TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Tvorba reportu v prostředí .NET core

Bakalářská práce

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

Autor práce: **Jan Prokorát**
Vedoucí práce: Ing. Jan Kraus Ph.D.

Liberec 2019

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Creating a .NET core Report

Bachelor thesis

Study programme:	B2646 – Information technology
Study branch:	1802R007 – Information technology
Author:	**Jan Prokorát**
Supervisor:	Ing. Jan Kraus Ph.D.

Liberec 2019

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání bakalářské práce

Tvorba reportu v prostředí .NET core

Jméno a příjmení:	**Jan Prokorát**
Osobní číslo:	M16000049
Studijní program:	B2646 Informační technologie
Studijní obor:	Informační technologie
Zadávající katedra:	Ústav mechatroniky a technické informatiky
Akademický rok:	**2018/2019**

Zásady pro vypracování:

1. Seznamte se s prostředím .NET core, s jeho omezeními a s možnostmi pro potřebu tvorby sestav (reportů) na jednotlivých podporovaných platformách.
2. Seznamte se s typickými veličinami, které se používají pro posouzení správnosti funkce běžného kompenzačního systému prostřednictvím vzdáleného dohledu.
3. Vyberte vhodné nástroje a s jejich pomocí v prostředí .NET core vytvořte ukázkovou aplikaci pro zpracování reálných dat.
4. Funkce vytvořené aplikace musí být v rámci prostředí .NET core přenositelné mezi různými podporovanými systémy a výstupy včetně grafických sestav by měly používat některý z běžných formátů.
5. Vyhodnoťte dosažené výsledky, shrňte klady a zápory použitého řešení a diskutujte možnosti dalšího rozvoje tématu.

Rozsah grafických prací: dle potřeby dokumentace

Rozsah pracovní zprávy: 30–40 stran

Forma zpracování práce: tištěná/elektronická

Seznam odborné literatury:

[1] ROTH, Daniel, Rick ANDERSON a Shaun LUTTIN, Introduction to ASP.NET Core [online]. Microsoft [cit. 2017-10-10]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/.

[2] Deploying the .NET Framework and Applications: .NET Framework 4.6 and 4.5 [online], [cit. 2015-10-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/6hbb4k3e(v=vs.110).aspx.

[3] ANDREWS, Douglas; BISHOP, Martin T.; WITTE, John F. Harmonic measurements, analysis, and power factor correction in a modern steel manufacturing facility. IEEE Transactions on Industry Applications, 1996, 32.3: 617-624.

Vedoucí práce: Ing. Jan Kraus, Ph.D.
Ústav mechatroniky a technické informatiky

Datum zadání práce: 10. října 2018

Předpokládaný termín odevzdání: 30. dubna 2019

L. S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci 10. října 2018

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Abstrakt

Cílem bakalářské práce je analyzovat komponenty prostředí .NET core. Představit omezení a možnosti při tvorbě sestav (reportů) na všech podporovaných platformách. Dále se seznámit s typickými veličinami při posuzování správnosti funkce běžného kompenzačního systému jalového výkonu prostřednictvím vzdáleného dohledu. Součástí praktické části bude webová MVC aplikace, která zpracovává konkrétní data, a následně z nich vytváří reporty o funkci kompenzačního systému.

Klíčová slova: .NET core, aplikace, report, platforma, MVC, jalový výkon, kompenzace, vzdálený dohled

Abstract

The goal of this bachelor thesis is to analyze components of .NET core environment. Introduce limitations and options while creating reports in all supported platforms. In addition, the practical part will be a web-based MVC application that processes specific data and then reports on a functional compensation system.

Keywords: .NET core, application, report, platform, MVC, reactive energy, compensation, remote surveillance

Poděkování

Rád bych poděkoval všem, kteří přispěli ke vzniku tohoto projektu.

Obsah

Seznam zkratek 11

Úvod **12**

1 Prostředí .NET core **14**
1.1 Vlastnosti 14
1.1.1 Přenositelnost na ostatní operační systémy 14
1.1.2 Dostupná rozšíření 14
1.1.3 Vysoký výkon 15
1.1.4 Rozšiřitelná mezipaměť 15
1.1.5 Sjednocené platformy MVC a Web API 15
1.2 Nástroje pro reporty 16
1.2.1 JavaScript pluginy 16
1.2.2 NuGet balíčky 17

2 Úvod do kompenzace jalového výkonu **19**
2.1 Definice jalového výkonu 19
2.2 Definice kompenzace 20
2.2.1 Průběh kompenzace 20
2.3 Sledované veličiny 21
2.3.1 Účiník 21
2.3.2 Regulační Odchylka 21
2.3.3 Odchylka jalového výkonu 22
2.3.4 Stykače 22
2.4 Důvody kompenzace 22
2.4.1 Elektrický důvod 22
2.4.2 Ekonomický důvod 22

3 Návrh a realizace řešení **24**
3.1 Volba operačního systému 24
3.2 Výběr typu aplikace 25
3.3 MVC webová aplikace 25
3.3.1 Model 26
3.3.2 View (Pohled) 29
3.3.3 Controller 30
3.4 Class diagram 32

3.5 Vložení elementů do pohledů 33
3.5.1 Chart.js 33
3.5.2 DevExtreme 34
3.6 Externí moduly 37
3.6.1 Chartjs-plugin-labels.js 37
3.6.2 Chartjs-plugin-zoom.js 38
3.6.3 Hammer.js 38
3.6.4 Progressbar.js 38
3.6.5 Momentjs 39
3.7 Export reportu 40
3.8 Vytvoření spustitelného souboru 41

4 Vyhodnocení řešení **42**

Závěr **44**

Literatura **45**

Přílohy **48**

Seznam obrázků

1.1 Porovnání rychlosti platforem . . . 15
1.2 Graf necele vyexportovaný přes konec stránky . . . 17

2.1 Trojúhelník výkonů . . . 19
2.2 Odebírání elektrické energie spotřebičem . . . 20

3.1 Odkazy jednotlivých komponent v modelu MVC . . . 25
3.2 Výpočet a ukládání regulační odchylky . . . 26
3.3 Výpočet a ukládání odchylky od účiníku . . . 27
3.4 Formátování čísla podle typu operačního systému . . . 28
3.5 Volání vykreslení obsahu do požadované sekce . . . 29
3.6 Odkaz na umístění šablony pohledu nad příkladem převodu mezi jazyky C# a JavaScript . . . 30
3.7 Vytvoření instance modelu v controlleru . . . 31
3.8 Implementace controlleru vykreslující pohled a ovládající graf . . . 31
3.9 Class diagram programu . . . 32
3.10 Příklad přenosu dat do jazyka JavaScript . . . 33
3.11 Příklad implementace Chart.js grafu . . . 34
3.12 Příklad implementace DevExtreme FileUploaderu . . . 35
3.13 Příklad implementace DevExtreme grafu . . . 35
3.14 Příklad implementace Range Selectoru . . . 36
3.15 Příklad implementace DevExtreme tabulky . . . 36
3.16 Implementace zobrazení popisů křivek . . . 37
3.17 Výsledné zobrazení popisů křivek . . . 37
3.18 Implementace přibližování a pohybu v grafu . . . 38
3.19 Implementace kruhového progressbaru . . . 39
3.20 Výsledné zobrazení dialogu . . . 39
3.21 Implementace momentu ve chtěném formátu . . . 39
3.22 Implementace exportu do formátu JPEG . . . 40

4.1 Příklad zobrazení grafu v Chart.js . . . 43
4.2 Příklad zobrazení grafu v DevExtreme . . . 43

Seznam tabulek

2.1 Rozmezí a ceny za nedodržení rozmezí odchylky účiníku 23

Seznam zkratek

TUL	Technická univerzita v Liberci
FM	Fakulta mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci
MVC	Model-view-controller
ARM	Advanced RISC Machines
REST	Representational State Transfer
VA	Voltampér
VAr	Voltampér reaktanční
S	Zdánlivý výkon [VA]
Q	Jalový výkon [VAr]
P	Činný výkon [W]
I	Proud [A]
I_c	Činný proud [A]
I_j	Jalový proud [A]
U	Napětí kondenzátoru [V]
Qfh	Regulační odchylka
Pfh	Činný výkon základní harmonické složky
IDE	Integrated Development Environment
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
CSHTML	C sharp Hypertext Markup Language
CSS	Cascading Style Sheets
NPM	Node Package Manager

Úvod

Vzhledem k tomu, že cena i spotřeba elektřiny v moderní době neustále roste, je čím dál více v zájmu lidí i firem své údaje sledovat a kompenzovat jalovou energii. Zmiňuje-li se kompenzace jalového výkonu, myslí se optimalizace zdánlivého výkonu natolik, aby se činná energie přenášela sítí za co nejlepších podmínek. V energetice jde o poměrně známý a běžný problém. Jde o snahu dodávat jalovou složku elektrické energie do spotřebiče z připojeného kondenzátoru místo toho, aby se dodávala společně s činnou složkou přes síť. Tím se sníží množství přenášené energie v síti, a sníží se ztráty. V konečném převodu na ušetřené peníze může účinná a efektivní kompenzace znamenat rozdíl mezi bankrotující a prosperující firmou.

Jak ale někdo, například manažer firmy, bez hlubších fyzikálních znalostí pozná z výstupu kompenzačního zařízení, zda přístroj kompenzuje správně, zda nedochází k nadměrnému odběru energie, zda nedochází k přepětí atp.? Na kladené otázky se bude snažit odpovědět tato práce vytvořením aplikace na platformě ASP.NET Core, která zpracovává konkrétní data ze sledovacího zařízení a vytváří report o stavu sledovaných veličin. Programování na open-source platformě .NET Core se ve vývojářském světě stává čím dál populárnější. Prostředí sjednocuje platformy Web API a .NET framework, čímž poskytuje široké množství nástrojů pro tvorbu aplikací. Umožňuje vytvářet webové služby, aplikace internetu věcí (LoT), mobilní aplikace i desktopové programy. To vše zvládá na všech nejrozšířenějších operačních systémech. Platforma je modulární, distribuuje jednotlivé nástroje v balíčcích, které si každý vývojář sám do svého projektu přidá dle potřeby. Tímto postupem se platforma stala paměťově odlehčenější a kompaktnější.

Bakalářská práce si stanovuje několik základních cílů. Nejprve si dává za úkol seznámit čtenáře s vývojářskou platformou .NET Core, přesněji s využitelností platformy při tvorbě reportů v běžných formátech, jako jsou Excel či PDF. Důležitou roli sehraje optimalizace platformy a jejích nástrojů na různých platformách. V rámci první kapitoly také představí několik způsobů a nástrojů k vytváření reportů v rámci .NET Core platformy. Druhým cílem je seznámení čtenáře s jalovým výkonem a jeho kompenzací, tedy představení základních pojmů a vysvětlení průběhu kompenzace. Dále pak objasnění důležitých důvodů pro kompenzaci a vybrání relevantních sledovaných veličin ze sledovacího zařízení kompenzačního systému.

V následující části se stává cílem spojení poznatků z předchozích dvou kapitol a vytvoření ukázkového programu v prostředí .NET Core, který zpracovává data o sledovaných veličinách, a následně generuje report o stavu kompenzačního zařízení. Zmíní se tedy důvod výběru jednoho či druhého operačního systému a typu aplikace, popíše se struktura programu i použité doplňující balíčky.

Závěrečným cílem je shrnutí dosažených výsledků a postupu řešení. Vyhodnotí se zde, zda byla použitá řešení efektivní, zda došlo nebo nedošlo během řešení práce k jakýmkoli problémům nebo jestli se nedosáhlo neočekávaných či překvapujících výsledků. Také se prodiskutuje možnost využití programu v praxi společně s možností budoucího rozšíření aplikace o další užitečné funkce.

1 Prostředí .NET core

Tato kapitola obsahuje seznámení s prostředím ASP.NET core či zkráceně .NET core. Představuje jeho vlastnosti a využitelnost při tvorbě reportů, dále také objasňuje využitelnost na jednotlivých počítačových platformách.

Pod názvem .NET core se skrývá open-source platforma pro vytváření aplikací napříč všemi operačními systémy. Hlavní myšlenka spočívá ve vytváření aplikací, které poběží v podstatě na jakémkoli zařízení bez ohledu na nainstalovaný operační systém. Používá se převážně k implementaci MVC aplikací (viz kapitolu 3.3). Autorem nástroje je americký softwarový gigant Microsoft.

1.1 Vlastnosti

1.1.1 Přenositelnost na ostatní operační systémy

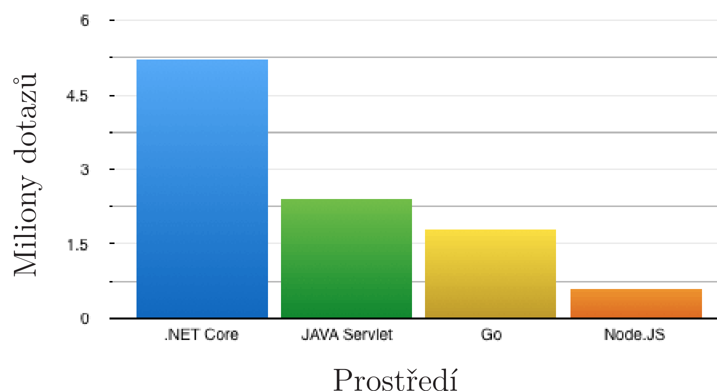
Platforma .NET core umožňuje zaujmout mnohem širší spektrum vývojářů softwaru díky tomu, že je navržená jako multiplatformní. Poskytuje podporu pro Windows, MacOS, některé distribuce systému Linux, ale i pro menší systémy, jako jsou Raspberry PI či ARM. Také nerozlišuje architektury x64 a x32, pokaždé se chová stejně. K rozšíření přispívá i Microsoft spoustou přehledných návodů a vysvětlivek v dokumentaci na svých stránkách. Umožňuje vývoj klasických programů, webových či mobilních aplikací nebo LoT systémů. Platforma je stále ve vývoji, čímž přislibuje budoucí přibývání nástrojů i knihoven. To je žádoucí hlavně ve verzích pro MacOS a Linux, které zatím ani zdaleka neobsahují tolik nástrojů, jako verze pro Windows.

1.1.2 Dostupná rozšíření

Veškeré .NET core aplikace se dají rozšířit o moduly, které nejsou základní součástí platformy. Těmto balíčkům se říká NuGet a obsahují spoustu rozličných knihoven pro implementaci všech možných funkcionalit programu, které jen vývojáře můžou napadnout. Některé jsou zdarma ke stažení, jiné vyvíjejí specializované firmy a je tedy nutno za ně platit. I v této práci byl pro implementaci funkce vytváření reportů použit balíček NuGet k tomu určený, protože čistý .NET core bohužel žádné nástroje pro efektivní tvorbu a export reportů nenabízí. Více o konkrétních nástrojích pro tvorbu reportů bude řečeno v kapitole 1.2, popřípadě o konkrétní implemmentaci v tomto projektu v kapitole 3.7.

1.1.3 Vysoký výkon

Prostředí .NET core je považováno za jednu z nejvýkonnějších a nejrychlejších platforem vůbec. Microsoft toho dosáhl implementací platformově univerzálního webového serveru Kestrel, speciálně pro potřeby .NET aplikací. Při vývoji také není nutné mít nainstalovaný žádný server, ať už lokální nebo hoštěný na webu. Kestrel pracuje s daty na pozadí, zatímco uživatel provádí jiné operace. Tomuto postupu se říká asynchronní zpracování dat, a výrazně zvyšuje množství zpracovaných operací oproti ostatním. O jak velký rozdíl se jedná, dokládá i graf přiložený níže (obr. 1.1). Vyplatí se tedy nástroj použít při vývoji aplikací, které jsou výkonově náročné.



Obrázek 1.1: Porovnání rychlosti zpracování dotazů za jednu sekundu oproti jiným nástrojům pro tvorbu webových aplikací
Zdroj: https://stackify.com/asp-net-core-features/

1.1.4 Rozšiřitelná mezipaměť

Funkce rozšiřitelné mezipaměti úzce souvisí s předchozím bodem. Umožňuje ukládat výstup vygenerovaný stránkou do mezipaměti, aby mohl být v případě potřeby znovu k dispozici. ASP.NET core ukládá ta data, která nejsou často aktualizovaná, aby nezatěžovala procesor opětovným výpočtem. Díky této funkci může vytvořený program data rychleji načítat a zároveň má i prostor pro vykonávání důležitějších nebo akutnějších operací.

1.1.5 Sjednocené platformy MVC a Web API

Před příchodem .NET Core vývojáři nejčastěji používali MVC a Web API platformy. MVC byla přizpůsobena pro vytváření webových aplikací, které sloužily jako HTML. Web API byl naproti tomu navržen tak, aby vytvořil služby REST pomocí JSON nebo XML. S rozhraním .NET Core došlo ke sloučení platforem, kdy se dle potřeby využívá obojího. Oba nástroje měly vždy mnoho společných prvků, a nainstalováním obojího došlo mnohokrát k duplicitě dat. Kombinace nástrojů tedy vedla k výraznému zjednodušení.

1.2 Nástroje pro reporty

Jak už bylo řečeno výše v bodě 1.1.2, .NET core nenabízí žádný zabudovaný nástroj, kterým by bylo možné vytvářet reporty. Z toho důvodu přišlo několik vývojářských firem s NuGet moduly, které umožňují vytvářet reporty ze zpracovávaných dat, a které se následně dají exportovat do některého z běžných formátů, jako jsou PDF, Word, Excel nebo JPEG.

Všechny NuGet balíčky se vyznačují tím, že obsahují návrhovou stránku s širokou paletou elementů, které vývojář může dle libosti skládat dohromady a tím vytvářet žádoucí návrh reportu. Rozdíly mezi jednotlivými moduly spočívají v intuitivnosti, jednoduchosti, rozsahem funkcí a v ceně. Cena samozřejmě závisí na rozsahu funkcí, ale i tak se ceny za licence pohybují v řádu tisíců dolarů. Naštěstí všechny firmy nabízejí třicetidenní trial verzi postačující potřebám tohoto projektu. Protože se ceny pohybují opravdu vysoko, na webových stránkách jednotlivých výrobců se dají najít on-line dema, která si vývojář může vyzkoušet před tím, než se rozhodne pro stažení či koupi produktu.

Následující body obsahují seznámení s několika metodami či moduly pro vytváření reportů, o kterých se uvažovalo jako o kandidátech pro implementaci v praktické části bakalářské práce. Nejprve je však třeba upozornit, že důvody výběru jednoho nástroje na úkor jiných jsou čistě subjektivní a můžou se s každým uživatelem lišit.

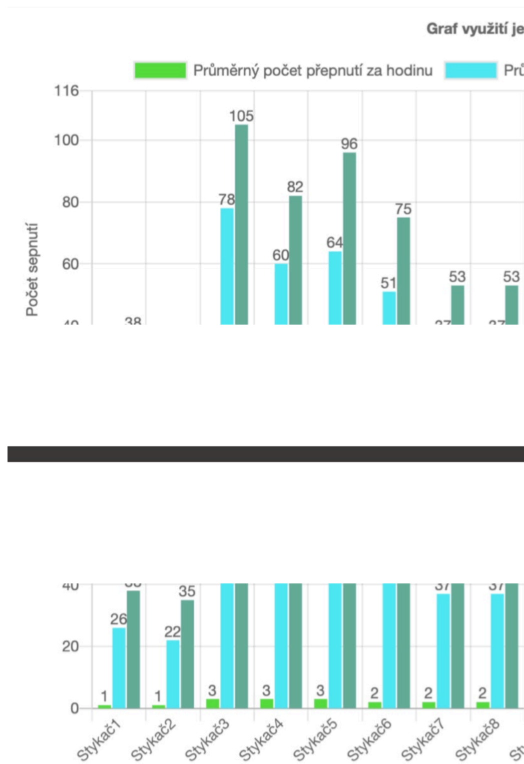
1.2.1 JavaScript pluginy

První možnost, jak vytvořit v ASP.NET core aplikaci report, se jeví vytvořit HTML pohled (popsán v kapitole 3.3.2), jenž se poté vyexportuje pomocí vhodného javasctiptového pluginu do formátu PDF. Šlo by o klasickou webovou stránku, kam by se pomocí značkovacího jazyka HTML vkládaly jednotlivé součásti. Práci s daty by zajišťoval JavaScript. Javascriptové moduly jsou volně k dostání na internetu, stačí je jen stáhnout a uložit do složky k právě vyvíjenému projektu nebo je lze nainstalovat pro dostupnost ve všech případných projektech.

K exportu HTML stránek je k dispozici například javascriptový plugin jménem Html2pdf. Funguje tím způsobem, že převede obsah zvoleného HTML tagu na formát PDF. Z hlediska implementace i následné manipulace se zobrazovanými daty tento postup působí neefektivně a značně složitě, protože uživatel musí například sám nastavovat šířku zobrazované stránky, aby se případné grafy do požadovaného formátu stránky vešly. Dále také plugin nenabízí žádnou možnost si elementy v reportu jakkoli rozvrhnout a napozicovat. Příklad toho, jak špatně může tento postup dopadnout, lze vidět na obrázku 1.2.

Kromě Html2pdf existují samozřejmě i další javascriptové moduly určené k exportu dat. Za zmínku stojí například PDF.js, který se odlišuje tím, že kromě exportu

HTML může fungovat i opačně, tedy že s jeho pomocí lze vytvořit aplikace načítající a zobrazující obsah již existujícího PDF dokumentu. Jako další zajímavou a propracovanou alternativu lze využít FileSaver.js určený čistě k ukládání obsahu HTML stránek do formátu TXT, JPEG aj. Trpí však podobným nedostatkem jako předešlé zmíněné moduly, nelze jakkoli napozicovat či upravit vytvářený dokument.



Obrázek 1.2: Graf necele vyexportovaný přes konec stránky

1.2.2 NuGet balíčky

Alternativní řešení pro ty, kteří neovládají javascript, nabízejí NuGet balíčky určené k vytváření reportů. Jsou to dostahovatelná rozšíření umožňující přidání do projektu funkce, které .NET neobsahuje v základní verzi při instalaci. V případě reportů jde o většinou placené produkty firem, specializují se přímo na vývoj daného softwaru. Některé produkty fungují jako samostatné aplikace, které doplňují moduly pro vývojová prostředí implementovatelné do jednotlivých projektů, jiné pak existují pouze jako stažitelné moduly. Jedním z nejrozšířenějších a nejpopulárnějších nástrojů k implementaci reportů je DevExpress. Disponuje přehlednou dokumentací se spoustou instruktážních videí či dem. Díky popularitě lze také relativně snadno najít podporu na programátorských fórech typu stack overflow. Disponuje prvky pro práci na webových aplikacích na všech běžných platformách (ASP.NET WebForms, ASP.NET MVC a Core, Bootstrap WebForms, HTML/JS, React a jQuery).

Řadí se k těm produktům, jež nejsou k dispozici jako samostatné aplikace a lze s nimi pracovat pouze integrací do vývojového prostředí. DevExpress do něho přidává designér s širokou paletou nástrojů pro tvorbu reportů. Vývojáři stačí propojit datovou strukturu projektu s designérem, aby měl odkud čerpat data. Pak už jen stačí jednotlivé elementy vkládat do generované stránky. Samotný DevExpres je k dispozici pouze na Windows. K vývoji na MacOS a Linux vytvořila společnost stojící za DevExpressem modul jménem DevExtreme. Je dostupný i na Windows v rámci DevExpress balíku, samostatně ale zatím na ostatní operační systémy nenabízí tolik funkcí jako mateřský produkt na Windows. I tak se však těší mezi vývojáři značné popularitě. Na ostatní platformy mimo Windows existuje ve formě javascriptových pluginů (viz kapitolu 1.2.1).

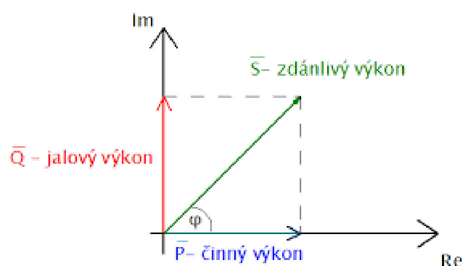
Na druhé straně barikády mezi samostatnými aplikacemi stojí Stimulsoft Reports. Jedná se o aplikaci umožňující samotnému uživateli sestavit si v samostatném okně zobrazovaný report poté, co si do programu načte zpracovávaná data, či aplikaci propojí s databází. Kromě této funkce také disponuje NuGet modulem pro implementaci do vývojářského prostředí. K dispozici je na Windows, MacOS i Linux. Podobných nástrojů, jako jsou DevExpress a Stimulsoft, existuje samozřejmě více. Mají podobné vlastnosti i design a liší se pouze v detailech. Aby se zde ale neopakovaly dokola víceméně stejné informace, zmíní už jen jménem další zajímavé aplikace, což jsou Telerik Reporting, Microsoft Report Viewer a Syncfusion.

2 Úvod do kompenzace jalového výkonu

Kapitola pojednává o jalovém výkonu a jeho kompenzaci. Vymezuje pojem jalový výkon stejně tak objasňuje pojem, důvody a důležitost kompenzace. Následně jsou představeny vhodné veličiny důležité při samotném sledování kompenzace. Výklad je v principu zaměřen na oblast elektrických sítí distribučních, ze kterých se napájí objekty určené ke kompenzaci a oblast problematiky kompenzace je zde nejvíce názorná.

2.1 Definice jalového výkonu

Jalový výkon tvoří spolu s činným výkonem složku tzv. zdánlivého elektrického výkonu. Značí se Q a jeho jednotkou je VAr – voltampér reaktanční. Spotřebovávají ho přístroje založené na induktivním či kapacitním principu. Mezi takové se řadí například elektrické motory, svářečky, zářivky apod. Tato energie je sice nutná ke správnému provozu přístroje, není však zužitkována v elektrické rozvodné síti, a proto se v ní považuje za nežádoucí. Jalový výkon se Dodá do spotřebiče ze sítě společně s činnou energií. Zatímco tu činnou spotřebuje přístroj při výkonu své práce, jalovou potřebuje pro tvorbu elektromagnetických polí (např. u transformátorů a motorů pro indukování napětí). Jalová energie se spotřebovává jen v místě připojení spotřebičů a v rozvodné síti zůstává nevyužita. Dle rozmístění jednotlivých spotřebičů se v síti přelévá tam, kde je jí nedostatek. Z toho důvodu se také někdy označuje jako „fluktuační". V souvislosti s jalovým Q [VAr] a činným P [W] výkonem se běžně uvádí i výkon zdánlivý. Značí se S a jeho jednotkou je VA – voltampér. Jde o zdánlivě celkový výkon potřebný pro chod napojeného induktivního spotřebiče. Lze ho vypočítat vektorovým součinem činného a jalového výkonu. Znázorněn je na přiloženém Obr. 2.1 níže.



Obrázek 2.1: Trojúhelník výkonů

$$S = \sqrt{P^2 + Q^2} \tag{2.1}$$

$$Q = P \cdot \tan(\phi) \tag{2.2}$$

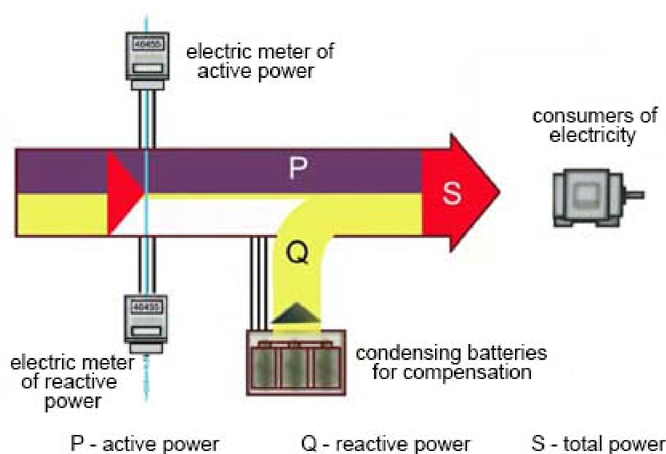
$$P = \frac{Q}{\tan(\phi)} \tag{2.3}$$

2.2 Definice kompenzace

V elektrických distribučních sítích bývají zapojené zejména spotřebiče indukčního charakteru (jedná se vesměs o podnikatelské objekty). Při přenosu elektrického proudu do stroje se činná a jalová energie sčítají, čímž zatěžují rozvodnou síť. Aby se tedy zabránilo zbytečně velkému zatížení sítě, její provozovatel motivuje odběratele, aby co nejvíce jalové energie přicházelo do spotřebiče jinudy, než právě skrz síť. Toho se dociluje paralelním připojením cívky či kondenzátoru co nejblíže ke spotřebiči, odkud se dodává jalový výkon s opačnou fází. Zde se jalová energie uchovává, a dle potřeby přesouvá do spotřebičů.

2.2.1 Průběh kompenzace

Odběr energie spotřebičem se dá rozdělit na tři kroky. V prvním nedochází k žádné kompenzaci, protože se veškerá energie vyměňuje mezi výrobcem elektřiny, a spotřebičem. Zde ještě není v kondenzátoru žádná nahromaděná jalová energie, kterou by mohl poskytnout a odlehčit tak síti. V druhém kroku se již část jalového výkonu dodává z kondenzátorů, stále však ještě dochází k dodávání jalové energie společně s činnou po síti, jde tedy o kompenzaci částečnou. Názorný průběh znázorněn na obrázku 2.2. K úplné kompenzaci dochází v posledním kroku, když už jalová energie kmitá pouze mezi kondenzátory a spotřebičem, a po síti se dodává pouze energie činná. V tomto stavu se poměr mezi činnou a jalovou energií (viz kapitolu 2.3.1) rovná jedné. K takovému případu ale v praxi téměř nedochází, protože by mohlo dojít k překompenzování sítě do kapacitní oblasti, a tedy i ke zvyšování napětí. Z tohoto důvodu se standardně kompenzuje na hodnoty *0,97* či *0,98*, zůstává se tedy u druhého kroku kompenzace.



Obrázek 2.2: Znázornění odebírání elektrické energie spotřebičem
Zdroj: http://sofelektro.rs/kompenzacija-reaktivne-snage/?lang=en

2.3 Sledované veličiny

Následující body obsahují zmínky o veličinách, které je žádoucí sledovat při kompenzaci jalového výkonu, a které budou zpracovány v praktické části bakalářské práce.

2.3.1 Účiník

Vedle vztahů mezi jednotlivými výkony si lze na Obr. 2.1 ještě všimnout jedné veličiny. Nazývá se účiník resp. cos( ), a vyjadřuje podíl mezi činným a zdánlivým výkonem (viz Obr. 2.4), a jde o jeden z nejdůležitějších ukazatelů při sledování kvality odběru energie. Hodnota účiníku se vždy pohybuje v rozsahu od nuly do jedné. Pokud se účiník rovná jedné, odběr je celý tvořený činným výkonem. Naopak nulový účiník znamená pouze jalový výkon. Pokud nastane případ, kdy jsou nízké hodnoty účiníku, znamená to, že dochází k vyšším ztrátám energie.

$$\cos(\phi) = \frac{P}{S} \tag{2.4}$$

kromě obecného vzorce pro výpočet účiníku je v tomto projektu žádoucí znát ještě vzorec pro výpočet tzv. trojfázového účiníku základní harmonické složky. Základní harmonická složka je hodnota jalového výkonu v síti skládající se ze tří cívek či kondenzátorů. Pro výpočet tohoto účiníku je žádoucí se vyhnout hodnotě zdánlivého výkonu. Zároveň se obvykle vypočítává průměrná hodnota za sledované období. Proto se jako lepší vzorec jeví rovnice 2.5.

$$\sum \cos(\phi) = \cos(\arctan(\frac{\sum Q}{\sum P})) \tag{2.5}$$

Jak už bylo řečeno v kapitole 2.2.1 o průběhu komepnzace, účiník $\cos(\phi) = 1$ se považuje za nebezpečnou a téměř nereálnou hodnotu. Z toho důvodu se běžně povoluje odchylka. Rozmezí této odchylky udává Energetický regulační úřad podle § 2c zákona č. 265/1991 Sb., o působnosti orgánů České republiky v oblasti cen v energetických odvětvích na 0 - 0,05 Kvar.

2.3.2 Regulační Odchylka

Nejdůležitější veličinou při regulaci účiníku zmíněné na konci minulé podkapitoly je regulační odchylka. S její pomocí lze zjistit přebývající část základní harmonické složky tj. jalové energie v elektrické síti, kterou je nutné vykompenzovat, aby se dosáhlo požadovaného optimálního účiníku. Hodnota může vyjít kladná, kdy má induktivní charakter, a regulátor připojil kondenzátory odpovídajícího výkonu. Na druhé straně může vyjít záporná, tedy kapacitní. V takovém případě musí regulátor připojit kompenzační tlumivky. Vzorec pro výpočet regulační odchylky vypadá takto:

$$\cos(\phi)_{Odchylka} = \cos(\phi)_{Ocekavany} - \cos(\phi)_{Skutecny} \tag{2.6}$$

2.3.3 Odchylka jalového výkonu

Pomocí zdánlivého výkonu nebo účiníku lze matematicky zjistit, kolik jalové energie spotřebič teoreticky potřebuje ke svému chodu. Prakticky se však stává, že díky ztrátám, špatnému výpočtu či jinému faktoru se skutečně dodá jalové energie více nebo naopak méně. V těchto situacích se pak sleduje, o kolik se skutečná hodnota od očekávané jalové energie odchyluje.

$$Q_{Odchylka} = Q_{Ocekavamy} - Q_{Skutecny} \tag{2.7}$$

2.3.4 Stykače

Stykače slouží jako spínací přístroje určené pro časté spínání kondenzátorů. Každý stykač je zásobený jalovou energií, kterou v případě sepnutí dodá do spotřebiče. Pokud poptávaná hodnota jalové energie přesahuje množství obsažené v jednom stykači, může sepnout více stykačů najednou, aby suma jejich výkonů dorovnala požadované množství, a uspokojila poptávku na jalovou energii. Protože jde o mechanickou součástku, sledují se nejrůznější doby provozu a počty sepnutí. S přibývajícími počty těchto veličin se stykače opotřebovávají, proto je potřeba je sledovat.

2.4 Důvody kompenzace

2.4.1 Elektrický důvod

Bez kondenzátorů či jiných zásobníků jalové energie se zvyšuje množství elektrické energie vedené po síti, čímž se síť více zatěžuje. Dále dojde ke snížení tepelných ztrát. Pokud se při přenosu zdánlivého elektrického proudu podaří snížit jalovou složku, sníží se i celkové přenášené množství, čímž poklesnou ztráty. Vztah je znázorněn ve vzorci 2.8. Snížením přenášeného zdánlivého výkonu se sníží i úbytek přenosového napětí v síti.

$$P = R \cdot I^2 = R \cdot (I_c^2 + I_j^2) \tag{2.8}$$

2.4.2 Ekonomický důvod

Předepsané hodnoty účiníku se dle zákona pohybují v rozmezí od 0,95 - 1. Jakkoli velké nedodržení daného rozmezí je penalizováno, a odběratel musí zaplatit svému distributorovi elektrické energie nemalé pokuty. Dále případné přetěžování elektrické sítě může zkracovat její životnost, a majiteli se může stát, že bude muset častěji měnit staré obvody za nové. Vyšší spotřeba zmíněná v minulé podkapitole také samozřejmě úměrně zvyšuje cenu za celkové odebrané množství. Do kompenzačního zařízení se vyplatí investovat, protože návratnost této investice se pohybuje ve velice krátkém čase.

Pokuty za odchýlení od účiníku i rozmezí tolerované odchylky se v různých světových státech liší. Tímto tématem se zabýval Jacques Peronnet, strategický ředitel normalizace ve společnosti Schneider-Electric [1], který získal a publikoval informace o účiníku v šestnácti světových zemích. Ve svém článku tvrdí [15], že účtování jalového výkonu při fakturaci elektřiny je tou nejlepší cestou k postrčení kompenzace jalové energie vpřed, protože nabízí rychlou návratnost investice. Na jeho zjištěná data lze nahlédnout v tabulce 2.1. Bohužel se nepodařilo nalézt novější data, než z roku 2009, i dnes by však údaje měly být relevantní, protože i s jakoukoli úpravou se částky či velikosti odchylky obvykle nijak razantně nemění.

Tabulka 2.1: Rozmezí a ceny za nedodržení rozmezí odchylky účiníku

Seznam zemí			
Země	Povolený účiník	Cena za odchýlení / 1 Kvar	Komentář
Austrálie	0.9 – 0.95	10 $	-
Česko	0.95 – 1	0.0285 – 1 Kč	záleží na pásmu účiníku a přirážce
Čína	0.9	-	záleží na regionu/městě
Estonsko	0.95	-	-
Francie	0.92	0.2 €	-
Kanada	0.9 – 1	-	záleží na členských státech
Litva	0.95	-	-
Lotyšsko	0.95	-	-
Německo	0.9 – 0.95	-	záleží na el. rozvodné síti
Portugalsko	0.92	0.016 – 0.18 €	záleží na měsíci
Rakousko	0.95	0.018 – 0.025 €	záleží na měsíci
Rumunsko	0.95	0.6 – 2 Leu	záleží na velikosti $\tan\phi$
Rusko	-	-	žádné pokuty ani povinné rozmezí účiníku
Španělsko	0.95	0.0001 – 0.038 €	záleží na míře odchylky
Thajsko	0.85 – 1	15 ฿	-
USA	0.9 – 0.95	3 – 7 $	záleží na členském státě

[1] francouzská společnost, působící v oblasti elektrotechnického průmyslu zaměřující se na efektivní hospodaření s energiemi

3 Návrh a realizace řešení

Čtvrtá kapitola pojednává o postupu při návrhu a vytváření aplikace zpracovávající konkrétní výstupy ze zařízení, které sleduje tok elektrické energie v síti. Dále poskytuje náhled do procesu výběru platformy a typu aplikace. Mimo návrhu výsledné aplikace popisuji i neúspěšné řešení.

3.1 Volba operačního systému

První krok, nad kterým jsem se zamyslel při tvorbě programu, byla správná volba optimálního operačního systému. Již bylo zmíněno, že .NET Core běží na všech rozšířených prostředích. Záleží však i na tom, jak uzpůsobený je samotný operační systém k tomu, aby se v něm .NET Core aplikace daly efektivně a snadno vyvíjet. V zadání práce také není specifikovaný typ systému, pro který má být aplikace uzpůsobena. Jako evidentní a jasná volba se mi však i na první pohled jevil Windows. Je totiž obecně známo, že Windows je obecně nejrozšířenější operační systém a aplikace by tak mohla zaujmout co největší množství uživatelů. Ve prospěch systému od Microsoftu hraje i fakt, že mají s .NET Core společného tvůrce, což zaručuje stoprocentní kompatibilitu a dostupnost všech potenciálně potřebných nástrojů.

Na Windows jakožto správnou volbu tedy ukazují zajímavá fakta. Existuje ale i důležitý faktor, odrazující mne od využití tohoto systému. Tím faktorem je dostupné vývojové prostředí (dále už jen IDE) pro vývoj aplikací v .NET Core. Microsoft poskytuje a doporučuje Microsoft Visual Studio. Problém však nastává v okamžiku, kdy se v tomto programu někdo pokouší pracovat. IDE má tendenci zamrzat, leckdy pracovat velmi pomalu, a také velice výrazně zatěžovat systém, což dokáže práci znepříjemňovat a zdržovat. V tomto směru určitě vede verze Visual Studia optimalizovaná pro operační systém MacOS. Apple obecně ohledně aplikací na svůj systém vyznává politiku naprosté optimalizace, čímž zaručuje, že programy poběží naprosto hladce. Díky tomu Visual Studio na tomto systému funguje svižně a bez jakýchkoli problémů.

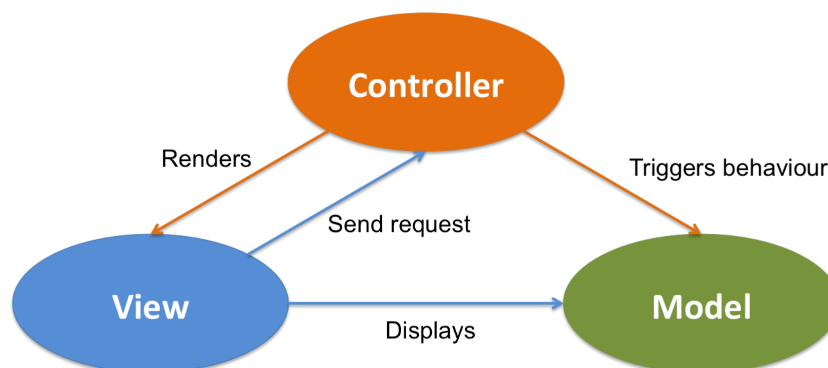
Verze pro Apple však nepodporuje některé NuGet moduly třetích stran potřebné pro vývoj reportové aplikace. Jedním z nich je i již zmiňovaný DevExpress, což mne vedlo k rozhodnutí vytvořit program pro systém Windows. Po přečtení kapitoly 1.2.2 by se dalo namítat, že přece lze použít knihovnu DevExtreme pro vývoj na MacOS. Tomu však zabraňuje zvolený typ aplikace.

3.2 Výběr typu aplikace

ASP.NET Core podporuje vývoj široké škály aplikací. Podporuje vývoj lokálních aplikací i webových. Správný výběr tedy silně závisí na tom, k čemu má být aplikace určena. Jelikož musí umět zobrazovat zpracovaná data v grafech či tabulkách, nehodí se vytvářet konzolovou aplikaci. Lepší volbu by mohla představovat aplikace Windows Forms, která vývojářům nabízí možnost si sestavit celé grafické rozhraní včetně oněch zmíněných grafů a tabulek. V dnešní době moderních javascriptových frontendových aplikací však výsledné vzezření působí zastarale a komplikovaně pro širší využití. Vyvstává také otázka, zda bude aplikace fungovat lokálně na jednom zařízení, nebo zda půjde o aplikaci hoštěnou na webu, ke které bude mít uživatel přístup přes internetový prohlížeč. Ani tento parametr není specifikovaný v zadání práce, proto výběr typu programu závisí čistě na návrhu fungování aplikace. Jako nejoptimálnější řešení při zhodnocení všech výhod či nevýhod zmíněných druhů mi připadal typ webové aplikace Model-View-Controller (dále už jen MVC) realizovaný na lokálním Kestrel serveru (viz kapitolu 1.1.3).

3.3 MVC webová aplikace

ASP.NET Core MVC je bohatou architekturu pro vytváření webových aplikací a rozhraní API pomocí Model-View-Controller návrhu [16]. Vyznačuje se oddělenou sekcí pro logiku programu, která se celá realizuje v sekci Model, a grafickým rozhraním (view - pohledem) typicky zobrazovaném ve formě HTML souboru, mezi kterými řídí komunikaci kontrolér (controller). Jelikož jsem před tím neměl moc zkušeností s .NET Core, nejdůležitějším důvodem volby MVC aplikace byla snadná implementace zobrazení grafů a tabulek žádoucích při vytváření reportů. Pohled i kontrolér závisejí na modelu, model ale nezávisí na žádném z ostatních komponentů, tudíž může být samostatně implementován, testován a odladěn.



Obrázek 3.1: Vzájemné odkazy jednotlivých komponent v modelu MVC
Zdroj: https://www.javacodegeeks.com/2017/09/mvc-delivery-mechanism-domain-model.html

3.3.1 Model

Model představuje backendovou implementační část aplikace. Obsahuje několik tříd zajišťujících chod programu. Ty jsem si rozdělil do dvou základních kategorií. Jako první druh jsem si navrhl logickou třídu obsahující metody všech nutných operací. Ke druhému druhu jsem přiřadil ty třídy, které seskupují data do objektů potřebných pohledem ke správnému zobrazení výsledných grafů.

Hlavním modelem jsem zvolil třídu Logic. V ní bylo třeba nejprve řešit načítání souborů obsahujících výstupní data ze sledovacího zařízení kompenzace jalového výkonu. Informace jsem načetl do pomocné tabulky sloužící k rozčlenění informací. První sloupec obsahuje datum, ostatní pak názvy jednotlivých sledujících veličin. Podle těchto ukazatelů sloužících jako souřadnice lze pak v tabulce snadno dohledat jakoukoli uloženou informaci. Každý řádek pomocné tabulky obsahuje jeden moment sledování, tedy časový údaj a hodnotu sledované veličiny. Kromě souboru s daty jsem si do programu načetl ještě konfigurační textový soubor. V něm jsou uložené informace o nastavení stupňů jednotlivých stykačů.

Jako první jsem se rozhodl si vytvořit atributy pro sledování doby záznamu, uložil jsem si časy startu a konce záznamu a vytvořil z nich hodnotu typu TimeSpan vyjadřující dobu záznamu. Tu jsem si uložil, protože se v konečném reportu zobrazí informace o době sledování, a zároveň je tato hodnota velmi nápomocná při výpočtu průměrných hodnot všelijakých veličin. Z hodnot typu TimeSpan se totiž dají vyjádřit celkové časové údaje, jako je celková doba sledování v sekundách, minutách, hodinách atd. Zároveň umí časové jednotky mezi sebou i převádět, nemusel jsem to tedy řešit já jakožto vývojář ručně a eliminoval jsem možnost omylu.

Za další atributy logické třídy jsem zvolil tři listy, ve kterých se po vypočítání budou uchovávat jednotlivé informace o sledovaných veličinách. Právě pro listy jsem se rozhodl oproti obyčejným polím, protože se množství uchovávaných dat může měnit v závislosti na délce sledovaného období. Do prvního listu ukládám informace o odchylce jalového výkonu. Pro každý zaznamenaný moment program vypočítává očekávaný jalový výkon úpravou vzorce 2.5 tím způsobem, že hledaná veličina je nyní jalový výkon Q. Za původně hledaný účiník potřebný v používaném vzorci dosazuji hodnotu požadovaného účiníku nastaveného ve sledovacím zařízení. Skutečnou hodnotu jalové energie získávám z pomocné tabulky. Výslednou požadovanou odchylku následně dostanu dosazením obou hodnot do vzorce 2.6.

```
List<Deviation> tmp = new List<Deviation>();
for (int i = 0; i < Table.Rows.Count - 1; i++){
    tmp.Add(new Deviation{
        Date = DateTime.Parse(Table.Rows[i]["čas záznamu[s]"].ToString()),
        ExpectedValue = Math.Tan(Math.Acos(expectedCosPhi)) / double.Parse(Table.Rows[i]["prm.3P[kW]"].ToString()),
        DeviationValue = Math.Tan(Math.Acos(expectedCosPhi)) / double.Parse(Table.Rows[i]["prm.3P[kW]"].ToString())
        - double.Parse(Table.Rows[i]["prm.3Q[kvar]"].ToString())
    });
}
```

Obrázek 3.2: Výpočet a ukládání regulační odchylky

Obdobným způsobem si do druhého listu ukládám informace o odchylce od účiníku. Hodnotu skutečného naměřeného jalového výkonu v síti opět získávám z pomocné tabulky. Očekávaný jalový výkon poté získává dosazením do vzorce 2.5. V tomto případě se za účiník dosadí očekávaná hodnota získaná ze sledovacího zařízení.

```
List<Deviation> tmp = new List<Deviation>();
foreach (DataRow row in Table.Rows){
    string[] data = row["3Cosφ[]"].ToString().Split(" ");
    double expectedCosPhi = Math.Cos(Math.Atan(formatPlatform(row["prm.3Q[kvar]"].ToString()) /
                formatPlatform(row["prm.3P[kW]"].ToString())));
    if (data.Length == 1){
        continue;
    }else{
        if (data[0] == "L"){
            tmp.Add( new Deviation{
                Date = DateTime.Parse(row["čas záznamu[s]"].ToString()),
                DeviationValue = expectedCosPhi - formatPlatform(data[1])
            });
        }else{
            tmp.Add(new Deviation{
                Date = DateTime.Parse(row["čas záznamu[s]"].ToString()),
                ExpectedValue = expectedCosPhi - formatPlatform(data[1])
            });
        }
    }
}
```

Obrázek 3.3: Výpočet a ukládání odchylky od účiníku

Zbývající list slouží k ukládaní informací o využití stykačů. Informace o využití stykačů jsem si opět společně se všemi ostatními načetl do pomocné tabulky při startu programu. I tak si program musí některé doplňující informace vypočítat. V tabulce se totiž nacházejí data pouze v absolutních hodnotách. K dispozici má tedy pouze celkovou dobu v provozu od zapojení, celkový počet sepnutí za dobu od zapojení, a stupně v kilovarech načtené separátně z konfiguračního souboru. Proto jsem naimplementoval metody pro vypočítávání průměrných počtů sepnutí a dob v provozu za různá období od minut po dny a počty sepnutí za jednotlivé celé dny sledovacího období. Tyto informace mají totiž důležitou vypovídající hodnotu a rád bych je tedy zobrazil v pohledech v grafech.

Poslední funkcí, kterou musí třída Logic zajišťovat, je zjišťování a uchovávání dat popisujících zobrazované grafy. Protože jsem chtěl v pohledech vypisovat legendy grafů v tabulkách, nejjednodušší formou uchování dat v logice se mi jevila také tabulka. Při přenosu dat do pohledu se totiž informace nemusí nijak převádět či formátovat, tabulky v pohledech a logice jsou plně kompatibilní. Navrhl jsem si tedy pro každý pohled (jsou celkem tři) globální atribut typu DataTable, kam shrnuji zajímavé či důležité hodnoty pro popis grafu, jako jsou minima, maxima, průměry atp.

Na obrázcích 3.2 a 3.3 lze vidět, že do listu neukládám samotné číselné údaje o veličinách, ale objekt jménem Deviation. Jde o jednu ze dvou tříd, kterou jsem vytvořil pro seskupení vypočtených zobrazovaných dat. Grafy v pohledu totiž umí získávat data pouze jako celek, nikoli jednotlivě. Proto jsem naimplementoval třídu

Deviation pro informace o odchylkách a třídu SwitchData pro informace o využití všech stykačů. Jednotlivé atributy těchto tříd představují argumenty pro osy v grafu. Zároveň i zpřehledňují kód třídy Logic, protože snižují počet nutných globálníhc atributů. Každá instance pomocných objektů tedy znázorňuje jeden moment záznamu sledování kompenzace. Objekt Deviation sdružuje u každého zaznamenaného okamžiku časový údaj, hodnotu očekávaného jalového výkonu a odchylku od této hodnoty. Objekt Deviation jsem chtěl použít univerzálně pro ukládání hodnot o regulační i účiníkové odchylce. V případě účiníkové odchylky však neukládám očekávanou hodnotu a odchylku, ale dvě odchylky. Jedna je induktivního charakteru (označení L na zobrazení 3.3), druhá kapacitního (v podmínce else na zobrazení 3.3). Abych ušetřil místo v paměti, rozhodl jsem se nealokovat samostatný atribut, ale ukládat výslednou odchylku do atributu pro očekávanou hodnotu, který by jinak zůstal nevyužit. Instance SwitchData naproti tomu nemusí pracovat s časem, protože ve výsledném pohledu zobrazují jiný druh grafu. Místo jednotlivých momentů zaznamenává o každém jednom stykači jeho označení, průměrné počty sepnutí za různá časová období a celkový počet sepnutí za dobu záznamu i dobu zapojení. Dále uchovává ještě údaje o době zapojení, jejich průměrné hodnoty a stupně v kilovarech.

Na obrázku 3.3 si lze všimnout ještě jednoho důležitého detailu, a to že jsem před zjištěním odchylky zavolal na naměřenou veličinu účiníku metodu formatPlatform(). Přistoupil jsem k tomuto kroku, jelikož jsem potřeboval převést veličinu z typu string na double a desetinná místa byla v řetězci oddělena čárkou. Za normálních okolností bych s tím neměl žádné problémy a snadno bych pomocí příkazu double.Parse(<řetězec>) výraz přeformátoval z textového řetězce na číslo. Já jsem ale při implementaci logiky programu nepracoval na operačním systému Windows, ale MacOS. Problém nastal z toho důvodu, že Visual Studio na MacOS nezná čárku jakožto oddělovač desetinných míst, ale čte ji jako oddělovač jednotek tisíců. Vypočítávala se pak a následně se i zobrazovala nesmyslná data v řádech tisíců. Přidal jsem proto tuto metodu, ve které nejprve nahradím čárku tečkou a až pak jí převádím na typ double. Tato drobná metoda programu umožňuje bez problému běžet na všech rozšířených operačních systémech.

```
private double formatPlatform(String s){
    if (RuntimeInformation.IsOSPlatform(OSPlatform.OSX)){
        return double.Parse(s.Replace(",", "."));
    }
    return double.Parse(s);
}
```

Obrázek 3.4: Formátování čísla podle typu operačního systému

3.3.2 View (Pohled)

Pohled zprostředkovává komunikaci s uživatelem. Jedná se o jednu nebo více tříd vykreslujících zobrazované elementy a informace. Vkládat obsah do těchto tříd se dá dvěma způsoby. Jako první možnost může programátor použít funkci vkládání elementů pomocí značkovacího jazyka HTML užívaného při vývoji webových stránek. Alternativu má vývojář v podobě modulu speciálně navrženého pro potřeby vývoje MVC aplikací jménem Razor. Razor kombinuje HTML značky a syntaxi jazyka C#. Sekci, kde se používá jazyk C#, odděluje znak "@". Názorná ukázka se nachází v HTML značce <h1> v zobrazení 3.6. Výsledná třída podporující Razor modul se vyznačuje příponou CSHTML.

Architektura MVC dělí pohledy do dvou kategorií. Do první kategorie spadají pohledy obsahující elementy společné pro všechny CSHTML třídy. I já jsem si vytvořil jednu takovou jménem Layout. V ní jsem vytvořil základní strukturu typickou pro HTML soubory tvořenou značkami <html>, <head> a <body>. Jelikož má mít celá aplikace jednotné stylování, umístil jsem sem odkazy na CSS soubory. Vykreslení a obsluhu grafů a tabulek umožňují javascriptové knihovny. Proto jsem do třídy Layout vložil odkazy i na ně. Poslední důležitou součást třídy tvoří navigační lišta s tlačítky na přesměrování na každou jednotlivou třídu zobrazení. Aby se dokázal v šabloně tvořené třídou Layout vykreslit pohled některé z dalších tříd, musí se v těle třídy nacházet volání funkce RenderBody. Ta zajistí zobrazení určitého obsahu v bloku svého umístění.

```
<div class="container body-content">
    @RenderBody()
</div>
```

Obrázek 3.5: Volání vykreslení obsahu do požadované sekce

Do druhé kategorie patří jednotlivé třídy vykreslující každá vlastní obsah. Pro potřeby aplikace jsem vytvořil čtyři pohledy. První pohled nese název Index(), a slouží k načítání těch souborů, o kterých chce uživatel vytvořit report. Pojmenoval jsem ho Index, protože jde o prvotní zobrazení vykreslené při spuštění aplikace. Druhý pohled slouží k zobrazení informací o odchylce od očekávaného jalového výkonu. Pojmenoval jsem ho RegulationView. Ve třetím pohledu jménem SwitchAvgWiew jsem vytvořil zobrazení reportu o stavu stykačů v síti. O účiníku jsem vložil informace do posledního pohledu, kterému jsem dal název CosPhiView. Pohledy přejímají HTML šablonu od třídy Layout a vykreslují se do sekce div znázorněné na obrázku 3.5. Proto jsem v těchto třídách mohl rovnou vkládat chtěné elementy. Samozřejmě je možné implementovat sekce či divy nižší úrovně, dokonce to i byla nutnost, už jsem ale nemusel znovu definovat značky <head>, <body> apod., i když šlo o samostatné CSHTML soubory. Každý z pohledu však musí obsahovat jednu důležitou součást, aby se mohl vykreslit v šabloně Layout. Všem se musí definovat cesta k onomu souboru. Jak takový odkaz vypadá, znázorňuji na obrázku 3.6.

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1 id="jmenoSouboru">@Model.Table.TableName</h1>
```

Obrázek 3.6: Odkaz na umístění šablony pohledu nad příkladem převodu mezi jazyky C# a JavaScript

3.3.3 Controller

Controller slouží jako řídící jednotka celého programu. Reaguje na uživatelské požadavky, a následně je předává od uživatele z pohledu do modelu a opačně, také vykresluje zobrazení pohledu. MVC aplikace mohou mít controllerů klidně více v závislosti na tom, kolik modelů musí program řídit a kolik funkcí musí obsluhovat. Podle typu obsluhy mohou mít metody v controlleru různé návratové datové typy, některé vytvořené speciálně pro architekturu MVC. Jako první mohou metody vracet všechny známé primitivní datové typy či uživatelem definované objekty. Slouží především k ochraně formátu zobrazovaných dat. Dále se běžně jako návratový typ používá ActionResult. ActionResult zajišťuje komunikaci s pohledem tím způsobem, že se stará například o operace vytváření pohledu, vytváření částečného pohledu, přesměrovávání mezi vícero ActionResult metodami, navrácení uživatelsky definovaných objektů aj. V nutných případech ho lze i kombinovat se zmiňovanými primitivními datovými typy. Zapisuje se pak jako ActionResult<T>, kdy T reprezentuje datový typ či objekt. Může nastat situace, kdy lze jednou akcí vrátit více podtypů ActionResultu. V těchto případech se využívá nadřazená třída IActionResult.

Program potřebuje pro obsluhu všech součástí pouze jeden controller. Do něj jsem umístil několik metod obsluhující všechny potřebné součásti. Jména vykreslovacích metod v controlleru se pak musejí vždy shodovat se jmény příslušných ovládaných pohledů. Za hlavní řídící metodu jsem zvolil ActionResult metodu Index(). Spustí se při prvotním spuštění aplikace, a vykreslí pohledovou třídu Index. Tu samou stránku obsluhuje ještě funkce Upload(). Tu program zavolá po vybrání souboru pro sestavení reportu.

Metoda vytvoří instanci modelové třídy Logic. Protože funkce nevykresluje žádný nový obsah v pohledu, ale pouze vytváří instanci modelu, nastavil jsem ji jako návratovou hodnotu objekt EmptyResult. Na zobrazení 3.7 si lze všimnout značky [httpPost]. Upload takhle musí být označená, jelikož jí pohled posílá přeš HTTP data, v tomto případě soubor k vytvoření modelu. Controller při každém zavolání obsluhující metody vytvoří novou instanci všech atributů, aby tedy nedocházelo k null referencím při volání atributu Logic, definoval jsem logiku jako statický globální atribut. Vlastnost static zajišťuje, že se vytvoří jen jedna instance objektu.

```
[HttpPost]
public ActionResult Upload(){
    try{
        var file = Request.Form.Files["fileUploader"];
        Stream stream = file.OpenReadStream();
        Logic = new Logic(stream);
    }catch{
        Response.StatusCode = 400;
    }
    return new EmptyResult();
}
```

Obrázek 3.7: Vytvoření instance modelu v controlleru

Pro vykreslování všech pohledových tříd jsem napsal víceméně stejné metody. Liší se pouze ve jménu, jelikož jak už jsem avizoval, každé jméno se musí shodovat s názvem obsluhovaného pohledu. Všechny pohledy kromě Indexu pak společně s vykreslením zobrazení předávají instanci vytvořeného modelu. Díky tomu lze v pohledech pracovat i s modelovými daty. Příklad metody, která v tomto případě vykresluje pohled s regulační odchylkou, lze nahlédnout v horní části obrázku 3.8.

V tomto momentě by tedy aplikace byla schopná zobrazit prázdné stránky aplikace. V pohledech se ale mají nacházet grafy a tabulky. Samotné vložení elementů se sice řeší až v samotných pohledech, poskytování těch správných dat ve správném formátů ale také řídí controller. Dohromady jsem byl nucen naimplementovat šest metod pro obsluhu grafů a tabulek. Jméno metod tentokrát nehraje roli. Jako návratový typ metod jsem zvolil podtřídu třídy ActionResult jménem ContentResult, protože umožňuje vracet uživatelsky definovaná data. V kapitole 3.3.1 jsem se zmínil o objektových třídách potřebných pro seskupení relevantních dat k zobrazení. Zde v controlleru jsem musel ona sdružená data převést do formátu JSON, aby je mohl daný element v pohledu zpracovat a vykreslit (viz obr. 3.8). Každou z těchto metod si z pohledu volá každý jednotlivý zobrazovací element. Ne každý graf nebo tabulka musí mít v controlleru svou vlastní metodu, klidně může mít více prvků stejný zdroj dat, pouze pak zobrazuje jiná obsažená data.

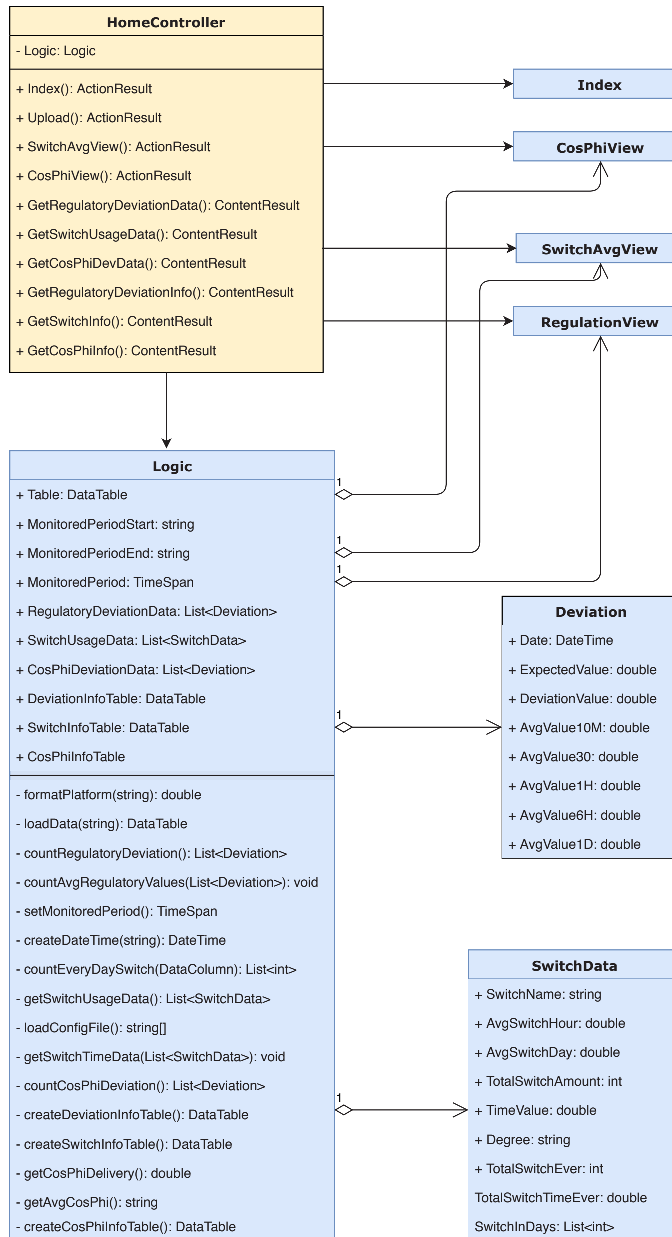
```
public ActionResult RegulationView(){
    return View(Logic);
}

public ContentResult GetRegulatoryDeviationData(){
    return Content(JsonConvert.SerializeObject(Logic.RegulatoryDeviationData),
        "application/json");
}
```

Obrázek 3.8: Implementace controlleru vykreslující pohled a ovládající graf

3.4 Class diagram

Pro lepší představu o struktuře ukázkové aplikace přikládám i class diagram.



Obrázek 3.9: Class diagram programu

3.5 Vložení elementů do pohledů

Důležitými součástmi reportu jsou bezesporu grafy a tabulky. Ty poskytují uživateli přehledně informace ve srozumitelné formě. Dále je potřeba zobrazované informace nějakým způsobem načítat a obsluhovat. V této kapitole popíši postupy při implementaci těchto elementů do pohledů aplikace. Vyjádřím se ke dvěma způsobům tvorby, které jsem při implementaci vyzkoušel.

3.5.1 Chart.js

Prvním příhodným postupem, který jsem zvolil, bylo využití značkovacího jazyka HTML v kombinaci s jazykem JavaScript, přesněji s knihovnou Chart.js určenou k vykreslování grafů. Chtěl jsem zobrazovat grafy v elementu canvas, jelikož ho podporuje většina dnešních nejrozšířenějších internetových prohlížečů. Nejprve bylo ale třeba vyřešit otázku načítání zpracovávaného souboru do programu. K tomu účelu jsem využil HTML tag <input> s vlastností `type="file"`. Ten po kliknutí otevře dialogové okno pro vybrání souboru. Dále jsem vložil potvrzující tlačítko, které odešle zvolený soubor do controlleru resp. do metody Upload() (viz obr. 3.7). Měl jsem tedy způsob, jak načíst zpracovávaný soubor a vytvořit model. Vyvstala však otázka, jak přenést informace z modelu do pohledu v takovém formátu, jaký Chart.js zvládne zpracovat. Dokáže totiž vkládat data do grafu pouze v jazyce JavaScript. Naštěstí modul Razor umožňuje převod atributů mezi jazyky C# a JavaScript. Jak takový přenos funguje lze nahlédnout na obrázku 3.10.

```
let expectedReactivePowerData = [];
@foreach(var item in Model.expectedReactivePowerData){
    @:expectedReactivePowerData.push("@item");
}
```

Obrázek 3.10: Příklad přenosu dat do jazyka JavaScript

Vkládat křivky do grafu v modulu Chart.js lze provést dvěma způsoby. První možností je definování pole hodnot pro každou křivku grafu představujících osu Y a pole argumentů pro osu X. Graf podle indexů v oněch polích spojí body a sestaví graf. K realizaci druhého způsobu stačí jen jedno pole obsahující obě souřadnice os X a Y. Abych nemusel v modelu definovat nové struktury pro jednotlivé body se souřadnicemi, rozhodl jsem se realizovat první možnost. Abych zachoval přehlednost kódu v pohledových třídách, vytvořil jsem si pomocnou javascriptovou třídu, kam jsem implementoval veškeré metody potřebné k realizaci a obsluze grafů. Samotná implementace každého Chart.js grafu se skládala z několika částí. Ze všeho nejdřív bylo třeba sdělit grafu při jeho inicializaci, kam se má vykreslit. To znamenalo načtení příslušného canvasu příkazem `document.getElementById("<jméno canvasu>")`. Následně jsem přiřadil jednotlivá pole s daty k osám či zdroji dat a zvolil typ grafu. Posledním důležitým elementem Chart.js grafu jsou jeho vlastnosti. V nich se nachá-

zejí definice popisků os, název grafu, nastavení kroku mezi body v grafu, inicializace externích pluginů apod.

V této fázi implementace již program zvládne vykreslit grafy jednotlivých sledovaných veličin. Samotný JavaScript ani knihovna Chart.js však neobsahují žádné nástroje například pro přizpůsobení grafu. Když jsem tedy chtěl přidat do grafu popisky křivek, musel jsem stáhnout modul k tomu určený. Může také nastat situace, kdy bude zobrazovací doba dlouhá, graf bude hodně nahuštěný a uživatel si bude chtít některý moment přiblížit. Ani tuto funkci ale JavaScript nepodporuje, musel jsem tedy do projektu přidal další plugin. Výčet všech modulů nutných pro uvedení grafů do stavu, kdy splňují základní požadavky a kdy vyhovují potřebám běžného uživatele, obsahuje následující kapitola 3.6.

```
new Chart(document.getElementById("switchCountCanvasAvg"), {
    type: 'bar',
    data: {
        labels: labels,
        datasets: [{
            label: "Průměrný počet přepnutí za hodinu", backgroundColor: "#53dc46",
            data: avgHourData
        }]
    },
    options: {
        title: {
            display: true, text: 'Graf využití stykačů'
        }
    }
});
```

Obrázek 3.11: Příklad implementace Chart.js grafu

Jak už jsem zmiňoval při popisu modelu v kapitole 3.3.1, popisky a vysvětlivky grafů ukládám do tabulek. K zobrazení těchto informací v pohledu jsem využil HTML značku table. Řádky i sloupce jsou plně kompatibilní s modulem Razor. Nebylo tedy nutné převádět data do JavaScriptu, jak jsem to prováděl u grafů, ale rovnou jsem přiřadil příslušné hodnoty HTML elementu podobným způsobem zobrazenému na obrázku ??.

3.5.2 DevExtreme

Jako druhou možnost jsem zvolil vytvoření obsahu pohledů pomocí modulu DevExtreme. DevExtreme podporuje nativně bez jakékoli doinstalace dalších nástrojů tvorbu elementů od tabulek a grafů až po dialogová okna, mapy, davigační stromy apod. Právě kvůli této knihovně jsem byl nucen vytvořit v modelu pomocné seskupující objekty a v controlleru zmiňované ContentResult metody pro převod dat do formátu JSON. V pohledu při implementaci DevExtreme elementů se totiž data již nedají nijak upravovat. Opět jsem jako první řešil, jak načíst zpracovávaný soubor do programu. K tomu účelu nabízí DevExtreme nástroj FileUploader. Definování vlastností FileUploaderu i ostatních elementů je velmi snadné. Pro přidání jakéhokoli atributu stačí přidat řádek s názvem funkce ve formě `.název(argument)`. Já

jsem, například, potřeboval nastavit vlastnost okamžitého načtení bez potvrzovacího tlačítka, přidal jsem tedy vlastnost .UploadMode(FileUploadMode.Instantly). Stejným způsobem jsem omezil výběr na pouze jeden soubor a definoval, kam do controlleru se má soubor odeslat ke zpracování. Samozřejmě jde opět o zmiňovanou metodu Upload() (viz obr. 3.7). Celou implementaci lze nahlédnout na obrázku 3.12.

```
@(Html.DevExtreme().FileUploader()
        .ID("file-uploader")
        .Name("fileUploader")
        .Multiple(false)
        .Accept("*")
        .UploadMode(FileUploadMode.Instantly)
        .UploadUrl(Url.Action("Upload", "Home"))
        .OnValueChanged("fileUploader_valueChanged")
    )
```

Obrázek 3.12: Příklad implementace DevExtreme FileUploaderu

Po vyřešení načítání souborů jsem přistoupil k implementaci grafů. DevExtreme grafy mají několik specifických vlasností. Jak už jsem zmínil v úvodu kapitoly, data nešla nijak upravovat, mohl jsem pouze nastavit, jaká data patří jako argument ose X a která ose Y. Zdroj dat může obsahovat i více veličin kromě těch zobrazovaných, lze zvolit i pouze některé k vykreslení. Zajímavou funkci nabízí při nastavování typu jednotlivých křivek. Lze definovat pro všechny křivky společný nebo každé veličině svůj vlastní. Obrázek 3.13 znázorňuje implementaci jednoduchého grafu pouze s nastavením zobrazovaných veličin. Například když jsem potřeboval rozlišovat jednotlivé grafy pomocí ID, stačilo připsat řádek .ID("<jméno grafu>"). Obdobným způsobem jsem nastavil grafům barvy, popisky os, možnost přiblížení momentů, formátování hodnot na ose X k ukazování času a zobrazení dialogu o načítání grafu. Na rozdíl od knihovny Chart.js jsem nemusel zajišťovat externí pluginy pro každou chtěnou funkci, DevExtreme všechny nástroje obsahuje již v základní verzi při pořízení. Obecně tedy poskytuje ucelenější a jednodušší sadu nástrojů pro tvorbu grafů v pohledech.

```
@(Html.DevExtreme().Chart()
    .DataSource(d => d.StaticJson().Url(Url.Action("GetRegulatoryDeviationData")))
    .CommonSeriesSettings(s => s.ArgumentField("Date"))
    .Series(s =>{
        s.Add().Name("Ocekavany jalovy vykon").ValueField("ExpectedValue").Type(SeriesType.Line);
        s.Add().Name("Regulacni odchylka od jaloveho vykonu").ValueField("DeviationValue").Type(SeriesType.Bar);
    })
)
```

Obrázek 3.13: Příklad implementace DevExtreme grafu

Velmi zajímavý nástroj jsem použil při implementaci přibližování grafu. Často může nastat situace, kdy chci v grafu zobrazit pouze vybranou oblast. Pokud je na ose X čas, tak bych například rád zobrazil jen určitý interval. K tomu účelu DevExtreme podporuje funkci zoomu pomocí myši i procházení užitím scroll baru, také jsem

přemýšlel i nad naprogramováním nějaké obdoby kalendáře. Následné procházení grafu by pak ale vyžadovalo implementaci několika tlačítek či jiných elementů, jejichž programování a vzájemná synchronizace by byla zbytečně náročná. Jako daleko efektivnější a jednodušší řešení se mi jevilo použití nástroje jménem Range Selector. Umožnil mi pod jednotlivými grafy vytvořit časovou osu, která je s grafy propojená a na které si mohu nastavit hranice zobrazovaného intervalu. Pro pohyb v grafu mi pak stačí pouze interval na časové ose uchopit, a posouvat ho chtěným směrem. Se změnou intervalu se pak změní i zobrazovaný rozsah v připojeném grafu. Výhoda selektoru spočívá kromě jednoduchosti implementace i v tom, že nemusím specifikovávat žádné pevné intervaly, uživatel si podle potřeby sám nadefinuje vlastní.

```
@(Html.DevExtreme().RangeSelector()
    .Chart(c => c
        .Series(s =>{
            s.Add().ArgumentField("Date").ValueField("ExpectedValue");
            s.Add().ArgumentField("Date").ValueField("DeviationValue");
        })
    )
    .Behavior(b => b.CallValueChanged(ValueChangedCallMode.OnMoving))
    .OnValueChanged(@<text>
        function(e) {
            var zoomedChart = $("#deviationChart").dxChart("instance");
            zoomedChart.getArgumentAxis().visualRange(e.value);
        }
    </text>)
    .DataSource(d => d.StaticJson().Url(Url.Action("GetRegulatoryDeviationData")))
)
```

Obrázek 3.14: Příklad implementace Range Selectoru

Aby mohla v Range Selectoru časová osa určitého grafu vůbec vzniknout, definoval jsem mu stejný zdroj dat zavoláním též metody controlleru v atributu `.DataSource()`. Synchronizaci s grafem zajišťuje vlastnost `.OnValueCanged()`. Do argumentu jsem napsal jQuery funkci, ve které se podle označení načítá instance požadovaného grafu. Následně, pokud zaznamená změnu délky intervalu v Selectoru, změní automaticky zobrazení i v grafu. Na celou implementaci obou zmíněných vlastností lze nahlédnout na obrázku 3.14.

Ani v případě tabulek není možné data jakkoli měnit. Stejně jako při implementaci grafů jsem pouze stanovil, které veličiny chci zobrazit v jednotlivých sloupcích. Dále už mi zbývalo jen tabulku vizuálně a funkčně přizpůsobit. Nastavil jsem možnost selekce jednotlivých řádků, přidal pole pro vyhledávání v tabulce a možnost seskupení hodnot do sekcí podle veličin.

```
@(Html.DevExtreme().DataGrid()
    .DataSource(d => d
        .StaticJson()
        .Url(Url.Action("GetRegulatorDeviationInfo"))
    )
)
```

Obrázek 3.15: Příklad implementace DevExtreme tabulky

Aplikaci jsem navrhl tak, aby v případě, že jakékoli sledované veličiny nejsou v oboru přípustných hodnot, na to aplikace nějakým způsobem upozorňovala. Tuto funkci jsem naprogramoval do tabulek s popisky grafů. Řekl jsem sice, že zobrazovaná data v tabulkách nelze nijak měnit, DevExtreme však podporuje vizuální úpravu zobrazovaných veličin pomocí JavaScriptu. Napsal jsem funkci, která sleduje rozsah hodnot a pokud se jakákoli veličina od rozmezí lehce odchýlí, zvýrazní ji oranžově. Pokud najde hodnotu překračující dalece hranici rozsahu, označí ji červeně za kritickou chybu. Tuto funkcionalitu jsem do tabulky přidal pomocí atributu .OnCellPrepared("<jméno obsluhující funkce>").

3.6 Externí moduly

V kapitole popisující implementaci grafů pomocí jazyka JavaScript (viz kapitolu 3.5.1) zdůrazňuji, že lze uživatelsky přívětivěji přizpůsobit Chart.js grafy pomocí externích volně dostupných modulů. Mohl jsem si buď volně stáhnout zdrojový soubor modulu z GitHubu a následně ho přidat pouze do právě vyvíjeného projektu nebo si ho nainstalovat pomocí správce javascriptových balíčků známým jako NPM. V takovém případě by pak byly moduly dostupné v případě potřeby pro všechny další možné projekty. Jelikož jsem ale během vývoje programu měnil počítače, rozhodl jsem se pro první možnost, abych se nemusel zabývat opětovnou instalací všech balíčků v případě, že bych je na druhém počítači nainstalované neměl.

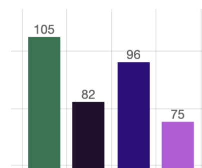
V následujících podkapitolách představím jednotlivé moduly pro přizpůsobení grafů. Všechny jsem použil při implementaci pohledů pomocí knihovny Chart.js, DevExtreme totiž žádné takové moduly k fungování nepotřebuje, všechny nástroje obsahuje již v základní verzi.

3.6.1 Chartjs-plugin-labels.js

Plugin Chartjs-plugin-labels umožňuje přidávat do grafu popisky křivek. Podporuje zobrazování názvu křivek, hodnotu aktuálního bodu v numerické i procentuální hodnotě či dokonce vkládání obrázků místo popisů. Kromě formy zobrazení nabízí i možnost poznámky stylovat do různých barev a upravovat natočení textu. Abych mohl modul použít, vložil jsem do třídy Layout odkaz na zdrojový soubor knihovny. Samotný kód jsem napsal do sekce options při definici grafu (viz obr. 3.11). Samotný kód i výsledek předvádím na obrázcích níže.

```
plugins: {
    labels:{
        render: 'value'
    }
}
```

Obrázek 3.16: Implementace zobrazení popisů křivek



Obrázek 3.17: Výsledné zobrazení popisů křivek

3.6.2 Chartjs-plugin-zoom.js

Díky modulu Chartjs-plugin-zoom.js jsem získal možnost si přibližovat jednotlivé body či křivky v grafu. Mohl jsem nastavit, zda chci přibližovat pouze osu X, pouze osu Y nebo obě. Jelikož by se mohlo stát, že by přiblížení bodů křivek pouze na jedné ose nemuselo být moc efektivní, naimplementoval jsem tuto funkci u obou os. Kromě samotného umožnění zoomování lze zadat i rychlost a maximální rozsah. Implementace je obdobná předchozímu pluginu. Opět jsem přidal odkaz na zdrojový soubor do šablony ve třídě Layout a naprogramoval funkcionalitu v oddílu options v příslušném grafu.

3.6.3 Hammer.js

Fungování úzce souvisí s předchozím bodem. Plugin Hammer.js totiž přidává možnost manipulace s grafem, což se využije hlavně při přiblížení grafu, kdy se některé body křivek dostávají mimo zorné pole. Opět jde nastavit rychlost a rozsah pohybu. Tentokrát jsem nemusel implementovat žádný kód modulu do konfigurace grafu, protože funkci Hammeru využívá modul Chartjs-plugin-zoom. V rámci své vlastní funkce tedy automaticky řeší i tuto. Stačilo jen odkazovat současně na zdrojový soubor Chartjs-plugin-zoom i na Hammer. Sekce v konfiguraci grafu s nastavením pohybu nese název pan, a implementuje se dohromady se sekcí pro zoom (viz obr. 3.18).

```
plugins: {
    zoom: {
        pan: {
            enabled: true,
            mode: 'xy',
            speed: 10
        },
        zoom: {
            enabled: true,
            mode: 'xy'
        }
    }
}
```

Obrázek 3.18: Implementace přibližování a pohybu v grafu

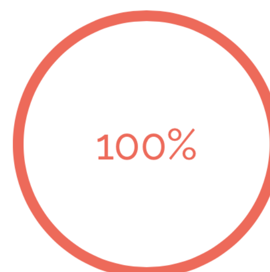
3.6.4 Progressbar.js

V častých případech nastává situace, kdy se graf z nějakého důvodu déle načítá do pohledu. Může to způsobovat například velké množství dat nebo pomalejší počítač. Abych zabránil dočasnému zobrazení prázdné stránky, dokud se graf nezpracuje, rozhodl jsem se naimplementovat dialog s informací, že se graf načítá spolu s indikátorem stavu procesu načítání. K provedení záměru jsem si vyhledal k tomu určený jménem Progressbar. Tento nástroj podporuje zobrazení indikátoru v nejrůznějších tvarech, od klasické čáry přes geometrické tvary, jako jsou kruh, půlkruh nebo čtverec, až po samostatně definované tvary. Číselně sděluje stav načítání v procentech, v případě nahrávání určitého počtu dat pak dokáže zobrazovat i průběžný počet

nahraných dat. Od ostatních výše i níže popisovaných modulů se odlišuje tím, že jsem ho naprogramoval jakožto samostatně stojící objekt, který jsem měl poté propojit s přiřazeným grafem. Bohužel úskalí této metody spočívalo v tom, že se mi nikdy nepodařilo synchronizovat zobrazovaný průběh zpracovávání a načítání grafu se skutečnou dobou procesu. Ve výsledku indikátor průběhu načítání v procentech doběhl do sta procent, jenže graf stále nebyl připravený k vykreslení.

```
var bar = new ProgressBar.Circle(container, {
  color: '#ED6A5A',
  strokeWidth: 4,
  trailWidth: 1,
  from: { color: '#aaa', width: 1 },
  to: { color: '#ED6A5A', width: 4 },
  step: function(state, circle) {
    circle.path.setAttribute('stroke', state.color);
    circle.path.setAttribute('stroke-width', state.width);
    var value = Math.round(circle.value() * 100)+"%";
    if (value === 0) {
      circle.setText('');
    } else {
      circle.setText(value);
    }
  }
});
bar.animate(1.0);
```

Obrázek 3.19: Implementace kruhového progressbaru



Obrázek 3.20: Výsledné zobrazení dialogu

3.6.5 Momentjs

V ukázkové aplikaci zobrazuji v některých grafech časovou osu, kdy každý bod reprezentuje stav sledované veličiny v čase. Do logiky programu jsem načtená data uložil ve formátu DD-MM-YYYY hh:mm:ss. Stejný tvar jsem poté zapsal do polí při převodu dat mezi C# a JavaScriptem, a poté předal do grafu k vykreslení jako popisek osy X. Tím vznikl dost nepřehledný graf, kdy se kvůli vysoké hustotě dat popisky překrývaly, nebo se nezobrazily všechny. Z části pomohlo změnit datový typ zobrazovaných veličin ze string na DateTime. Ten se ale špatně vykresloval při přiblížení grafu, protože nezobrazil časové hodnoty u bodů, které byly kvůli zaokrouhlení na vyšší časové jednotky skryty.

Vzniklý problém jsem vyřešil vložením modulu Moment.js do projektu. Jde o datový typ určený k manipulaci s časovými veličinami. Pro každý okamžik záznamu jsem si tedy vytvořil ze stringu moment, a až ten jsem pak vložil do grafu. Tím se vyřešily všechny zmíněné problémy. Jedinou věc, kterou nebyl modul schopný vyřešit, byla změna zobrazení času z dvanáctihodinového formátu na čtyřiadvacetihodinový. Všechny grafy tedy vykreslují pouze dvanáctihodinové časové hodnoty.

```
function createRegulatoryDeviationChart(dates, expectedData, data) {
    var labels = [];
    for (var i in dates) {
        labels.push(moment(dates[i], "dd.MM.yyyy hh:mm:ss"));
    }
```

Obrázek 3.21: Implementace momentu ve chtěném formátu

3.7 Export reportu

Nyní už aplikace obsahuje celistvou sadu funkcí pro zobrazení plnohodnotného reportu o stavu energetické sítě a účinnosti kompenzace jalového výkonu. Uživatel však může chtít report nejen vidět, ale i si ho z různých důvodů ukládat. Proto jsem do programu přidal možnost exportu grafů i tabulek. Vyřešení tohoto úkolu se stalo jednou z nejsložitějších věcí při vývoji celého programu. Export samotných dat z tabulek nebyl těžký úkol, JavaScript dokáže exportovat tabulky do excelovského formátu bez jakýchkoli doplňujících nástrojů. Uložit grafy nebo dokonce grafy společně s tabulkami už nezvládá.

Při tvorbě pohledů pomocí Chart.js jsem se snažil najít jakýkoli javascriptový modul, který by umožňoval exportovat pohromadě grafy i tabulky s možností pozicování elementů do formátu A4 stránek pro případ tisku. Bohužel, žádný takový modul (aspoň zatím) neexistuje. Zkusil jsem využít moduly popsané v kapitole 1.2.1, žádný však nesplňoval všechny požadavky na funkce, které jsem potřeboval, hlavně z důvodu neschopnosti automaticky pozicovat rozvržení do formátu A4. Jako další možnost jsem se pokusil propojit projekt s designérem knihovny DevExpress. Nešlo ale přenést grafy z canvasů do DevExpressu, tuto funkci designér nepodporuje. Nebyl jsem tedy schopný naplnit předsevzatý cíl. Jako jediná možnost mi zbylo exportování jednotlivých výstupů aplikace do běžných formátů, jako jsou PDF, JPEG, EXCEL aj. K tomu účelu posloužil modul Html2pdf. Podobně jako u programování grafů jsem si mohl upravit nastavení generovaného souboru v atributu options, kde jsem si definoval jméno a typ generovaného souboru. Poté jsem generátoru předal spolu s atributem options ukládaný element a knihovna se už postarala o samotný export. Celou implementaci přikládám k nahlédnutí na obrázku 3.22.

```
function exportToPdf() {
    var elements = document.getElementById('content');
    var name = document.getElementById('jmenoSouboru').innerText;
    var options = {
        margin: 1,
        filename: name + '-report.pdf',
        image: { type: 'jpeg', quality: 1 },
        html2canvas: { scale: 2 },
        jsPDF: { unit: 'in', format: 'letter', orientation: 'portrait' }
    };
    html2pdf().set(options).from(elements).save();
```

Obrázek 3.22: Implementace exportu do formátu JPEG

Knihovna DevExtreme umožňuje situaci řešit snadněji. Ani k exportu elementů nepotřebuje doplnění o modul třetí strany i tuto funkci má zabudovanou v základní verzi. Nástroj jsem implementoval prostým připsáním krátkého příkazu ve formě `.Export(e => e.Enabled(true))` do definice grafu či tabulky na stejnou úroveň, v jaké se nachází například `.DataSource()` (viz obr. 3.13 či obr. 3.15). Po přidání tohoto kusu kódu se u příslušného grafu objevila ikonka umožňující export elementu do formátů PNG, JPEG, PDF a SVG nebo dokonce i rovnou tisk. U tabulek pak obdobná ikonka umožňuje export všech či pouze vybraných řádků do excelovského

formátu XLSX. DevExtreme nabízí také možnost exportu pomocí jQuery. Tento postup se mi jevil variabilnější, mohl jsem totiž pomocí jediného tlačítka exportovat i více elementů do jednoho souboru, a nastavit jméno a formát generovaného souboru. Opět jsem však narazil na neduh postihující javascriptové moduly. Tentokrát se mi sice nedělily grafy přes více stránek, výsledný soubor ale definoval výšku a šířku zobrazení podle rozměrů vkládaných elementů. Pokud byl tedy například graf v zobrazení moc široký, nedal se uložit ve formátu A4. Knihovna také pro změnu nedělila stránky vůbec, takže vznikl soubor o jedné stránce, ale s délkou více A4 stran. Ve výsledku tedy při tisku mohla nastat situace, kdy se jakýkoli element zobrazení rozdělí na více stránek. JQuery řešení jsem tedy zavrhl, a realizoval efektivnější první možnost implementace.

3.8 Vytvoření spustitelného souboru

V posledním kroku celé implementace už zbývalo jen vygenerovat spustitelný soubor ke spouštění celého programu. K tomu účelu jsem využil funkci publikování. ASP.NET Core umožňuje publikovat projekt do všech podporovaných systémů, tedy na Windows, Linux i macOS. Mohl jsem specifikovat, jestli chci vytvořit spustitelný soubor určený přímo pro určité prostředí nebo vytvořit multiplatformní soubor. Protože podle zadání měla být aplikace v rámci prostředí .NET Core multiplatformní, rozhodl jsem se pro druhou možnost. K publikování lze využít nástroj v Microsoft Visual Studiu či příkazovou řádku. Já pracoval s příkazovou řádkou.

Publikování obsluhuje terminálový příkaz `dotnet publish` spuštěný ve složce s projektem aplikace. V tomto znění příkazu se vytvoří spustitelný soubor umístěný v `<cesta do projektu>\bin\release\<verze .NET Core>\publish\`. V tuto chvíli jsem nedefinoval kompatibilitu s operačním systémem, vytvořil jsem tedy v rámci .NET Core univerzální program. Pro definování vlastního umístění výsledného souboru stačí za příkaz připsat absolutní cestu do zvolené složky. Pro definování kompatibility jen pro určitý operační systém pak stačí za přikaz připsat `--runtime <označení systému>`.

Po vytvoření spustitelného souboru stačí program už jen spustit. Jelikož jde o webovou aplikaci, běží pouze na serveru a přistupuje se k ní v prohlížeči zadáním její adresy na serveru. Pro spuštění se ve složce s aplikací musí v terminálu zadat příkaz `dotnet <jméno aplikace>.dll`. Tím se spustí lokální Kestrel server (viz kapitolu 1.1.3) s uloženým programem a v okně toho samého terminálu se zobrazí HTTPS adresa, pomocí které lze k programu přistoupit a začít s ním pracovat. V mém případě byla aplikace umístěna na `https//localhost:5000`.

4 Vyhodnocení řešení

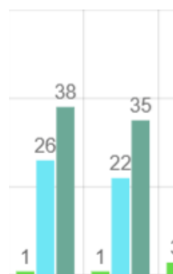
Při realizaci této práce jsem byl nucen realizovat dva postupy návrhu uživatelského rozhraní aplikace. Jako přirozená volba při návrhu MVC architektury se mi jevilo na základě dosavadních zkušeností jako nejlepší cesta implementovat pohledy pomocí JavaScriptu, respektive knihovny Chart.js. Postupem času mi však začalo docházet, že zvolené řešení není ani zdaleka snadné ani efektivní. Spousta potřebných nástrojů a funkcí v balíku chyběla a já pak byl nucen dohledávat externí nástroje, které by funkce doplnily. S tím přišlo mnoho problémů při integraci do knihovny Chart.js. Některé implementace i přes použití syntaxe doporučované v dokumentacích nefungovaly, jindy při integraci jednoho balíku přestal fungovat jiný apod. Celková doba programování pohledů se tak nakonec protáhla na několik týdnů.

I v konečné podobě se pak aplikace potýkala s několika neduhy. HTML element canvas by sice podle dokumentace měl být kompatibilní se všemi rozšířenými internetovými prohlížeči, tedy s Google Chrome, Safari, Operou, Mozillou i Microsoft Edge, ve výsledku ale často fungovala v každém klientovi odlišně. Zejména Google Chrome je známý tím, že při svém běhu zabírá veliké množství operační paměti. Samotný program pak také využívá více místa v paměti, protože zpracovává veliké množství dat. Při načítání aplikace se tedy zobrazovaly animace trhaně, plynule vše probíhalo pouze při načtení v Mozille Firefox. Kromě toho se pak v prohlížeči Microsoft Edge nezobrazovalo správně či vůbec CSS stylování.

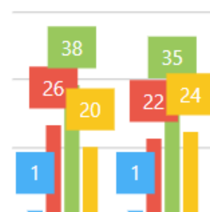
Nejfatálnější problém s tímto řešením však nastal při ukládání reportu. Po neúspěchu s exportem (viz kapitolu 3.7) jsem se rozhodl využít designér NuGet modulu DevExpress (viz kapitolu 1.2.2). Po propojení projektu s designérem jsem však zjistil, že DevExpress komunikuje pouze s modelem aplikace a nelze žádným způsobem přenést grafy z pohledových tříd do vytvářeného reportu. Po konzultaci s vedoucím práce jsem došel k závěru, že implementace programu za použití knihovny Chart.js je slepou uličkou a že tímto způsobem nelze dojít k uspokojivému výsledku.

Na základě doporučení vedoucím jsem jako náhradu zvolil knihovnu DevExtreme. V čem jsou oproti Chart.js výhody tohoto modulu, vysvětluji v kapitole 3.5.2. S implementací tohoto modulu se doba tvorby zkrátila oproti několika týdnům s Chart.js na pouhé dny. DevExtreme totiž disponuje přehlednou, rozsáhlou a hlavně ucelenou dokumentací. Výsledná aplikace také pracuje plynule ve všech prohlížečích bez rozdílu. Jedinou větší nevýhodou, kterou tato knihovna má, je cena. DevExpress stojí

ve verzi pouze s nástroji pro ASP.NET okolo tisíce dolarů, samotný DevExtreme pak sedm set. Pro potřeby bakalářské práce jsem využil třiceti denní bezplatnou demo verzi, pro profesionální užívání by ale určitě bylo záhodnější koupit plnou verzi se všemi nástroji a funkcemi. Rozdíly se však nedaly nalézt pouze v přístupu k implementaci, ale i v kvalitě samotných výstupů aplikace. Graficky vykresluje DevExtreme ostřejší, kvalitnější a uživatelsky přívětivější zobrazení než Chart.js. Jednotlivá vykreslení porovnávám na obrázcích 4.1 a 4.2 níže.



Obrázek 4.1: Příklad zobrazení grafu v Chart.js



Obrázek 4.2: Příklad zobrazení grafu v DevExtreme

Ani s knihovnou výsledná aplikace bohužel nepodporuje exportování celkového reportu. Opět jsem se snažil propojit projekt s designérem DevExpress, bohužel pro tuto funkci již z časových důvodů nezbyl prostor. Do budoucna však nabízí daleko větší perspektivu pro další rozvoj aplikace. Kromě propojení s designérem by se do budoucna nabízela možnost propojení s jakýmkoli databázovým systémem, čímž by se odbourala nutnost ručně načítat soubory k reportu. Dále by se lehce daly přidat další sledované veličiny do aplikace.

Závěr

Cílem bakalářské práce se stalo seznámení s platformou .NET Core a její využitelností při tvorbě sestav. V teoretické části jsem tedy jako první shrnul specifikace platformy relevantní při tvorbě reportů na vícero operačních systémech. Následně jsem definoval pojem jalový výkon a jeho kompenzaci. Představil jsem průběh kompenzace, veličiny sledované při kompenzaci a důvody vedoucí k zájmu o toto téma obecně.

V praktické části jsem využil získané informace implementováním ukázkové aplikace na platformě .NET Core. Program měl zpracovávat reálná data ze zařízení sledující kompenzaci elektrické sítě, a vytvořit report o účinnosti kompenzace. Nejprve jsem představil návrh řešení, kterým se stala webová aplikace s architekturou MVC. Popsal jsem postup při volbě architektury a operačního systému, následně pak jednotlivé kroky implementace od návrhu logiky, propojení s uživatelským rozhraním (dále už jen UI), sestavení UI až po vkládání jednotlivých elementů reportu do programu. Zmínil jsem se také o všech použitých externích modulech, ať už se nacházely v konečné verzi programu nebo ne. V poslední části popisu řešení jsem představil způsob vytvoření spustitelného souboru na různé platformy.

Na závěr jsem porovnal dva použité způsoby implementace. Zhodnotil jejich náročnost, účinnost, využitelnost a rozšiřitelnost v eventuálním praktickém využití, a následně tím odůvodnil výběr jednoho ze dvou porovnávaných postupů pro finální verzi aplikace.

Literatura

[1] ROTH, D., ANDERSON R., LUTTIN, S. Úvod do ASP.NET Core. docs.microsoft.com [online]. © Microsoft 2019 [vid. 14. 02. 2019]
Dostupné z https://docs.microsoft.com/cs-cz/aspnet/core/?view=aspnetcore-2.2

[2] WATSON, MATT. Top 13 ASP.NET Core Features You Need to Know. stackify.com [online]. © 2012–2019 [vid. 31. 10. 2017]
Dostupné z: https://stackify.com/asp-net-core-features/

[3] KOOPMANS, Erik. Client-side HTML-to-PDF rendering using pure JS. github.com [online] © 2017-2019
Dostupné z: https://github.com/eKoopmans/html2pdf.js

[4] PDF.js dokumentace. mozilla.github.io/pdf.js [online]. © Mozilla
Dostupná z: https://mozilla.github.io/pdf.js/

[5] DevExpress. devexpress.com [online] © 1998-2019 Developer Express Inc.
Dostupné z: https://www.devexpress.com/support/demos/

[6] DevExtreme. js.devextreme.com [online] © 2011-2019 Developer Express Inc.
Dostupné z: https://js.devexpress.com/Demos/WidgetsGallery/

[7] Reactive Power. electronics-tutorials.ws [online] © 2019 AspenCore, Inc.
Dostupné z: https://www.electronics-tutorials.ws/accircuits/reactive-power.html

[8] KOŠŤÁL, Josef. Kompenzace elektrického jalového výkonu. odbornecasopisy.cz [online]. © 2014-2019 FCC Public s.r.o.
Dostupné z: http://www.odbornecasopisy.cz/elektro/casopis/tema/kompenzace-elektrickeho-jaloveho-vykonu--11073

[9] MAJDA, František. Individuální kompenzace jalového výkonu. odbornecasopisy.cz [online]. © 2014-2019 FCC Public s.r.o.
Dostupné z: http://www.odbornecasopisy.cz/elektro/casopis/tema/individualni-kompenzace-jaloveho-vykonu--11095

[10] Základy kompenzace. kbh.cz [online]. © 2007 - 2019 KBH Energy a.s.
Dostupné z: http://www.kbh.cz/o-kompenzaci/zaklady-kompenzace

[11] Compensation of reactive power. sofelektro.rs [online]
Dostupné z: http://sofelektro.rs/kompenzacija-reaktivne-snage/?lang=en

[12] Třífázové regulátory jalového výkonu a síťové analyzátory NOVAR 2600. kmb.cz [online] © 2011-2019 KMB systems s. r. o.
Dostupné z: http://kmb.cz/index.php/cs/component/phocadownload/category/4-docspfc?download=211:novar-2600-manual-k-pristroji

[13] Stykače pro spínání kondenzátorů. benediktcz.cz [online] © 2019 Benedikt Gmbh
Dostupné z: http://www.benediktcz.cz/index.php/produkty/17-stykace-pro-spinani-kondenzatoru

[14] Cenové rozhodnutí Energetického regulačního úřadu č. X/2015 ze dne XX. listopadu 2015, kterým se stanovují ceny za související službu v elektroenergetice a další regulované ceny. eru.cz [online]
Dostupné z: https://www.eru.cz/documents/10540/1659899/CR_ERÚ_2016_elektro_151026.pdf/72583bf7-f697-4765-a006-74318027a239

[15] PERONNET, Jacques. Power factor correction kvar policy in countries. engineering.electrical-equipment.org [online]. © 2009–2019 [vid. 07. 05. 2009]
Dostupné z: http://engineering.electrical-equipment.org/power-quality/reactive-power-billing-policy-in-countries.html

[16] SMITH, Steve. Přehled ASP.NET Core MVC. docs.microsoft.com [online]. © Microsoft 2019 [vid. 08. 01. 2018]
Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/mvc/overview?view=aspnetcore-2.2#web-apis

[17] Chart.js. chartjs.org [online] © The MIT License
Dostupné z: https://github.com/chartjs/Chart.js

[18] Plugin for Chart.js to display percentage, value or label in Pie or Doughnut. github.com [online].
Dostupné z: https://github.com/emn178/chartjs-plugin-labels

[19] Zoom and pan plugin for Chart.js. github.com [online]. © The MIT License
Dostupné z: https://github.com/chartjs/chartjs-plugin-zoom

[20] Hammerj.js. hammerjs.github.io [online]. © The MIT License
Dostupné z: https://github.com/hammerjs/hammer.js/tree/master/

[21] Progressbar.js. kimmobrunfeldt.github.io/progressbar.js/ [online].
Dostupné z: https://github.com/kimmobrunfeldt/progressbar.js/

[22] Moment.js. momentjs.com [online]. © The MIT License
Dostupné z: https://momentjs.com

[23] Publikování DotNet. docs.microsoft.com [online]. © Microsoft 2019 [vid. 29. 05. 2018]
Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/core/tools/dotnet-publish?tabs=netcore21

Přílohy

Graf regulacni odchylky od jaloveho vykonu

v Kvar

Ocekavany jalovy vykon

Regulacni odchylka od jaloveho vykonu

60
40
20
0
-20
-40
-60

1 12:00 AM
3:00 AM
6:00 AM
9:00 AM
12:00 PM
3:00 PM
6:00 PM
9:00 PM
2
3:00 AM
6:00 AM

12:00 AM
8:22 AM

12:00 AM
4:00 AM
8:00 AM
12:00 PM
4:00 PM
8:00 PM
12:00 AM
4:00 AM
8:00 AM

1
Tuesday, 2

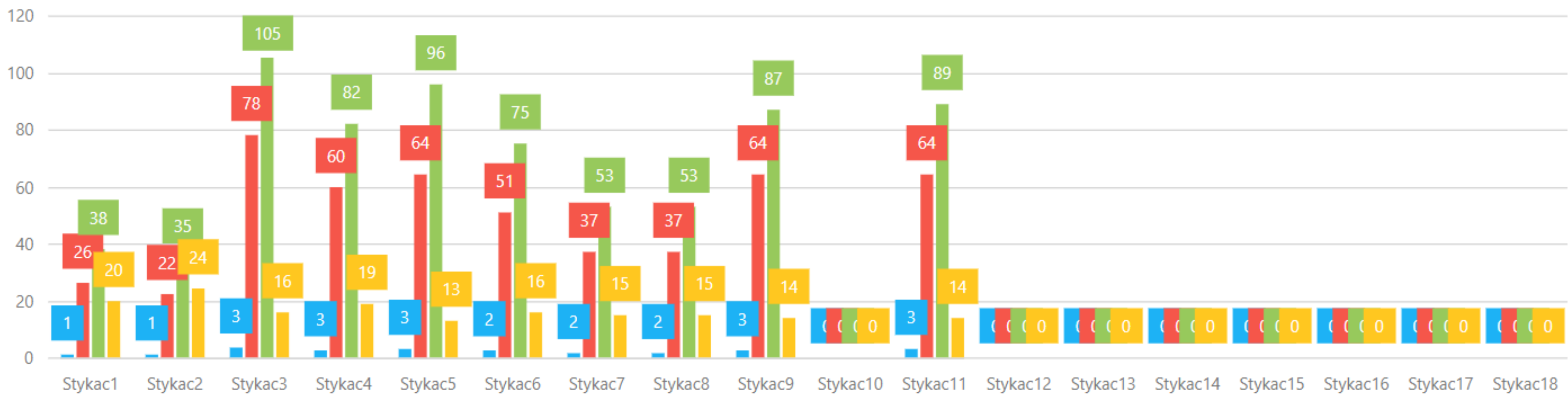
Graf zprumerovane regulacni odchylky

v Kvar

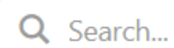
Drag a column header here to group by that column

Search...

	Sledovana Velicina	Cas	Hodnota	Jednotka
	Jalová dodávka		184,013	Kvarh
	Nejvyšší regulační odchylka	01.10.2018 11:38:45	41,93	Kvar
	Nejnižší regulační odchylka	01.10.2018 11:35:00	-47,4	Kvar
	Průměrná regulační odchylka		1,808	Kvar


Graf vyuziti jednotlivych stykacu
v počtu sepnutí
Prumer za hodinu
Prumer za den
Celkovy pocet
Doba provozu [hod]
120
100
80
60
40
20
0
1
26
38
20
1
22
35
24
3
78
105
16
3
60
82
19
3
64
96
13
2
51
75
16
2
37
53
15
2
37
53
15
3
64
87
14
0
3
64
89
14
0
0
0
0
0
0
0
Stykac1
Stykac2
Stykac3
Stykac4
Stykac5
Stykac6
Stykac7
Stykac8
Stykac9
Stykac10
Stykac11
Stykac12
Stykac13
Stykac14
Stykac15
Stykac16
Stykac17
Stykac18

Drag a column header here to group by that column

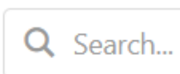


	Stykac	Pocet sepnuti celkem	Pocet sepnuti za sledovani	Pocet sepnuti za den 1	Cas v provozu celkem	Stupne
	Stykac1	2943	38	26	1567	95,33
	Stykac2	2943	35	22	1387	95,16
	Stykac3	8006	105	78	1452	46,98
	Stykac4	9907	82	60	1852	23,71
	Stykac5	15570	96	64	2025	12,25
	Stykac6	14765	75	51	2212	6,51
	Stykac7	12052	53	37	2085	3,00
	Stykac8	12039	53	37	2106	2,00
	Stykac9	9911	87	64	1619	23,32
	Stykac10	6	0	0	0	0

1 2

Odchylka uciniku od ocekavane hodnoty

Drag a column header here to group by that column



Sledovana velicina	Cas	Skutecna hodnota	Hodnota odchylky
Očekávaná hodnota účiníku		1	
Průměrný účiník		1	
Maximální povolená odchylka		0.95	0.05
Nejvyšší indukční odchylka	01.10.2018 02:05:30	0.989	0.011
Nejnižší indukční odchylka	01.10.2018 12:00:00	1	0
Nejvyšší kapacitní odchylka	02.10.2018 05:00:15	0.992	0.008
Nejnižší kapacitní odchylka	01.10.2018 12:00:00	1	0