockChartCrosshairSettings>
    <StockChartSeriesCollection>
       <StockChartSeries DataSource="@ChartData" Type="ChartSeriesType.Candle"
                                    XName="@nameof(ChartDataDto.Date)"
                                    YName="@nameof(ChartDataDto.Close)"
                                    High="@nameof(ChartDataDto.High)"
                                    Low="@nameof(ChartDataDto.Low)"
                                    Open="@nameof(ChartDataDto.Open)"
                                    Close="@nameof(ChartDataDto.Close)"
                                    Volume="@nameof(ChartDataDto.Volume)">
      </StockChartSeries>
   </StockChartSeriesCollection>
</SfStockChart>
```

Obrázek 50: Použití komponenty pro zobrazení svíčkového grafu

Ukázkou takového použití může být například komponenta SfStockChart – jedná se o komponentu z balíčku Blazor komponent od společnosti SyncFusion, která poskytuje poměrně složité možnosti vizualizace dat o cenovém vývoji nejen kryptoměn. Samozřejmostí je plná podpora data bindingu – grafovou komponentu tak lze bindovat přímo na kolekci komplexních objektů a specifikovat, které vlastnosti objektu mají být použity.

4.7 Zabezpečení aplikace

Při vývoji webové aplikace je v dnešní době bezpodmínečně nutné dbát na důkladné zabezpečení všech jejích součástí. Framework ASP.NET poskytuje mnoho možností, jak dosáhnout vysokého stupně zabezpečení – často je ovšem potřeba toto zabezpečení správně nakonfigurovat. Nelze se tedy spolehnout na to, že framework sám o sobě je zcela bezpečný. Klíčová je především bezpečnost serverové části aplikace a databázového úložiště, ale nelze opomíjet ani zabezpečení klientské části.

4.7.1 Nastavení HTTPS komunikace

```
webBuilder.ConfigureKestrel(options =>
{
   var port = 5001;
   var pfxFilePath = @"C:\certificate.pfx";
   var pfxPassword = "pwd";

   options.Listen(IPAddress.Any, port, listenOptions =>
   {
      listenOptions.Protocols = HttpProtocols.Http1AndHttp2;
      listenOptions.UseHttps(pfxFilePath, pfxPassword);
   });
});
```

Obrázek 51: Ukázka nastavení HTTPS certifikátu pro server Kestrel

Pro vyšší bezpečnost je pro komunikaci po síti mezi klientem a serverem vhodné používat výhradně protokol HTTPS. Ten zajišťuje důvěryhodnou autentizaci webového serveru a dále pak soukromí a integritu veškerých vyměněných dat – veškerá data, která si mezi sebou server a klient vymění jsou šifrována. Pro použití HTTPS je potřeba, aby server měl k dispozici platný certifikát. V ASP.NET Core existuje několik možností načtení certifikátu – nejčastěji se používá načtení ze souboru či ze systémového úložiště certifikátů.

```
app.UseHttpsRedirection();
```

Obrázek 52: Nastavení automatického přesměrování

Další z vlastností frameworku ASP.NET Core je podpora automatického přesměrování HTTP požadavků na protokol HTTPS – toho lze docílit zavoláním metody UseHttpsRedirection ve třídě Startup. Pro správnou funkci protokolu HTTPS je samozřejmě potřeba, aby server disponoval platným certifikátem – v opačném případě by většina moderních prohlížečů při vstupu na stránku uživatelům vypsala varování o tom, že se pokouší přistoupit k webu, který není zabezpečen.

4.7.2 Bezpečné ukládání uživatelských hesel

```
public string Hash(string password)
{
   using (var algorithm = new Rfc2898DeriveBytes(password,
                                                  _saltSize,
                                                  _options.Iterations,
                                                  HashAlgorithmName.SHA256))
   {
      var key = Convert.ToBase64String(algorithm.GetBytes(_keySize));
      var salt = Convert.ToBase64String(algorithm.Salt);

      return $"{_options.Iterations}.{salt}.{key}";
   }
}
```

Obrázek 53: Generování bezpečného hashe z uživatelského hesla

Jelikož aplikace umožňuje registraci uživatelů, je potřeba z hlediska zabezpečení dbát i na ochranu uživatelských údajů, především pak samotného hesla. Obecným doporučením je nikdy neukládat hesla uživatelů v textové podobě. To může vést k několika velmi závazným komplikacím. Jednak by při úniku dat z databáze útočník mohl získat přístup k heslům samotným – navzdory zásadám pro tvorbu hesel je zcela běžné, že uživatelé používají stejná hesla pro přístup k více službám, což může být zásadní bezpečnostní problém. Druhým problémem je to, že ukládání hesel touto formou v podstatě znamená, že služba samotná zná uživatelovo heslo, ačkoliv k jeho uchování není pro účely fungování služby žádný důvod.

Je tedy jednoznačně vhodnější ukládat hashe hesel. Použití obyčejného hashe však již v dnešní době nelze považovat za bezpečné – je potřeba vhodně zvolit hashovací funkci a dodržovat jisté další zásady. Jednou z těchto zásad je použití soli (salt), což je náhodná informace, která se přidá k heslu před tím, než bude proveden výpočet hashe. Smyslem tohoto opatření je znemožnit útočníkovi použití předem připravených tabulek pro daný hashovací algoritmus.

Ve frameworku .NET existuje celá řada možností, jak hashovat hesla. Pro účely tohoto projektu byla zvolena funkce pro odvození klíče RFC2898, která je v jazyku C# dostupná pod třídou Rfc2898DeriveBytes. Tato funkce na textovou podobu hesla společně se zvolenou solí aplikuje pseudonáhodnou funkci. Tento postup se opakuje v závislosti na nastavení – pro vyšší bezpečnost je doporučeno používat vyšší množství iterací, například 10 000, ale tato hodnota logicky stoupá spolu s neustále se zvyšujícím výpočetním výkonem počítačů. Dalším faktorem je velikost soli – v současné době je doporučováno používat sůl

o délce alespoň 128 bitů. Je potřeba pamatovat i na délku klíče samotného – za bezpečné lze v tomto případě stále považovat klíče o délce 256 bitů a vyšší. [11]

```
public (bool Verified, bool NeedsUpgrade) Check(string hash, string password)
{
   var parts = hash.Split('.', 3);
   if (parts.Length != 3)
   {
      throw new FormatException("Unexpected hash format.");
   }

   var iterations = Convert.ToInt32(parts[0]);
   var salt = Convert.FromBase64String(parts[1]);
   var key = Convert.FromBase64String(parts[2]);

   var needsUpgrade = iterations != _options.Iterations;

   using (var algorithm = new Rfc2898DeriveBytes(password,
                                                 salt,
                                                 iterations,
                                                 HashAlgorithmName.SHA256))
   {
      var keyToCheck = algorithm.GetBytes(_keySize);
      var verified = keyToCheck.SequenceEqual(key);

      return (verified, needsUpgrade);
   }
}
```

Obrázek 54: Metoda pro ověření hesla vůči uložené hashované podobě

Stejnou funkci pak lze aplikovat i při potřebě zpětně ověřit heslo. Uživatel při pokusu o přihlášení zašle své heslo v textové podobě. Z databáze je pak získán příslušný hash a salt, s pomocí kterého lze opětovně provést celý algoritmus. Výsledný hash je poté porovnán s původně uloženým hashem a na základě tohoto lze vyhodnotit, jestli je zadané heslo správné.

4.7.3 Autentizace uživatelů pomocí tokenů

Obrázek 55: Schéma struktury JWT (JSON Web Token) [32]

Pro autentizaci a autorizaci při komunikaci mezi serverem a klientem je použita standardizovaná technologie JWT. Princip fungování je takový, že klient první pošle serveru své přihlašovací údaje, na základě kterých potom server vygeneruje platný JWT token, který pošle zpět klientovi. Klient pak při každém volaném požadavku posílá zpět takto získaný token, kterým prokazuje, že se jedná o úspěšně ověřeného uživatele.

Vygenerovaný token se skládá ze tří částí znázorněných na obrázku 55. První částí je hlavička (header), která obsahuje popis algoritmu, pomocí kterého byl token vytvořen a typ tokenu. Druhou částí jsou pak hlavní data tokenu (payload), ve kterých může být libovolný obsah – typicky se jedná o údaje identifikující subjekt tokenu a seznam takzvaných claimů, které definují práva daného uživatele.

Poslední důležitou součástí tokenu je podpis (signature), který zajišťuje, že data obsažená v tomto tokenu jsou platná a nedošlo k jejich modifikace nějakou třetí stranou (pokud tato třetí strana nedisponuje klíčem použitým k vytvoření podpisu).

```
private string GenerateJwtToken(UserDto user)
{
   var tokenHandler = new JwtSecurityTokenHandler();
   var rsa = RSA.Create();
   rsa.ImportRSAPrivateKey(
      source: Convert.FromBase64String(_jwtOptions.Private),
      bytesRead: out int _
   );

   var tokenDescriptor = new SecurityTokenDescriptor
   {
      Subject = new ClaimsIdentity(new[] { new Claim("id", user.Id) }),
      Expires = DateTime.UtcNow.AddMinutes(60),
      SigningCredentials = new SigningCredentials(new RsaSecurityKey(rsa),
                                                  SecurityAlgorithms.RsaSha256)
   };

   var token = tokenHandler.CreateToken(tokenDescriptor);
   return tokenHandler.WriteToken(token);
}
```

Obrázek 56: Ukázka generování tokenu na serveru

Na obrázku lze vidět postup generování tokenu pro daného uživatele – v první řadě je potřeba získat privátní RSA klíč serveru, který je součástí konfiguračních souborů serverové aplikace. Používá se zde algoritmus SHA-256 s RSA o délce klíče 2048 bitů. S využitím tohoto algoritmu je potom vytvořen samotný token, součástí jehož dat je informace o tom, o kterého uživatele se jedná a po jakou dobu má token zůstat platný. Token lze následně v podobě hashovaného textového řetězce předat klientovi, který o jeho vygenerování zažádal.

```
private async Task AttachUserToContextAsync(HttpContext context,
                                            IUserService userService,
                                            string token)
{
   try
   {
       var tokenHandler = new JwtSecurityTokenHandler();
       var rsa = RSA.Create();
       rsa.ImportRSAPublicKey(
       source: Convert.FromBase64String(_jwtOptions.Public),
                                        bytesRead: out int _
       );

       tokenHandler.ValidateToken(token, new TokenValidationParameters
       {
           ValidateIssuerSigningKey = true,
           IssuerSigningKey = new RsaSecurityKey(rsa),
           ValidateIssuer = false,
           ValidateAudience = false,
           ClockSkew = TimeSpan.Zero
       }, out SecurityToken validatedToken);

       var jwtToken = (JwtSecurityToken)validatedToken;
       var userId = jwtToken.Claims.First(x => x.Type == "id").Value;

       context.Items["User"] = await userService.GetAsync(userId);
    }
    catch
    {
       // do nothing if jwt validation fails
    }
}
```

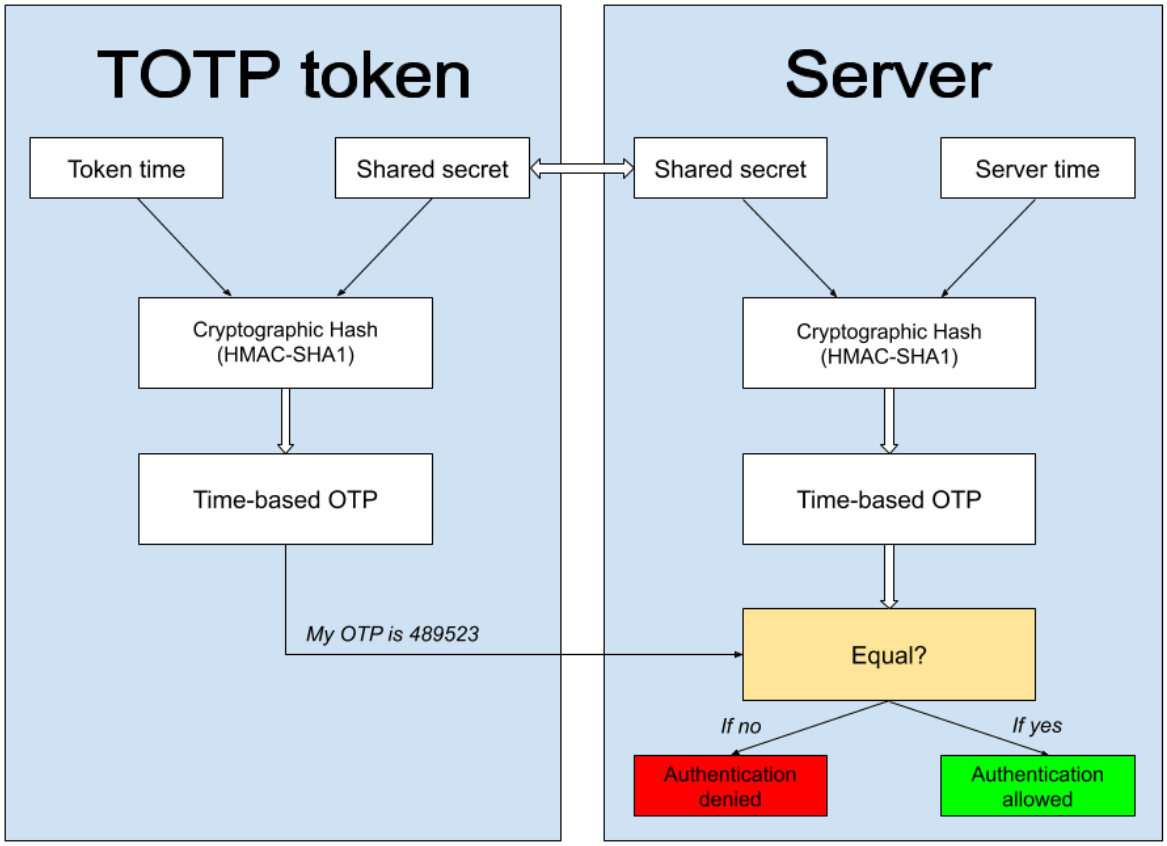
Obrázek 57: Validace přijatého tokenu

Při přijetí tokenu od klienta se postupuje obdobně – vzhledem k tomu, že je použito asymetrické šifrování s využitím algoritmu RSA, tak je ovšem pro ověření tokenu nutno použít veřejný klíč. Validační metoda je zavedena jako součást middleware, což znamená, že při každém přijatém HTTP požadavku dojde k vyvolání této metody, která zkontroluje, jestli je v hlavičce požadavku token, a pokusí se jej validovat.

4.7.4 Vícefázová autentizace

Vícefázová autentizace podstatně zvyšuje odolnost proti útokům na účty uživatelů – útočníkovi komplikuje situaci tím, že aby mohl provést úspěšný útok, tak by musel mít přístup nejen k heslu, ale i k zařízení, na které jsou odesílány autentizační kódy. V minulosti

byly pro tyto účely typicky používány mobilní telefony, na které se odesílaly kódy prostřednictvím SMS. V dnešní době se z důvodu obav o bezpečnost údajů zasílaných jako SMS často přechází na jiná řešení.



Obrázek 58: Schéma vícefázového ověření [33]

Jedním z relativně populárních řešení je použití aplikace Google Authenticator, která umožňuje vícefázovou autentizaci založenou na principu TOTP (Time-based One-Time Password). Fungování tohoto algoritmu je znázorněno na obrázku 58. Výhodou oproti SMS kódům je to, že autentizační kódy nejsou posílány přes nezabezpečený kanál směrem od serveru k uživatelovi – kódy jsou generovány automaticky na základě algoritmu, uživatel tedy může kdykoliv zadat svůj kód při pokusu o přihlášení, bez toho, aby si vyžádal jakékoliv informace od serveru. Hlavní nevýhodou je nutnost uložení tajného klíče na straně serveru – bez tohoto by nebyl schopen ověřovat kódy zaslané uživatelem. Toto tedy klade vysoké nároky na zabezpečení databáze serveru.

```
public static class GoogleAuthenticationHelper
{
   public static bool CheckCode(int code, string userKey)
   {
      if (userKey == null) return false;

      var generator = new TotpGenerator();
      var tfaValidator = new TotpValidator(generator);

      return tfaValidator.Validate(userKey, code, 300);
   }

   public static TotpSetup GenerateUserCode(string userName, string userKey)
   {
      var setupGenerator = new TotpSetupGenerator();

      var generatedCode = setupGenerator.Generate("DVCRYPTO",
                                                  userName,
                                                  userKey);
      return generatedCode;
   }
}
```

Obrázek 59: Generování a ověřování bezpečnostních kódů pro Google Authenticator

Pro aplikaci vícefázové autentizace založené na principu TOTP lze ve frameworku použít knihovnu AspNetCore.TOTP. Pro úvodní použití je potřeba uživateli vygenerovat kód, který může naskenovat nebo opsat do aplikace Google Authenticator. Důležitý je zde především tajný klíč, který je potřeba uložit na straně serveru – jen pomocí něj je pak možné ověřit kód, který uživatel zadá při přihlášení. Jelikož je algoritmus generování kódu závislý na času, je důležité, aby byl čas na mobilním telefonu i serveru správně synchronizovaný. Pro menší odchylky je vhodné nastavit jistou míru časové tolerance – v tomto případě je nastavena na pět minut. Všechny vygenerované kódy tedy mají defaultní dobu platnosti prodlouženou právě o pět minut.

4.7.5 Ochrana databáze

Jelikož projekt používá databázový systém MongoDB, který se řadí mezi takzvané NoSQL systémy, tak odpadá jedna z nejčastějších zranitelností, proti které je nutno se zabezpečit v případě použití SQL databáze, a to sice útok typu SQL injection. Vzhledem k tomu, že dotazy v Mongu jsou vytvářeny jako BSON objekty, tak je v tomto případě nemožné vložit do nich nějaký skript, který by vykonal nežádoucí operaci. Toto však platí jen při využití některého z nativních driverů, jako je například MongoDB C# driver. Specifickou oblastí

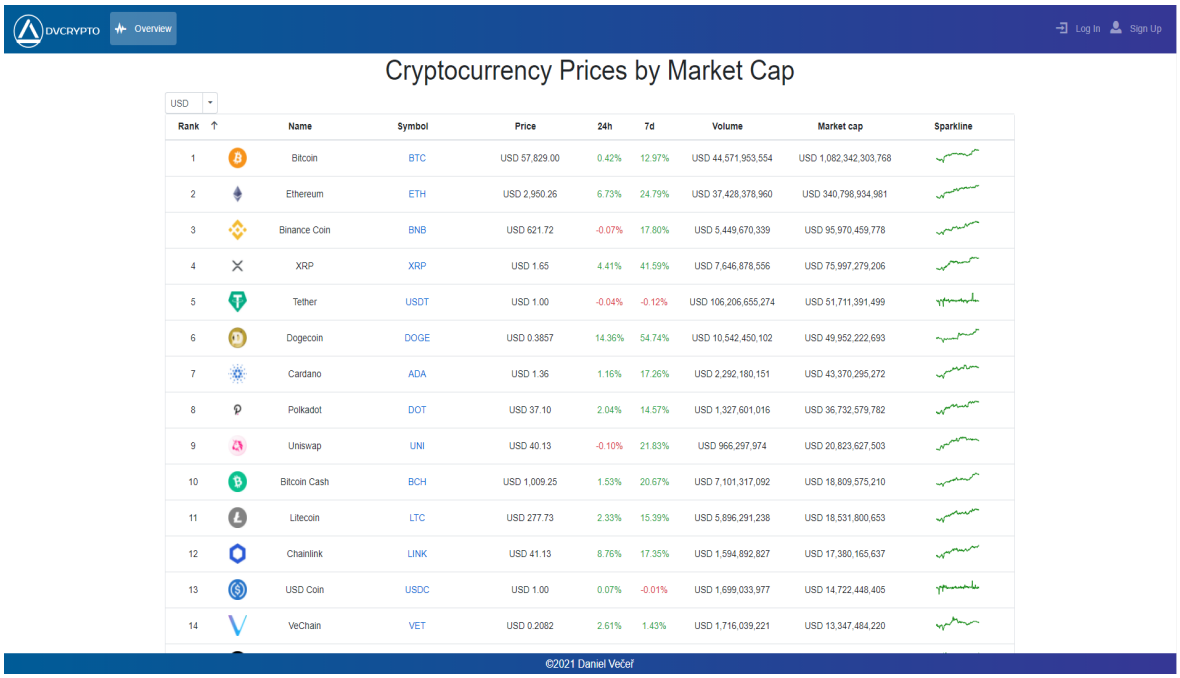
zranitelností je podpora JavaScriptových dotazů na MongoDB server, a to sice s použitím příkazů $where, mapReduce, $accumulator a $function. Pokud mají být tyto funkce podporovány, je potřeba důkladně ošetřit jakýkoliv uživatelský vstup – stejně jako by tomu bylo v případě sestavování SQL dotazu. Pro potřeby realizované aplikace ovšem není potřeba žádnou z těchto funkcí používat, a lze je tedy bezpečně zakázat přímo v nastavení serveru. K tomu lze využít parametr --noscripting při startu serveru, nebo nastavení parametru security.javascriptEnabled na false v konfiguračním souboru serveru.

5 DEMONSTRACE VÝSLEDNÉ APLIKACE

Výstupem praktické části práce je kromě samotného textu práce i plně funkční webová aplikace, jejíž vývoj byl popsán v předchozí kapitole. Serverová část webové aplikace je implementována s využitím již popsané technologie ASP.NET Core. Jedná se tedy o self-hosted webovou službu využívající integrovaný webový server Kestrel, která se navenek chová jako standardní konzolová aplikace. Hostování serverové části je možno provést na libovolném počítači s operačním systémem Windows, Linux nebo Mac OS X. Podporovány jsou také procesorové architektury x86-64 a ARM. Tato multiplatformní podpora se týká i databázového serveru MongoDB, který aplikace používá jako úložiště.

Klientská část webové aplikace je realizována formou SPA (Single Page Application), což znamená, že prohlížeč se k celé aplikaci chová, jako kdyby se jednalo o jednu jedinou HTML stránku. Obsah této stránky je nicméně dynamicky překreslován podle toho, která část aplikace má být v daný moment zobrazena. Výhodou tohoto přístupu je vyšší rychlost načítání při přechodu mezi jednotlivými částmi aplikace – není vždy potřeba načítat celou stránku, ale stačí znovu vykreslit pouze určitou část.

5.1 Hlavní stránka aplikace



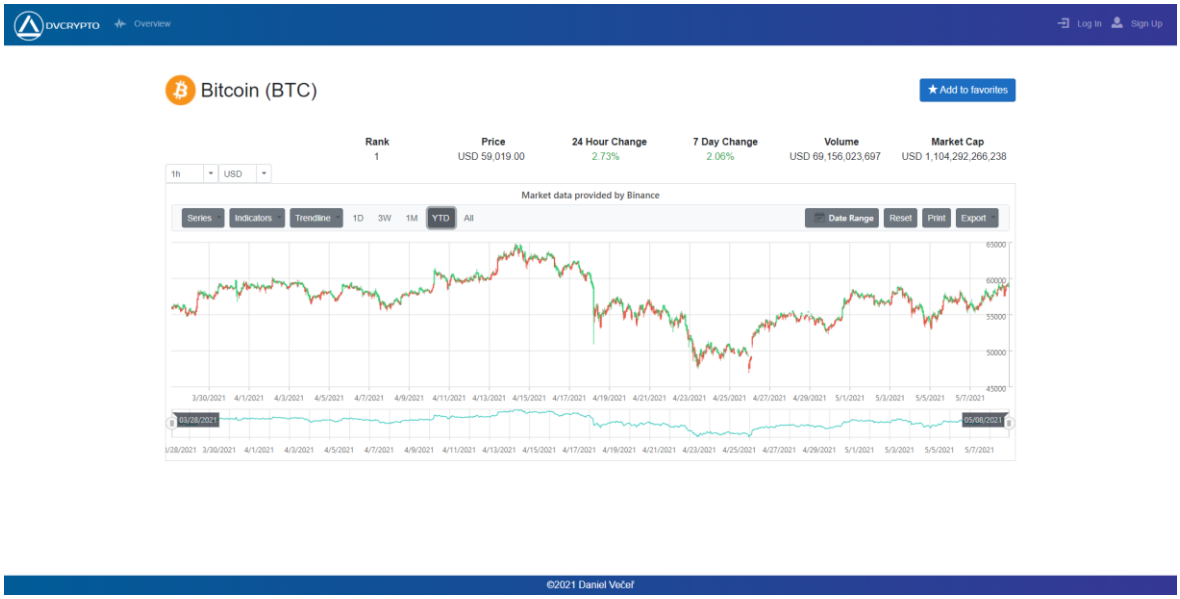
Obrázek 60: Vzhled hlavní stránky aplikace

Centrálním prvkem hlavního okna aplikace je data grid obsahující seznam všech monitorovaných kryptoměn, a to včetně přehledu základních údajů, tak jak lze vidět na

obrázku 60 (větší verze je dostupná jako příloha P II). Mezi základní zobrazované údaje patří grafické logo kryptoměny, plný název, symbol, pod kterým je kryptoměna obchodována na burzách, aktuální cena, procentuální změna ceny za posledních 24 hodin a 7 dní, celkový objem obchodů za posledních 7 dní, celková kapitalizace měny a graf zobrazující cenový vývoj za posledních 7 dní. Obsah gridu je plně dynamický – pokud má uživatel stránku otevřenou v prohlížeči, periodicky dochází k aktualizaci údajů, a to včetně změny pořadí (defaultní řazení je podle celkové hodnoty kapitalizace měny). Tento data grid obsahuje plnou podporu řazení podle libovolného sloupce (tam, kde to dává smysl – nelze například řadit podle sloupce s logem kryptoměny). Je rovněž plně implementováno stránkování – vždy je zobrazeno jen několik desítek záznamů, v dolní části komponenty se pak nachází ovládací prvky pro přepínání se mezi jednotlivými stránkami záznamů. Kliknutím na daný řádek pak lze přejít na detail vybrané kryptoměny.

Nad gridem samotným se nachází přepínač bázové měny – pomocí něj lze zvolit, ve které měně budou zobrazovány číselné hodnoty. Například při volbě EUR se v gridu samotném bude zobrazovat cena všech kryptoměn přepočítaná na eura, při volbě USD pak tato cena bude v amerických dolarech. V horní části stránky se nachází navigační menu, které se dynamicky mění podle toho, jestli je uživatel přihlášený nebo ne – přihlášený uživatel tak již například nevidí tlačítka pro přihlášení a registraci, ale naopak nově vidí tlačítko pro editaci svého profilu nebo tlačítko pro přechod na svůj seznam oblíbených kryptoměn.

5.2 Detail kryptoměny



Obrázek 61: Vzhled detailu kryptoměny

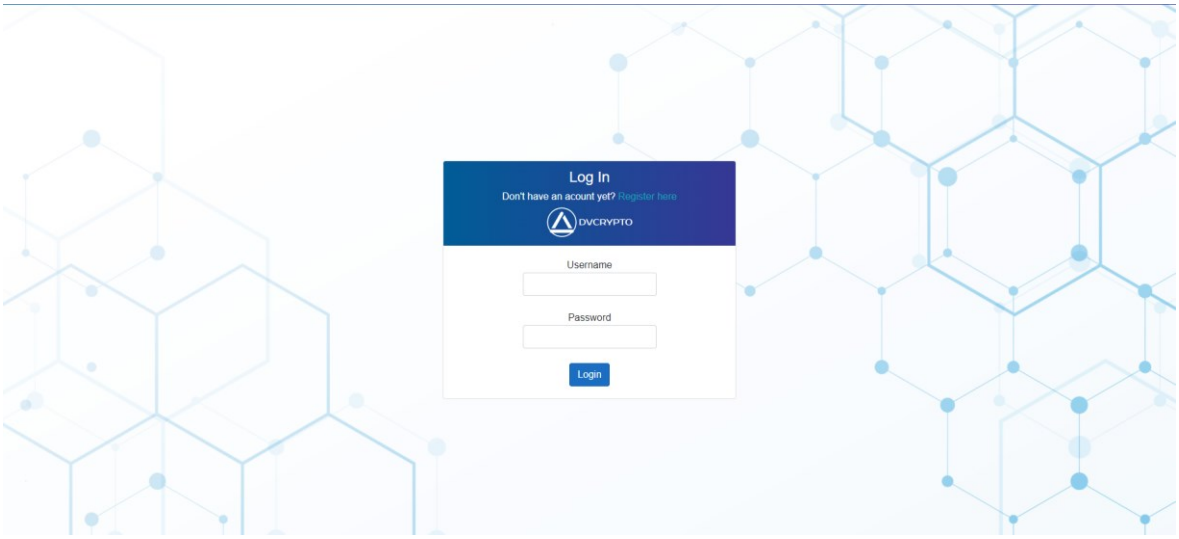
Při vybrání kryptoměny na hlavní stránce se zobrazí stránka obsahující detailní informace o vybrané kryptoměně. Tuto stránku lze vidět na obrázku 61 (větší verze je dostupná jako příloha P III). Kromě standardních údajů, které jsou zobrazeny i na hlavní stránce, se zde nachází především komplexní graf, který vizualizuje cenový vývoj dané kryptoměny v čase. Graf je vysoce interaktivní a je možné ho modifikovat podle potřeby. Samozřejmostí je možnost změny časového rozsahu, pro který jsou data zobrazována – toho lze dosáhnout hned několika způsoby, a to buď rychlou volbou z předdefinovaných časových filtrů v levé horní části grafového menu, kliknutím na tlačítko Date Range a volbou z kalendáře, případně pomocí posuvníků na časové ose umístěné ve spodní části grafu.

Pomocí tlačítka Series pak lze přepínat mezi jednotlivými typy grafů – defaultně se graf zobrazuje jako klasický svícový graf, který je schopný poskytnout nejvíce informací, ale komponentu lze přepnout například i do podoby klasického liniového grafu. V grafu lze rovněž vykreslit některý z dostupných indikátorů pro technickou analýzu. K vybrání je jich celá řada – například lze zmínit MACD (Moving Average Convergence Divergence), RSI (Relative Strength Index) a Moving Averages (MA). Graf umožňuje i vykreslení křivky znázorňující trend vývoje ceny. Stejně jako na hlavní stránce, tak i v detailu kryptoměny lze přepínat bázovou měnu. Při změně (například z defaultní hodnoty USD na EUR) tak dojde k aktualizaci grafu, který nově bude zobrazovat cenový vývoj dané kryptoměny v eurech. Na rozdíl od hlavní stránky je zde však definován i přepínač intervalu – ten určuje, jak jemné je členění jednotlivých bodů grafů. Při nastavení například na hodnotu 1h tak graf vykreslí jednu svíci každou hodinu. Dostupná je široká škála intervalů, od minutových až po měsíční.

Obrázek 62: Ukázka exportu grafu do obrázkového formátu

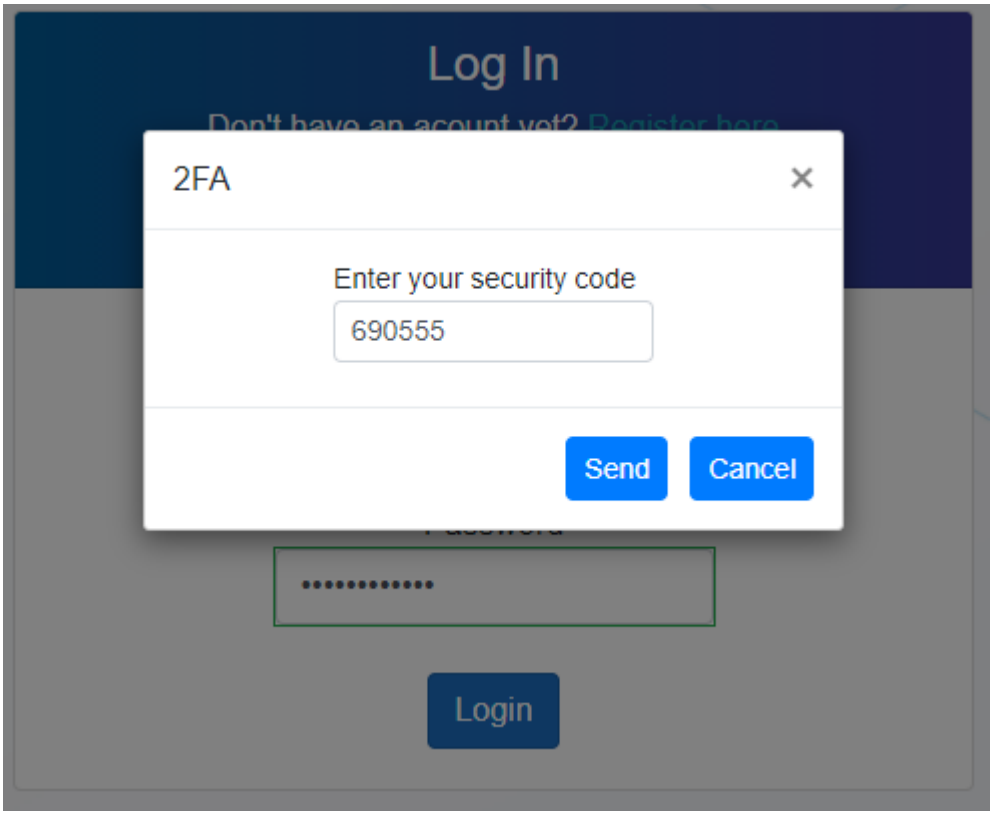
Uživateli je poskytnuta i možnost exportu zobrazeného grafu do obrázkového formátu (dostupné jsou formáty PNG, JPEG a SVG). Exportovat lze rovněž i do standardního formátu PDF dokumentu, případně lze požadovaný graf i přímo odeslat k tisku. Obrázek 62 ukazuje výstup exportu do obrázkového formátu.

5.3 Přihlášení uživatele



Obrázek 63: Vzhled přihlašovacího dialogu aplikace

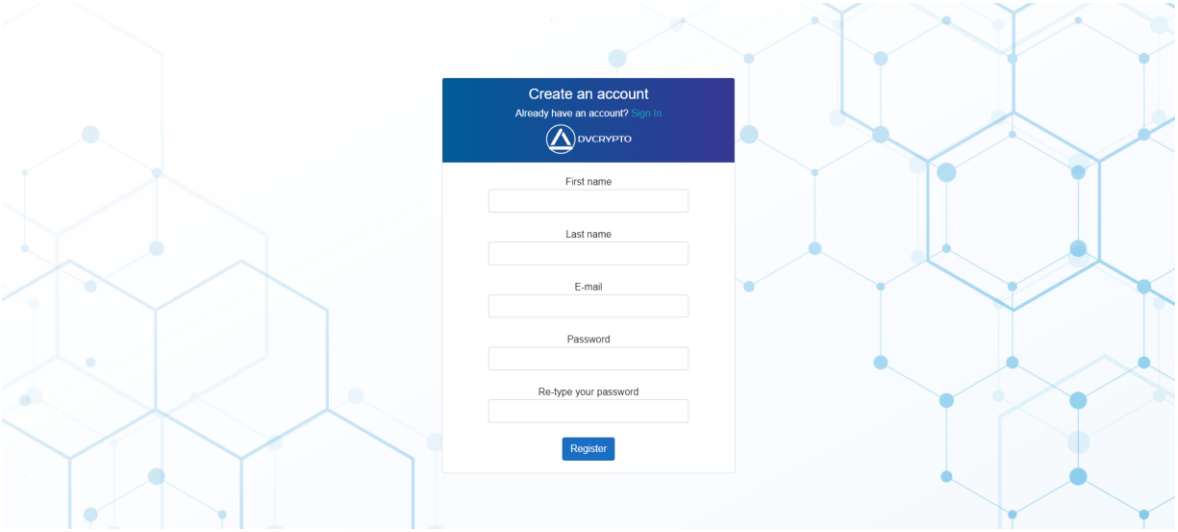
V pravé části horního navigačního menu se nachází tlačítko pro přechod na přihlašovací obrazovku (jak již bylo zmíněno, toto tlačítko je viditelné pouze v případě, že uživatel ještě není přihlášen). Jedná se o poměrně standardní formulář pro vyplnění uživatelského jména a hesla a odeslání těchto údajů na server pro ověření. Formulář lze vidět na obrázku 63 (větší verze je dostupná jako příloha P IV).



Obrázek 64: Vícefázová autentizace při přihlášení uživatele

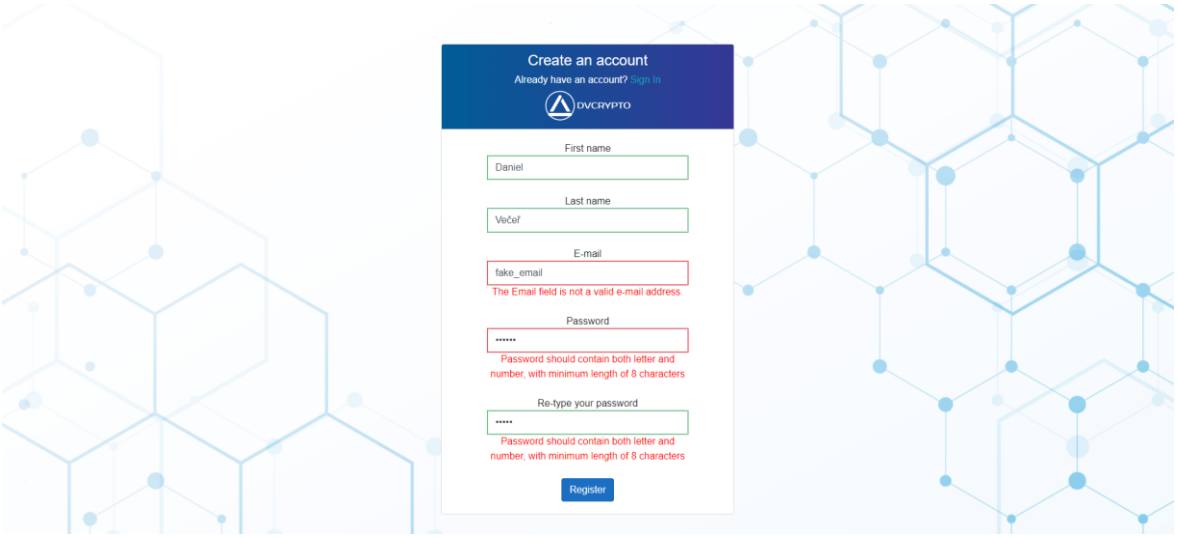
V případě, že má daný uživatel zapnutou dvoufázovou autentizaci pomocí mobilního zařízení, je ještě potřeba před schválením přihlašovacích údajů zadat autentizační kód z mobilní aplikace Google Authenticator. Po stisknutí tlačítka login se uživateli zobrazí modální dialog (viz obrázek 64), do kterého může tento kód zadat. Samotné přihlášení pak může proběhnout pouze v případě, že uživatel zadal validní kód.

5.4 Registrace nového uživatele



Obrázek 65: Vzhled registračního formuláře pro nového uživatele

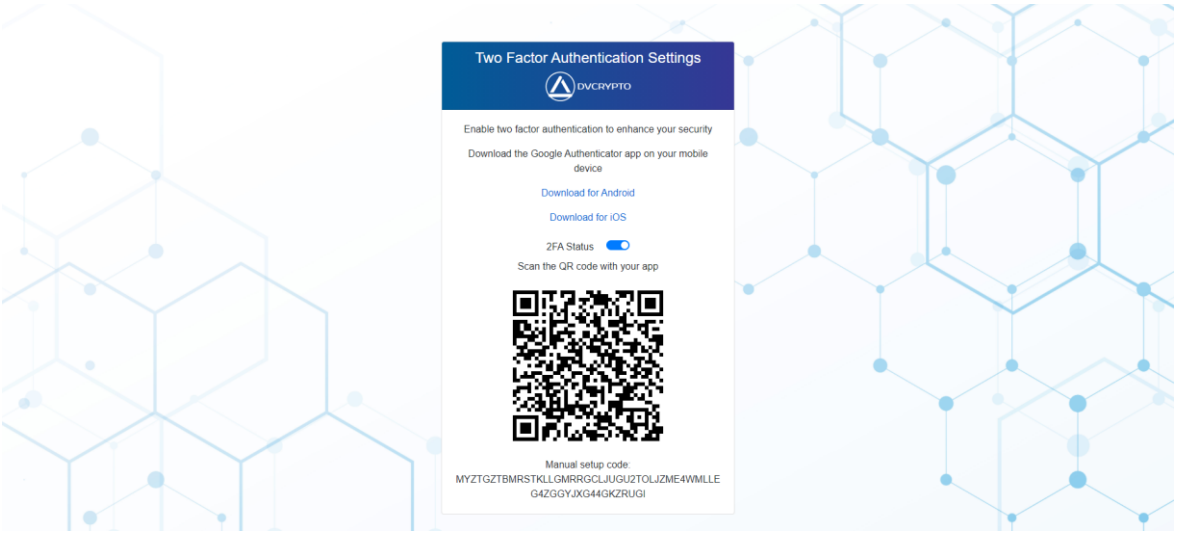
Na obrazovku pro registraci nových uživatelů lze přejít jak odkazem ze stránky pro přihlášení, tak příslušným navigačním tlačítkem v hlavním navigačním menu stránky Formulář lze vidět na obrázku 65 (větší verze je dostupná jako příloha P V). Do formuláře musí uživatel vyplnit základní údaje – jméno, příjmení, platnou emailovou adresu a své heslo. Stisknutím tlačítka pak dojde k odeslání těchto údajů na server. V případě platnosti všech zadaných údajů (kontroluje se například, jestli je zadaná emailová adresa v rámci všech uživatelů unikátní) dochází k úspěšné registraci a automatickému přesměrování na přihlašovací obrazovku. V opačném případě pak dojde k zobrazení chybové hlášky, která uživateli sdělí, který ze zadaných údajů není platný, a co musí udělat pro nápravu.



Obrázek 66: Ukázka dynamické validace vstupu při registraci nového uživatele

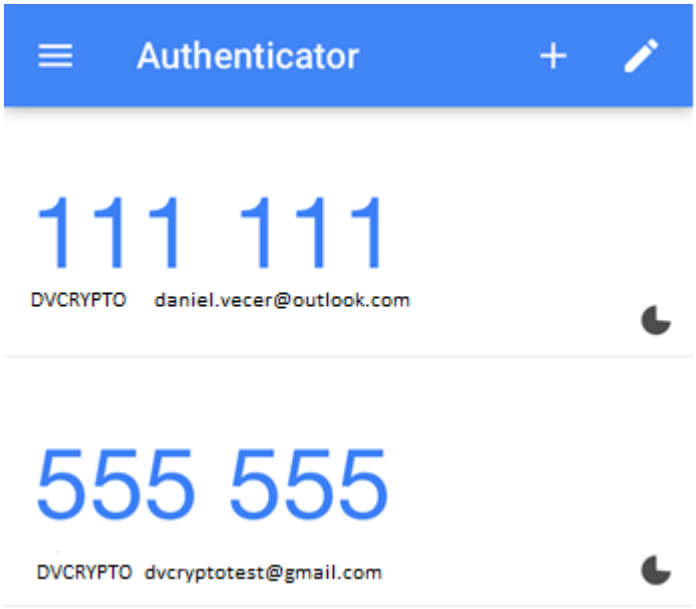
Registrační formulář používá automatickou validaci uživatelem vložených údajů. Všechny tyto údaje jsou ověřovány v reálném čase již během toho, když je uživatel zadává do příslušných textových polí. Následně je mu zobrazena indikace stavu – v případě validních dat se okraje textového vstupu zbarví zeleně, v případě neplatného vstupu pak červeně, tak jak je to znázorněno na obrázku 66. Pod samotným textovým polem se pak zobrazí doplňkové chybové hlášení poskytující upřesňující informace o tom, z jakého důvodu validace selhala. Odeslat údaje a provést tak registraci je možné až v momentě, kdy jsou splněny podmínky validace.

5.5 Nastavení vícefázového ověření



Obrázek 67: Nastavení vícefázového ověření uživatele

Přihlášený uživatel aplikace má možnost zapnout zabezpečení svého účtu s využitím vícefázového ověření skrze mobilní aplikaci Google Authenticator, která je bezplatně dostupná pro zařízení využívající operační systémy Android a iOS. Ve webovém rozhraní aplikace dojde při zapnutí k vygenerování QR kódu, který je možno naskenovat kamerou mobilního zařízení. Alternativou je pak opsání kódu pro manuální aktivaci vícefázové autentizace.



Obrázek 68: Ukázka bezpečnostních kódů v aplikaci Google Authenticator

Po zadání kódu aplikace automaticky generuje bezpečnostní kódy (každých 30 sekund je vytvořen nový kód a dojde k zneplatnění předchozích kódů). Při pokusu o přihlášení na web je po uživateli s aktivovanou vícefázovou autentizací požadováno opsání tohoto kódu. Tím je zajištěna podstatně vyšší úroveň zabezpečení uživatelského účtu – i kdyby útočník získal přihlašovací jméno a heslo, tak nemá možnost se přihlásit, pokud zároveň neovládá uživatelovo mobilní zařízení s aplikací Google Authenticator.

ZÁVĚR

Výsledkem teoretické části této diplomové práce je především popis technologií použitelných pro vývoj moderních webových aplikací, a to hlavně těch spojených s frameworkem .NET – primárně se tedy jedná o frameworky ASP.NET Core a Blazor WebAssembly. Práce pojednává o historii a možnostech využití těchto technologií v obecné rovině. V menší míře jsou pak v teoretické části analyzovány i další příbuzné technologie, které lze pro vývoj webových aplikací využít, jako je například databázový systém MongoDB a verzovací systém Git. Součástí teoretické části práce je i kapitola věnovaná základním principům fungování kryptoměn a možností, jak s nimi obchodovat.

Výstup praktické části práce pak lze členit na dvě části. První částí je samotný text práce, který poměrně detailně popisuje postup vývoje webové aplikace ve všech jejích fázích. Začíná definicí požadavků na aplikaci, na což následně navazuje kapitola věnovaná návrhu architektury a struktury celé aplikace. Následuje pak kapitola věnovaná realizaci aplikace, která především popisuje postup vývoje aplikace z programátorského hlediska. Součástí je mnoho ukázek kódu vyvíjené aplikace, u kterých je vždy popsáno, co daný kód zajišťuje. Poslední kapitola praktické části je poté věnována demonstraci aplikace, a to včetně screenshotů z aplikace samotné.

Druhotným výstupem praktické části práce je potom zdrojový kód a aplikace jako taková. Aplikace ve své současné podobě poskytuje základní možnosti pro monitorování trhů s kryptoměnami, k čemuž zpracovává data poskytovaná kryptoměnovými burzami. Je tedy nutné zmínit, že se nejedná o pouhou demo aplikaci, ale o aplikaci skutečně využitelnou, která uživateli poskytuje přehled o reálných datech. Architektura zdrojového kódu aplikace je vytvořena takovým způsobem, který umožňuje budoucí rozšíření aplikace o další funkcionalitu. Takové rozšíření by bylo možno provést bez větších zásahů do současné struktury aplikace a jejích částí, které již jsou v současné době plně funkční.

SEZNAM POUŽITÉ LITERATURY

[1] *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*. 4th Edition. Birmingham: Pack Publishing, 2019. ISBN 978-1788478120.

[2] ALLE, Mahesh. Code Execution Process. *C# Corner - Community of Software and Data Developers* [online]. Philadelphia, PA: C# Corner, 2021 [cit. 2021-02-22]. Dostupné z: https://www.c-sharpcorner.com/UploadFile/8911c4/code-execution-process/

[3] TODOROV, Tsvetomir. Understanding .NET Garbage Collection. *Telerik UI for ASP.NET AJAX, MVC, Core, Xamarin, Angular, HTML5 and jQuery* [online]. Sofia: Telerik, 2021 [cit. 2021-02-22]. Dostupné z: https://www.telerik.com/blogs/understanding-net-garbage-collection

[4] Microsoft .NET Framework Version History. *Execute Commands* [online]. [cit. 2021-02-22]. Dostupné z: https://executecommands.com/microsoft-net-framework-version-history/

[5] Microsoft .NET Core Versions History. *Execute Commands* [online]. [cit. 2021-02-22]. Dostupné z: https://executecommands.com/microsoft-net-core-versions/

[6] LANDER, Richard. Introducing .NET 5 | .NET Blog. *.NET Blog* [online]. Redmond: Microsoft, 2021 [cit. 2021-02-22]. Dostupné z: https://devblogs.microsoft.com/dotnet/introducing-net-5/

[7] ALBAHARI, Joseph a Ben ALBAHARI. *C# 7.0 in a nutshell*. 7th edition. Sebastopol: O'Reilly, 2018. ISBN 978-1491987650.

[8] Microsoft C# Version History. *Execute Commands* [online]. [cit. 2021-02-22]. Dostupné z: https://executecommands.com/csharp-version-history/

[9] *Professional C# 7 and .NET Core 2.0*. 11th edition. Indianapolis: Wrox, a Wiley Brand, 2018. ISBN 978-1119449270.

[10] TROELSEN, Andrew a Philip JAPIKSE. *Pro C# 9 with .NET 5 Foundational Principles and Practices in Programming*. Tenth Edition. New York: Apress, 2021. ISBN 978-1-4842-6939-8.

[11] *ASP.NET | Open-source web framework for .NET .NET | Free. Cross-platform. Open source* [online]. Redmond: Microsoft, 2021 [cit. 2021-02-22]. Dostupné z: https://dotnet.microsoft.com/apps/aspnet

[12] Common web application architectures | Microsoft Docs. *Microsoft Docs* [online]. Redmond: Microsoft, 2021 [cit. 2021-02-22]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

[13] ASP.NET MVC Architecture. *TutorialsTeacher - Learn Web Technologies* [online]. 2020 [cit. 2021-02-22]. Dostupné z: https://www.tutorialsteacher.com/mvc/mvc-architecture

[14] ASP.NET Core Blazor hosting models | Microsoft Docs. *Microsoft Docs* [online]. Redmond: Microsoft, 2021 [cit. 2021-02-22]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0

[15] MACIEJAK, David. WebAssembly 101: Bringing Bytecode to the Web. *Fortinet | Enteprise Security Without Compromise* [online]. Sunnyvale: Fortinet, 2021 [cit. 2021-02-22]. Dostupné z: https://www.fortinet.com/blog/threat-research/webassembly-101-bringing-bytecode-to-the-web

[16] HAAS, Andreas, Andreas ROSSBERG, Derek SCHUFF et al. Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2017, , 185-200. ISBN 9781450349888. Dostupné z: doi:10.1145/3062341.3062363

[17] SAINI, Ankita. *What is MongoDB - Working and Features - GeeksForGeeks* [online]. GeeksForGeeks, 2021 [cit. 2021-02-22]. Dostupné z: https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/

[18] PRIYA, Khushi. Binary JSON (BSON). *OpenGenus IQ: Learn Computer Science* [online]. OpenGenus IQ, 2021 [cit. 2021-02-22]. Dostupné z: https://iq.opengenus.org/binary-json/

[19] VOOR, Dillonde. MongoDB for DBAs: Introduction - CrocoDillon. *Blog - CrocoDillon* [online]. [cit. 2021-02-22]. Dostupné z: http://crocodillon.com/blog/mongodb-for-dbas-introduction

[20] Release Notes - MongoDB Manual. *Welcome to the MongoDB Documentation - MongoDB Documentation* [online]. New York: MongoDB Inc., 2021 [cit. 2021-02-22]. Dostupné z: https://docs.mongodb.com/manual/release-notes/

[21] Git - What is Git?. *Git* [online]. [cit. 2021-02-22]. Dostupné z: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

[22] KOTOPULOS, Filip. *Kryptoměny*. Zlín: Univerzita Tomáše Bati ve Zlíně, 2018, 58 s. Dostupné také z: http://hdl.handle.net/10563/44405. Bakalářská. Univerzita Tomáše Bati ve Zlíně. Fakulta aplikované informatiky, Ústav počítačových a komunikačních systémů. Vedoucí práce Matušů, Radek.

[23] Blockchain - Wikipedia. *Wikipedia, the free encyclopedia* [online]. San Francisco: Wikimedia Foundation, 2021 [cit. 2021-02-22]. Dostupné z: https://en.wikipedia.org/wiki/Blockchain

[24] What is Blockchain technology? - IBM Blockchain | IBM. *IBM - United States* [online]. Armonk: IBM, 2021 [cit. 2021-02-22]. Dostupné z: https://www.ibm.com/blockchain/what-is-blockchain

[25] CUMMINGS, Stephan. The Four Blockchain Generations. *The Capital - Medium* [online]. The Capital, 2021 [cit. 2021-02-23]. Dostupné z: https://medium.com/the-capital/the-four-blockchain-generations-5627ef666f3b

[26] WACKEROW, Paul a Sam RICHARDS. Consensus mechanisms | ethereum.org. *Home | ethereum.org* [online]. [cit. 2021-02-23]. Dostupné z: https://ethereum.org/en/developers/docs/consensus-mechanisms/

[27] Crypto Trading | What is Cryptocurrency Trading?. *Online Trading | Financial Trading | CFD and Forex Trading* [online]. London: IG Group, c2003-2021 [cit. 2021-02-23]. Dostupné z: https://www.ig.com/en/cryptocurrency-trading/what-is-cryptocurrency-trading-how-does-it-work

[28] MUELLER, Ken. Dependency Injection in ASP.NET Core 3.1. *Social Network for Programmers and Developers* [online]. Morioh, 2021 [cit. 2021-05-02]. Dostupné z: https://morioh.com/p/a3322a60d4d7

[29] Strongly typed SharePoint list operations using the repository pattern. *Bugfree Consulting* [online]. Bugfree Consulting, 2020 [cit. 2021-05-02]. Dostupné z:

https://bugfree.dk/blog/2012/12/02/strongly-typed-sharepoint-list-operations-using-the-repository-pattern

[30] Implement Repository & Unit of Work design patterns in .NET Core with practical examples [Part 1]. *Enlab Software | Software Outsourcing Company* [online]. Enlab Software, 2021 [cit. 2021-05-02]. Dostupné z: https://enlabsoftware.com/development/how-to-implement-repository-unit-of-work-design-patterns-in-dot-net-core-practical-examples-part-one.html

[31] Create a simple API gateway in ASP.NET Core. *Sudo Null company* [online]. Sudo Null company, 2019 [cit. 2021-05-02]. Dostupné z: https://sudonull.com/post/12845-Create-a-simple-API-gateway-in-ASPNET-Core

[32] Safety Cribs: JWT. *Https://sudonull.com/* [online]. Sudo Null Company, 2019 [cit. 2021-05-02]. Dostupné z: https://sudonull.com/post/26454-Security-Cheat-Sheets-JWT-Akribia-Blog

[33] OLIYNIK, Maxim. TOTP Algorithm Explained. *Protectimus Solutions Two-Factor Authentication Provider - Protectimus* [online]. Protectimus Solutions, 2021 [cit. 2021-05-02]. Dostupné z: https://www.protectimus.com/blog/totp-algorithm-explained/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

HTML	Hypertext Markup Language
CIL	Common Intermediate Language
CLR	Common Language Runtime
GC	Garbage Collection
OBDC	Open Database Connectivity
WPF	Windows Presentation Foundation
WCF	Windows Communication Foundation
AJAX	Asynchronous JavaScript and XML
XML	Extensible Markup Language
LINQ	Language Integrated Query
P2P	Peer-to-Peer
TPL	Task Parallel Library
DLR	Dynamic Language Runtime
DPI	Dots per inch
JIT	Just-in-time
UWP	Universal Windows Platform
ARM	Advanced RISC Machines
RISC	Reduced instruction set computer
IoT	Internet of Things
AOT	Ahead-of-time
SQL	Structured Query Language
DI	Dependency Injection
IoC	Inversion of Control
MVC	Model-view-controller
API	Application programming interface

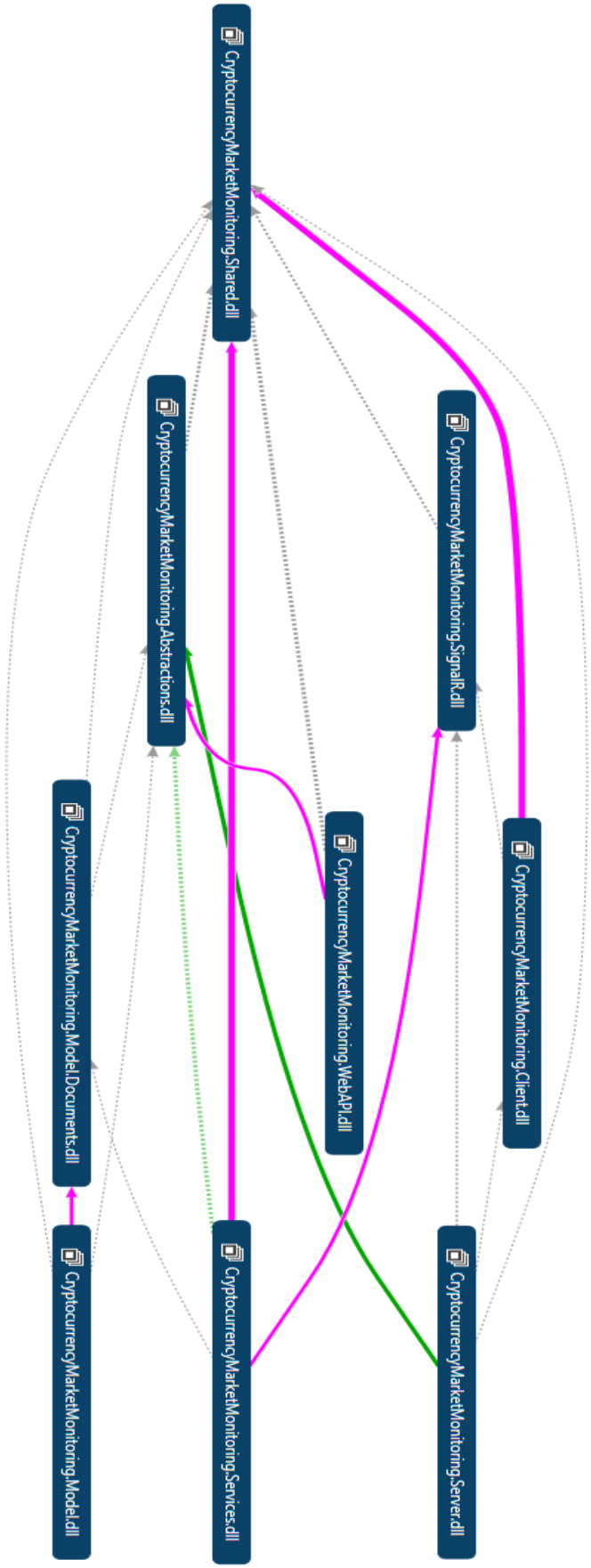
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
BSON	Binary JSON
TTL	Time to live
GNU	GNU's not Unix!
GNU GPL	GNU General Public License
BTC	Bitcoin
ETH	Ethereum
USD	United States dollar
CRUD	Create, read, update and delete
JWT	JSON Web Token
TOTP	Time-based One-Time Password
SPA	Single Page Application
MACD	Moving Average Convergence Divergence
RSI	Relative Strength Index
MA	Moving Averages
PNG	Portable Network Graphics
JPG	Joint Photographic Group
SVG	Scalable Vector Graphics
PDF	Portable Document Format

SEZNAM OBRÁZKŮ

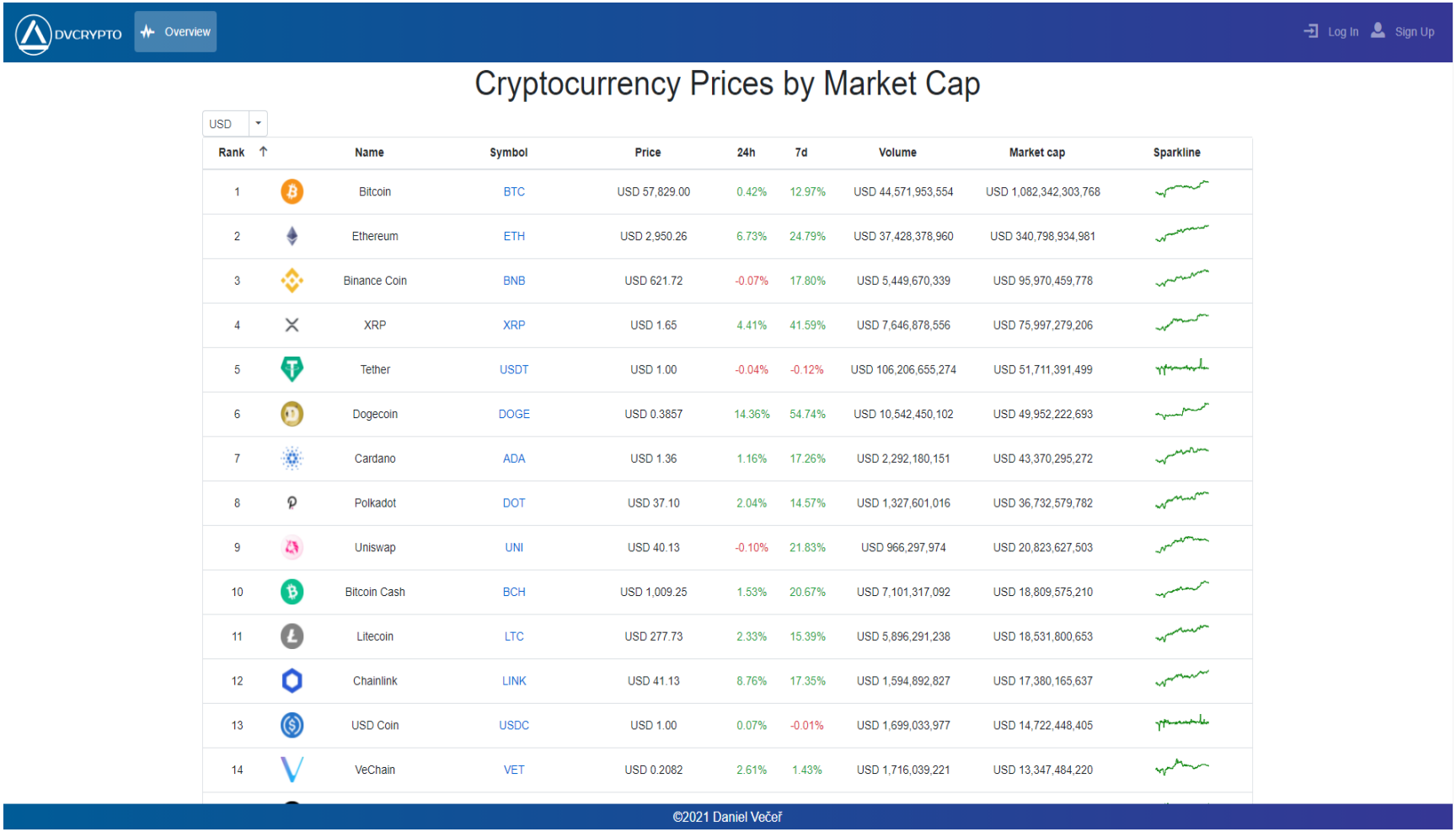
SEZNAM TABULEK

SEZNAM PŘÍLOH

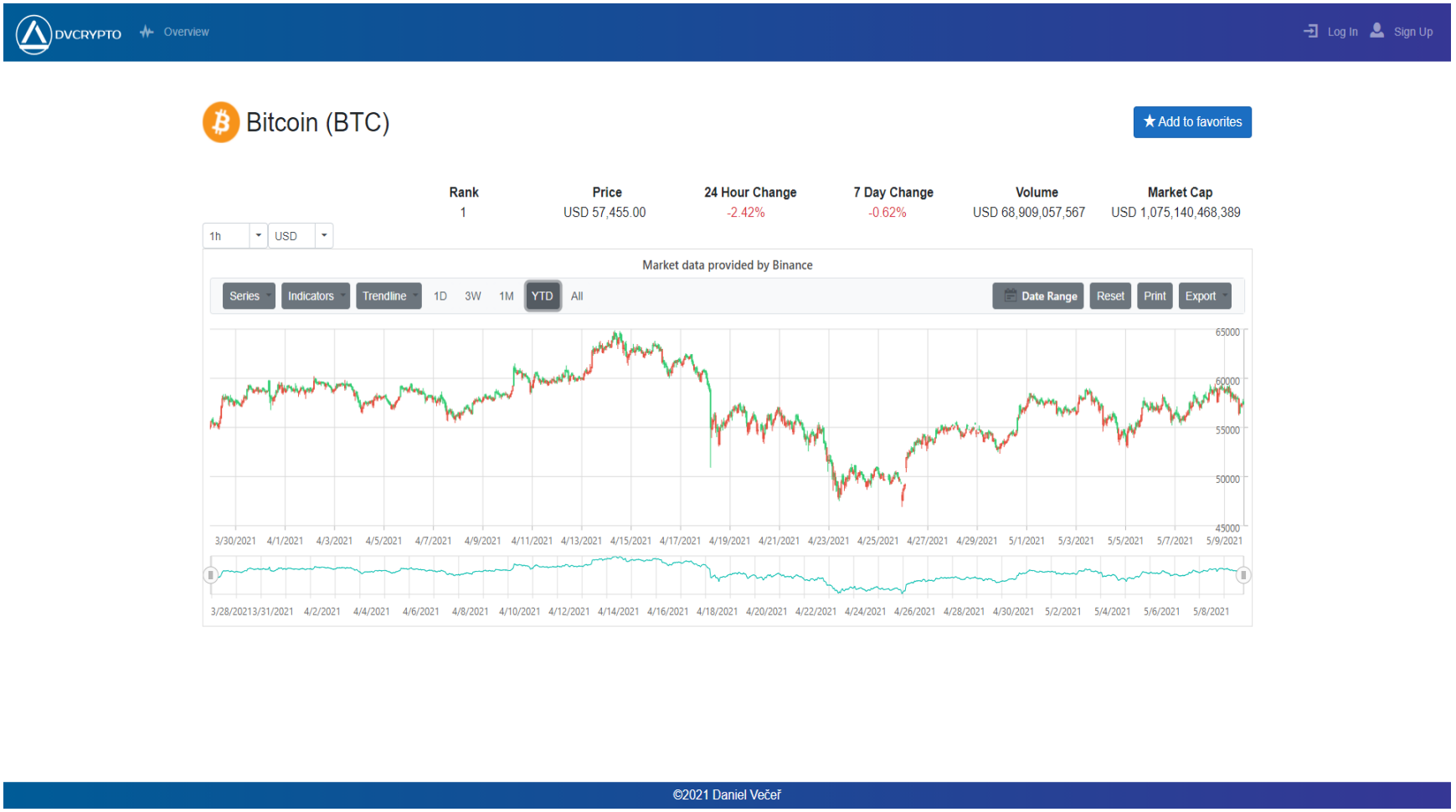
PŘÍLOHA P I: SCHÉMA REFERENCÍ PROJEKTU



PŘÍLOHA P II: HLAVNÍ STRÁNKA APLIKACE

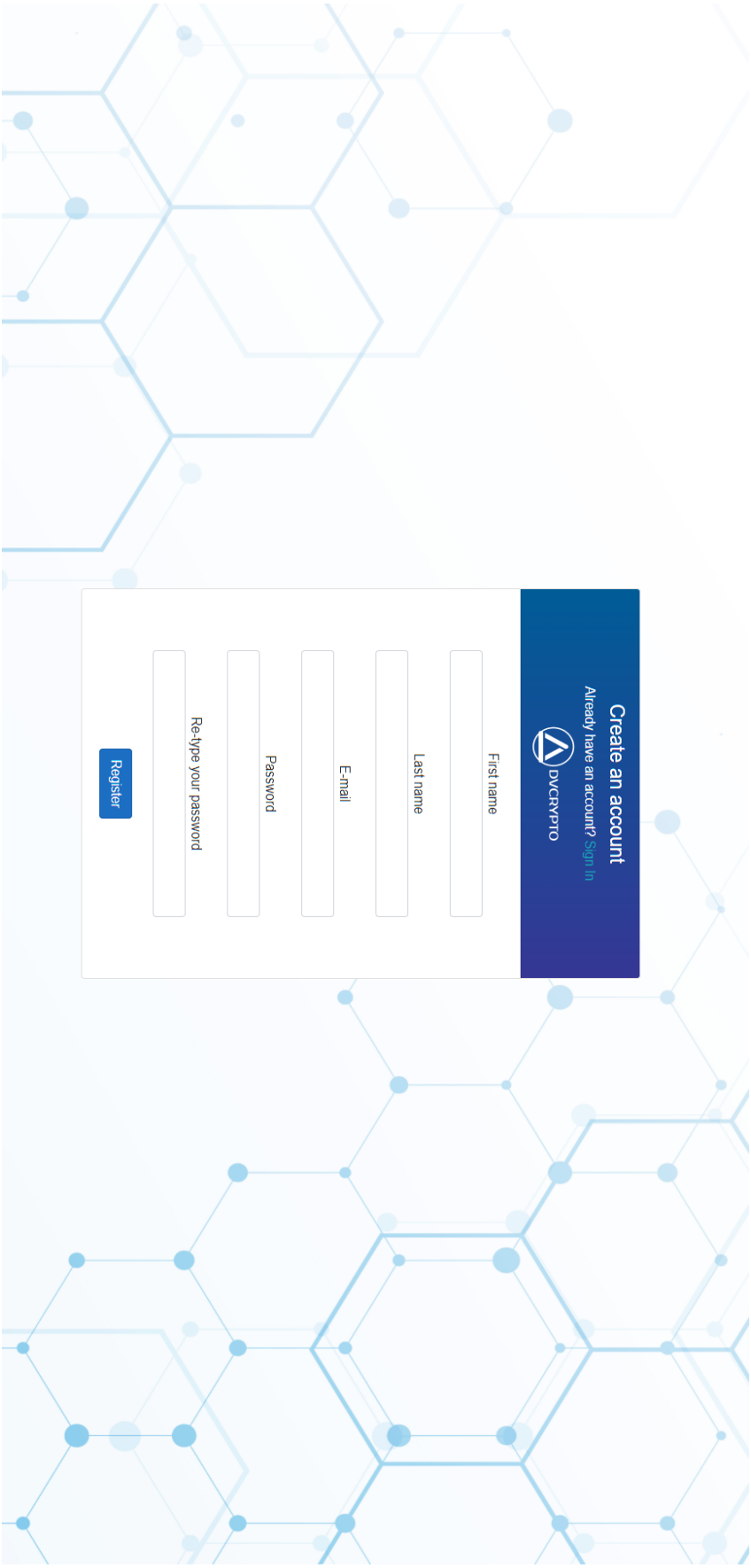


PŘÍLOHA P III: DETAIL KRYPTOMĚNY



PŘÍLOHA P IV: PŘIHLÁŠENÍ UŽIVATELE

PŘÍLOHA P V: REGISTRACE UŽIVATELE



PŘÍLOHA P VI: NASTAVENÍ VÍFEFÁZOVÉHO OVĚŘENÍ