VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

SOFTWAROVÉ MODULY PRO VIZUALIZACI STAVŮ SYSTÉMU A JEHO HISTORIE

SOFTWARE MODULES FOR SYSTEM STATE AND HISTORY VISUALISATION

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Martin Kubásek

AUTHOR

VEDOUCÍ PRÁCE — Ing. Václav Kaczmarczyk, Ph.D.

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. Martin Kubásek ***ID:*** 203270

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Softwarové moduly pro vizualizaci stavů systému a jeho historie

POKYNY PRO VYPRACOVÁNÍ:

1) Seznamte se s existujícím systémem pro sběr dat.
2) Navrhněte a implementujte množinu modulů pro vizualizaci (zobrazení proměnné, graf, tabulka historických hodnot, vícestavové zobrazení)
3) Navrhněte a realizujte webovou aplikaci v jazyce c# a frameworku DOTVVM, která bude tyto moduly využívat.

DOPORUČENÁ LITERATURA:

Gejza Dohnal: Teorie hromadné obsluhy.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 17.5.2023

Vedoucí práce: Ing. Václav Kaczmarczyk, Ph.D.

doc. Ing. Petr Fiedler, Ph.D.
předseda rady studijního programu

ABSTRAKT

Tato práce se zabývá tvorbou webové vizualizace, která je schopná zobrazovat stavy libovolného systému v reálném čase a jeho historická data. Na začátku je popsán stávající systém vizualizace sloužící jako předloha při tvorbě nové aplikace. Byly identifikovány nedostatky a omezení stávajícího řešení, které nový systém nakonec vyřeší. Poté proběhl rozbor technologií, návrhových vzorů a designových principů, vhodných pro tvorbu nového modulárního systému vizualizace. Zároveň byly uvedeny i další technologie, které by bylo možné použít jako alternativu. Následuje návrh nové webové aplikace na platformě DotVVM, kde je popsána struktura, základní myšlenky a tok dat v systému. Podařilo se vytvořit systém, který by měl umožnit implementaci téměř libovolného softwarového modulu jako vizualizačních indikátorů. Tyto moduly pak lze snadno napojit na libovolný datový model, pomocí systému datových bodů a výrazů. Nový systém je tak více modulární a lépe spravovatelný do budoucnosti. Vzniklo několik modulů pro zajištění základní funkce jako vizualizace, jako například numerický, binární a stavový indikátor. Zobrazení historických dat umožňuje modul grafu. Nová aplikace tvoří funkční systém vizualizace a zajišťuje navíc funkce jako správu a editaci vizualizačních scén, systém přihlašování a systém notifikací pro uživatele.

KLÍČOVÁ SLOVA

Vizualizace, P&ID, Web, .NET, C#, DotVVM, ASP.NET, MVVM, Dependency Injection, React

ABSTRACT

Překlad abstraktu (This work deals with the creation of a web visualization that is capable of displaying the states of any system in real time and its historical data. At the beginning, the existing visualization system serving as a template for the creation of a new application is described. The shortcomings and limitations of the existing solution were identified, which the new system will eventually solve. Next is an analysis of technologies, design patterns and design principles suitable for the creation of a new modular visualization system. At the same time, other technologies were presented that could be used as an alternative. After that a new web application on the DotVVM platform is presented, where the structure, basic ideas and data flow in the system are described. It was possible to create a system that should enable the implementation of almost any software module as visualization indicators. These modules can then be easily connected to any data model, using a system of data points and expressions. The new system is thus more modular and more manageable for the future. Several modules have been created to provide basic functionality as visualization, such as numeric, binary and status indicator. The graph module enables the display of historical data. The new application forms a functional visualization system and additionally provides functions such as management and editing of visualization scenes, a login system and a notification system for users.

KEYWORDS

Překlad klíčových slov (Visualization, P&ID, Web, .NET, C#, DotVVM, ASP.NET, MVVM, Dependency Injection, React)

Vysázeno pomocí balíčku `thesis` verze 4.07; `http://latex.feec.vutbr.cz`

KUBÁSEK, Martin. *Softwarové moduly pro vizualizaci stavů systému a jeho historie*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2023, 66 s. Diplomová práce. Vedoucí práce: Ing. Václav Kaczmarczyk, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Martin Kubásek
VUT ID autora:	203270
Typ práce:	Diplomová práce
Akademický rok:	2022/23
Téma závěrečné práce:	Softwarové moduly pro vizualizaci stavů systému a jeho historie

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno 15.5.2023 .....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce panu Ing. Václavu Kaczmarczykovi, Ph.D. za odborné vedení, přímý přístup, konzultace a podnětné návrhy k práci.
Zároveň chci poděkovat své rodině a přítelkyni za materiální a psychickou podporu po dobu psaní této práce a celého studia.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

Tato práce se zabývá tvorbou zcela nového vizualizačního systému jako webové aplikace postavené na DotVVM a ASP.NET Core. Aplikace bude umožňovat zobrazování stavů sledovaného procesu v reálném čase a zobrazování dat z jeho historie. Při návrhu nové aplikace bude dbáno na modularitu, rozšířitelnost a jednoduchou spravovatelnost při budoucím rozšiřování.

Tvorba bude vycházet z již existujícího sytému pro vizualizaci, který by měl být novou aplikací v budoucnu nahrazen. Důvodem jsou problémy se správou kódu a nasazených systémů, plynoucí ze specifických úprav pro každý vizualizovaný proces. Ty vytvořily nutnost při každé malé úpravě požadavků měnit kód aplikace a vytvoření nové nekompatibilní verze.

Nový systém by měl být dostatečně modulární na to, aby bylo možné tento problém překonat a aby každá nová instalace zahrnovala minimum programátorské práce. Zároveň by měl být co nejvíce kompatibilní s datovým modelem, který používá existující systém a měl by implementovat všechny jeho stávající funkce, nebo být připraven na jejich jednoduchou implementaci.

Pro dosažení těchto cílů budou jednotlivé části vizualizace odděleny jako modulární softwarové moduly, umožňující jednoduchý vývoj a následnou správu. Zároveň bude nutné postavit nový systém tak, aby ho bylo možné snadno napojit na libovolný datový model.

Na začátku práce bude proveden rozbor stávajícího systému a budou ukázány některé jeho funkce a omezení. Poté budou popsány technologie a návrhové vzory, použité v rámci vývoje nového systému, nebo které by bylo možné použít jako alternativu. Na závěr bude popsán návrh nové aplikace, napojení na datový model a její realizace. Vznikne několik softwarových modulů realizující základní vizualizační funkce.

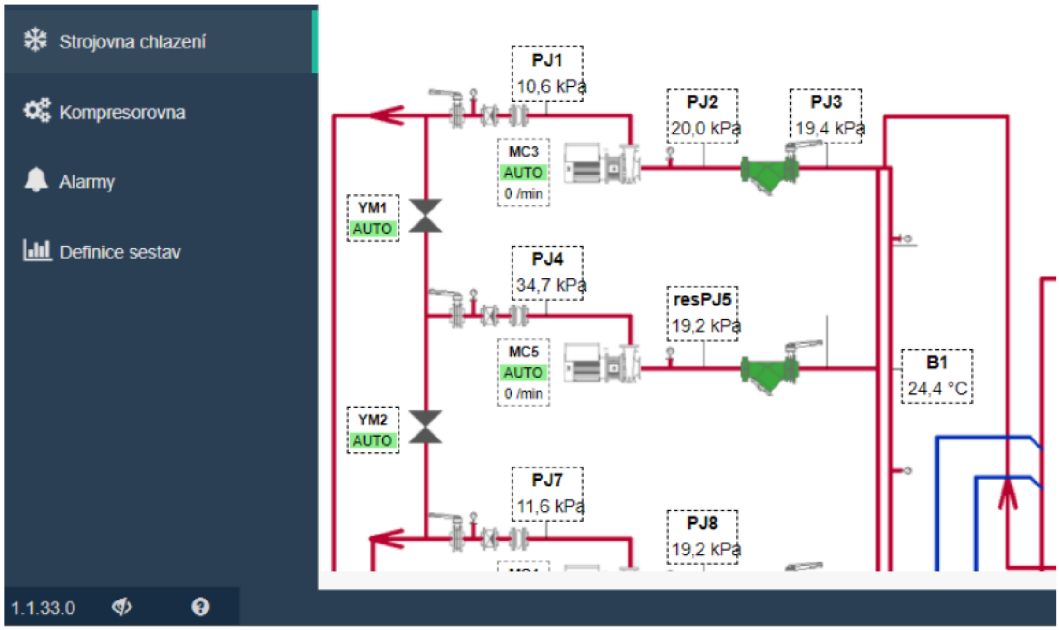
1 Stávající systém

V této kapitole bude popsán stávající systém vizualizace a místa, na které bude vhodné se zaměřit při tvorbě nové vizualizace. Stávající systém vznikl jako jednorázový projekt pro továrnu, kde zprostředkovává náhled na technologie kompresorovny a systému chlazení. Zároveň umožňuje nastavování některých těchto prvků.

1.1 Funkce

Do vizualizace je nutné se přihlásit a jednotlivé funkce jsou pak zpřístupněny na základě rolí uživatele. Správu uživatelů a jejich rolí zajišťuje externí systém a není možné je spravovat přímo ve vizualizaci.

Pro každou z částí technologie jsou definovány takzvané scény. Scéna vizualizace je prakticky P&ID diagram doplněný o aktuální hodnoty jednotlivých zařízení. Jak může taková scéna vypadat lze vidět na obrázku 1.1.



Obr. 1.1: Scéna ve stávající vizualizaci

Na scéně je možné definovat následující druhy zařízení:

- `Ventil` – Ukazuje stav (otevřeno/zavřeno/mezipoloha/porucha) a režim ovládání. Po rozkliknutí lze přepínat stav, režim ovládání a kvitovat poruchu.
- `Čerpadlo` – Ukazuje otáčky a režim ovládání. Po rozkliknutí lze přepínat stav, režim ovládání, kvitovat poruchu a nastavovat otáčky.

- `Snímač` – Ukazuje aktuální hodnotu. Po rozkliknutí lze zobrazit historii hodnoty.

Na každé zařízení lze kliknout, přičemž se vždy otevře dialogové okno s detailními informacemi/nastavením. V dialogovém okně je pak možnost zobrazit Archiv dat, což je stránka s historickými daty daného prvku.

Dále je možné přejít na stránku sestav. Zde můžeme definovat různá kritéria pro zobrazení historických dat, která se pak zobrazují jako graf. Tyto data pak můžeme exportovat jako soubor CSV.

1.2 Technický rozbor

Stávající systém je postaven tak, že je vždy udělán na míru konkrétní lokalitě, kde má být nasazen. Pokud bylo potřeba systém nasadit na další lokalitu, vznikla vždy další kopie, obsahující specifické úpravy pro danou instalaci. Ve výsledku tak existuje více verzí toho stejného systému, které nejsou mezi sebou zcela kompatibilní a jsou těžce udržovatelné.

Systém je realizován jako webová aplikace na platformě DotVVM a využívá dnes již zastaralou verzi 2.4[1]. Pro tvorbu webového rozhraní jsou zde využity komerční komponenty DotVVM Bootstrap a BusinessPack. Přístup k datům je realizován pomocí vlastních knihoven, které zprostředkovávají načítání a ukládání dat do MySQL databáze. Tyto knihovny jsou pak sdíleny s dalšími projekty, které se netýkají vizualizace.

V rozboru struktury se zaměříme na strukturu týkající se scén. Z hlediska navigace je struktura jednoduchá, stránky scén přímo dědí z kořenového MasterPage a všechen kód týkající se scén a technologií je koncentrován v této stránce. Komponenty jsou explicitně definovány v kódu stránky „vedle sebe“ jako několik seznamů, které se pak přes sebe vykreslují na stránce.

To v praxi znamená několik set řádků kódu pro každý typ technologie v jednom souboru. Tento kód zajišťuje například načítání dat, zobrazení UI, zobrazení dialogu a ukládání dat. Ve výsledku má ViewModel scény něco přes 1100 řádků kódu a jeho správa vyžaduje mnoho času na zorientování. Zároveň nelze získat dostatečný přehled o vazbách v kódu a bude obtížné zhodnotit důsledky případných změn. Další potencionální problém je náročnost přenosu tohoto ViewModelu. Při použití DotVVM dochází s některými požadavky k serializaci a přenosu tohoto velkého objektu mezi klientem a serverem a je tedy vhodné mít ViewModely co nejjednodušší [14]. Tento problém je naznačen na obrázku 2.3.

[1] Aktuální verze v prosinci 2022 je 4.0

1.3 Návaznost na nový systém

Tato práce se bude primárně zabývat právě částí týkající se scén, realizující náhled na části sledované technologie. Z rozboru je zřejmé, že bude vhodné nový systém navrhnout tak, aby bylo možné jednotlivé vizualizační prvky oddělit jako samostatné softwarové moduly. Ty by pak mělo být možné nasazovat bez ohledu na konkrétní strukturu scény, nebo datový model, na který bude vizualizace navázána. Zároveň díky vyšší modularitě umožní jednodušší softwarový vývoj. Oddělení zdrojových kódů týkající se částí technologie do samostatných souborů nebo projektů umožní vyšší spravovatelnost kódu - bude snazší tyto komponenty udržovat a vylepšovat. Pokud se podaří vytvořit jednotné rozhraní mezi scénou a softwarovým modulem, mělo by být teoreticky možné tyto komponenty jednoduše zaměňovat.

Vzhledem k nutnosti aktualizace využívaných platforem, nástrojů a výrazným změnám v architektuře, nebude upravován stávající systém, ale vznikne zcela nový. Vznikne tedy nová webová aplikace realizující vizualizaci.

Výhodou vytvoření zcela nové verze aplikace je také využití novějších technologií. Bude využita novější verze DotVVM, která již využívá mnohem rychlejší a multiplatformní ASP.NET Core s .NET 6 [14] [7] [19].

Některé části pak budou převzaty ze starého systému a budou případně upraveny aby zapadaly do nové architektury. To se týká například části přístupu k MySQL databázi a přihlašování.

2 Technologie a návrhové vzory

V této kapitole bude popsáno několik významných webových technologií, návrhových vzorů a designových principů. Jsou zde popsány primárně pojmy týkající se vývoje nového systému, ale jsou zde také uvedeny některé pojmy pro poskytnutí ucelenějšího náhledu na vývoj webových aplikací obecně. Většina uvedených pojmů se týká hlavně ekosystému .NET a firmy Microsoft.

2.1 .NET

V prvé řadě je vhodné popsat, co je to samotný .NET[1], jelikož tvoří základ většiny dále popisovaných technologií. .NET je vývojářská platforma uvedená společností Microsoft v roce 2002, která představuje množinu nástrojů, programátorských jazyků a knihoven pro tvorbu aplikací [8]. Z hlediska programátorských jazyků je možné psát aplikace v jazyce C#, *F#* a Visual Basic. Pokud vezmeme v potaz dokumentaci a aktuální podporu komunity, je nejvíce používaný C#[2]. Pravděpodobně díky jeho vysoké podobnosti s jazyky Java a C++. Podobně jako Java jde o objektově orientovaný jazyk, ve kterém není nutné řešit alokaci a dealokaci paměti.

.NET používá takzvanou Common Language Infrastructure a díky tomu umožňuje použití tří různých jazyků vedle sebe. Při kompilaci programu nedochází k převodu na strojový kód, ale na CIL kód, který už je stejný bez ohledu na jazyk. CIL kód pak překládá JIT kompilátor na strojový kód, již přizpůsobený cílové HW platformě, na které má aplikace běžet. Díky tomu je .NET velice flexibilní a umožňuje psát kód pro širokou škálu platforem a aplikací.

V ekosystému .NET se vyskytuje více verzí této platformy, které je nutné sledovat:

- `.NET Framework` – Takto se nazývají první verze této platformy. Tyto verze jsou pevně spjaty s Windows a procesory x86. Provozování na jiných zařízeních bylo spíše experimentální. Vzhledem k vysokému rozšíření aplikací, které využívají tuto verzi, budou pravděpodobně ještě dlouho podporovány (minimálně do roku 2029 [9]). Není doporučeno ji používat při tvorbě nových aplikací. Poslední verze v roce 2022 je 4.8.1.
- `.NET` – Nové verze této platformy se už jmenují pouze „.NET“. V určité chvíli došlo ke kompletnímu přepsání knihoven uvnitř platformy tak, aby je bylo možné provozovat i na jiných operačních systémech a architekturách procesoru (ARM). Tyto verze jsou zároveň o dost rychlejší než předchůdci [19]. Na konci roku 2022 vyšla verze .NET 7.

[1] Vyslovováno „dot net“

- `.NET Core` – Tvoří mezistupeň mezi `.NET` a `.NET Framework`. Žádná z verzí již není podporována. Tyto aplikace lze ale lehce aktualizovat aby používaly nejnovější .NET 7.
- `.NET Standard` – Umožňuje tvořit knihovny, které lze sdílet mezi .NET a .NET Framework. Poskytuje pouze základní funkce platformy .NET, takže nelze použít pro tvorbu složitějších knihoven.

2.2 ASP.NET a ASP.NET Core

Pojmem ASP.NET (Core) se označuje soubor nástrojů a platforem pro hostování webových aplikací s využitím .NET. Tyto platformy umožňují vytvářet serverové aplikace, které pak mohou hostovat webové aplikace. Kód lze psát v jednom z podporovaných jazyků, jako je *C#* nebo F#. Tyto serverové aplikace mohou využívat všechny knihovny a přejímat kód z jiných .NET projektů. To může být zvlášť výhodné, pokud již programujeme v .NET, nebo máme nějaké existující aplikace pro desktop, které chceme přesunout na web.

Historicky existoval nejdříve `ASP.NET`, který využíval .NET Framework a byl tak vázán na Windows. S příchodem .NET Core vznikl `ASP.NET Core`, využívající novější verze .NET nevázané na Windows. je díky tomu nativně multiplatformní a zároveň podstatně rychlejší. Tyto dvě verze existují a jsou podporovány vedle sebe, ale znovu platí, že nové aplikace by měly vznikat pouze pro Core verzi.

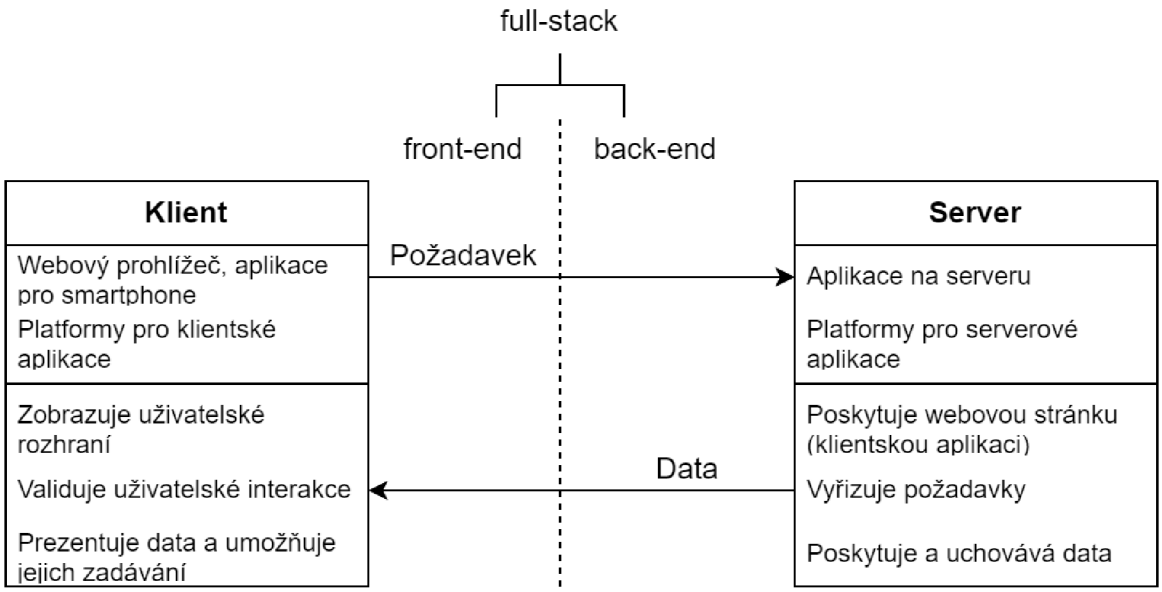
Obecně můžeme ASP.NET používat pouze jako serverovou aplikaci, se kterou komunikují libovolní klienti přes REST API. Zároveň ale lze použít ASP.NET jako takzvané full-stack řešení, kdy je i klientská část vytvářená v ASP.NET. Pro obě verze existují nástroje pro tvorbu webových aplikací, které jsou zde přímo integrovány (MVC, Blazor, WebForms). Ty budou popsány v následující kapitole.

2.3 Platformy pro tvorbu webových aplikací

2.3.1 Úvod

V následujících částech budou mnohokrát zmíněny pojmy jako klientská a serverová aplikace nebo platforma. Proto bude vhodné na úvod popsat, kde se co vyskytuje, jaké jsou úlohy jednotlivých stran a jaké jsou mezi nimi vazby.

Na obrázku 2.1 je diagram popisující zcela obecně klientskou a serverovou část webových aplikací. Vypsané funkce jsou pouze ty relevantní pro tuto práci. Vzhledem k rozmanitosti existujících webových aplikací nelze spolehlivě popsat všechny možné



Obr. 2.1: Obecný popis vztahu klient-server u webových aplikací

funkce. Dále je nutné zmínit, že tento diagram popisuje pouze klienta, kterého přímo ovládá člověk. Klientem ale může být i jiná webová služba bez uživatelského rozhraní.

Jako webovou aplikaci zde označujeme webovou stránku plnící složitější úlohu než pouze zobrazování jednoduchých informací (textu, obrázků). Její rozhraní je blíže desktopové aplikaci než běžné stránce, protože umožňuje složitější interakce s uživatelem. To zahrnuje například zadávání dat, navigaci, pokročilejší prohlížení dat, dialogová okna a podobné.

Jako platformy zabývající se tvorbou klientských aplikací tedy označujeme nástroje umožňující tvorbu aplikací s uživatelským rozhraním. Tyto aplikace mohou lokálně uchovávat data, ale většinou budou tvořit rozhraní pro složitější logiku, která se odehrává někde na vzdáleném serveru. Z toho důvodů se této části přezdívá „front-end“.

Serverové aplikace běží zpravidla někde na vzdáleném serveru bez přístupu uživatele a vyřizují požadavky klientů. Mohou být připojeny k databázi pro trvalé ukládání dat, nebo třeba i připojeny přímo k nějaké technologii (např. PLC). Tyto aplikace jsou prakticky schované uživateli a přezdívá se jim „back-end“.

Pokud platforma umožňuje tvorbu obou částí, hovoříme o full-stack řešení. Existuje mnoho programovacích jazyků používaných v těchto technologiích. Vzhledem k zadání této práce se omezíme na jazyky týkající se .NET a JavaScript. Je také vhodné poznamenat, že všude kde se používá JavaScript se lze setkat s jazykem TypeScript, který dlouhodobě roste v oblibě [2]. Ten slouží jako nadstavba zaručující typovou bezpečnost a dává tak programátorům větší jistotu při tvorbě komerčních

(enterprise) aplikací založených na JavaScript platformách [4].

U klientských platforem také rozlišujeme, jestli je uživatelské rozhraní vykreslováno na serveru, nebo až u klienta. Oba přístupy mají své výhody a nevýhody. Pokud vykreslujeme uživatelské rozhraní na serveru, dochází při vyřizování požadavku k vytvoření kompletního HTML (+ CSS) souboru, který je odeslán klientovi připraven k zobrazení. Většinou tak hovoříme o „tradičních webových aplikacích"[18]. Vykreslování na serveru má následující výhody a nevýhody:

- Malé nároky na klientské zařízení a rychlejší vykreslení v prohlížeči. Prohlížeč nemusí podporovat JavaScript ani WebAssembly.
- Snadné řešení autentizace - přistup k částem stránky a k datům.
- Každá aktualizace uživatelského rozhraní zahrnuje požadavek na server.
- Vyšší nároky na server.

Pokud použijeme vykreslování na klientovi, jsou v prohlížeči vykonávány skripty, které přímo modifikují stránku (DOM) za běhu [7]. Hovoříme pak o takzvaných single-page webových aplikacích (SPA). Tyto řešení bývají založeny na JavaScript nebo WebAssembly [18]. Vykreslování na klientovi má následující výhody a nevýhody:

- Aktualizace uživatelského rozhraní většinou nevyžaduje dotaz na server a děje se přímo u klienta, takže se stránka jeví uživateli jako více responzivní.
- Nižší nároky na server.
- Vyšší nároky na klientské zařízení a rychlejší vykreslení v prohlížeči. Prohlížeč musí podporovat JavaScript nebo WebAssembly.
- Lze implementovat i offline režim aplikace.
- Delší čas k zobrazení stránky - ke klientovi je nutné nejdříve stáhnout všechny soubory a skripty, až poté je stránka vykreslena. Tento efekt ale dnes umí pokročilejší platformy typu React.js potlačovat [6].

Jak bude ukázáno v kapitole 3, tyto přístupy lze s DotVVM i částečně kombinovat dohromady.

2.3.2 DotVVM Framework

Je relativně nová platforma pro tvorbu webových aplikací od české společnosti RIGANTI s.r.o., uvedená v roce 2016[2]. Je od začátku tvořena jako open-source projekt, který lze zdarma používat i modifikovat. K platformě je pak možné volitelně zakoupit před-připravené komponenty, například pro urychlení vývoje u větších, komerčních projektů. Vývoj je podporován nadací .NET [1].

Je koncipován jako platforma pro ASP.NET, která umožňuje vytvářet stránky pomocí MVVM návrhového vzoru s minimálním úsilím. Samotný MVVM vzor je pak popsán v kapitole 2.4.2. Při tvorbě stránek pak není nutné řešit, jakým způsobem se dostávají data ze serveru ke klientovi, protože tento proces je prakticky programátorovi skryt za standardní View ↔ ViewModel vazbu (binding). Ta se zde vytváří velmi podobným způsobem, jako u platforem WPF, Xamarin a MAUI[3]. Při vytváření stránek se využívá jazyk C# , HTML a CSS [15]. Ve většině případů by pak neměla být nutná ani znalost jazyka JavaScript. Je tak z principu vhodná pro vývojáře zainteresované do ekosystému .NET.

DotVVM Framework umožňuje tvořit webové stránky jako celek, od klientské části po serverovou. Umožňuje tedy full-stack vývoj webové aplikace. Tím se zásadně liší od platforem typu React, Angular a Blazor WebAssembly, které jsou cíleny na tvorbu klientských aplikací [15]. Pokud potřebujeme v těchto aplikacích jakoukoliv komunikaci se serverem, budeme muset vytvořit API, které si budeme muset sami spravovat a které bude muset být vytvořeno pomocí jiné technologie. DotVVM si tyto vazby mezi klientem a serverem tvoří automaticky.

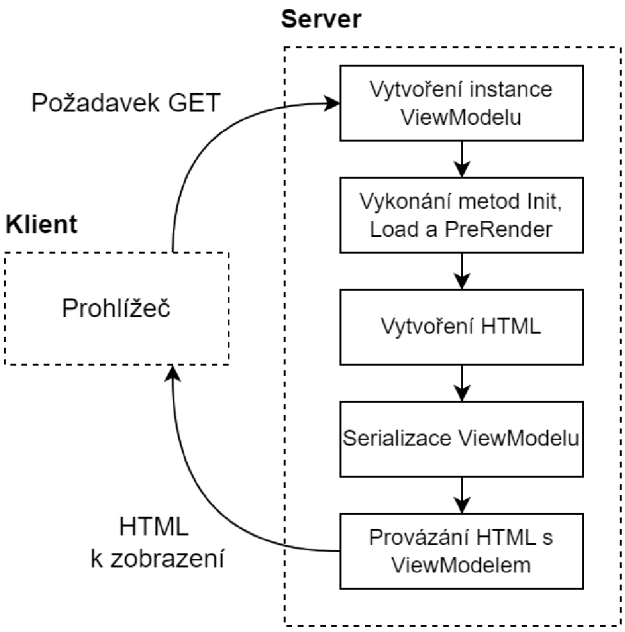
Stránky vygenerované přes DotVVM jsou běžné HTML stránky, které jsou provázané generovaným JavaScriptem. Ten dle potřeby posílá JSON dotazy na REST API, které mu zase vrací JSON objekty. API je pak generované automaticky v ASP.NET. Platforma na klientské straně využívá knihovnu KnockoutJS, která řeší veškerou interaktivitu mezi prvky provázané přes binding a komunikaci s ViewModelem [15] [14].

Na obrázku 2.2 je znázorněn proces představující základ fungování DotVVM. Zobrazování stránky vždy začíná dotazem GET na server. Na tento dotaz reaguje server tak, že vytvoří instanci ViewModel, která náleží k požadované stránce. Následně vykoná všechen kód definovaný ve ViewModel v metodách Init(), Load() a PreRender(). Tento kód musí zajistit naplnění property ve ViewModelu. Následuje vytvoření HTML, do kterého je zakomponován ViewModel, CSS a případně JavaScript moduly. Takto připravené HTML je odesláno zpět ke klientovi, který ho zobrazí ve svém prohlížeči.

Podobným způsobem funguje i vykonávání Commandů (například stisknutí tla-

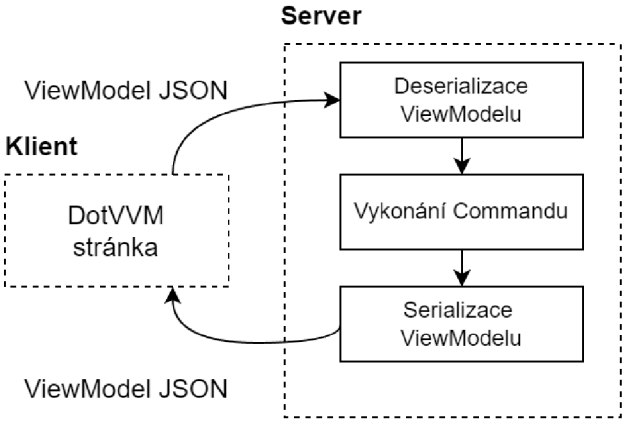
[2]Datum vytvoření verze 1.0 na GitHub

[3]Platformy pro vytváření mobilních a desktopových aplikací od společnosti Microsoft



Obr. 2.2: Životní cyklus ViewModelu v DotVVM při zobrazování stránky [14]

čítka na stránce). Tento proces je znázorněn na obrázku 2.3. V tomto případě klient odešle na server serializovaný ViewModel obsahující všechny aktuální hodnoty, které mohl klient mezitím změnit. Na serveru je pak vytvořena instance třídy představující tento ViewModel a je naplněna daty od klienta. Následně je zavolána metoda uvnitř této třídy patřící ke Commandu. Po dokončení operace je výsledný ViewModel serializován a vrácen zpět klientovi, který pomocí něho aktualizuje stránku.



Obr. 2.3: Životní cyklus ViewModelu v DotVVM při vykonávání Commandu [14]

Oba diagramy byly zjednodušeny tak, aby spíše obecně popisovaly přenos dat v DotVVM. Pro kompletní diagram a popis funkcionality je nutné nahlédnout do [14]

do sekce ViewModels.

2.3.3 ASP.NET WebForms, MVC a Razor Pages

WebForms, MVC a Razor jsou nadstavby nad ASP.NET umožňující vývoj i klientské části webové aplikace. Pro všechny platí, že stránka je vytvářena na straně serveru a odesílána klientovi, podobně jako v DotVVM. WebForms už je nyní považován za překonaný a není ani zahrnut v ASP.NET Core. Na druhou stranu MVC a Razor Pages stále podporovány jsou, ale pro nový vývoj je doporučeno využít Blazor [7].

Ve všech případech platí, že lze webové stránky psát kombinací HTML, *C#* a CSS. Zatímco DotVVM využívá MVVM návrhový vzor, ASP.NET MVC a Razor Pages využívají MVC návrhový vzor. Jejich kód je rozdělen na Model, View a Controller a řídí se konvencemi popsané v 2.4.1. Razor Pages umožňuje tento návrhový vzor obcházet a psát stránky implementující View a Controller v jednom.

2.3.4 Blazor

Společnost Microsoft vytvořila dvě verze této platformy, jako evoluci MVC a Razor - Blazor Server a Blazor WebAssembly. Obě lze považovat za relativně nové technologie, jelikož byly uvedeny v roce 2018. Zatímco verze Server je zamýšlená jako přímá náhrada za MVC a Razor, verze WebAssembly vznikla jako konkurent pro platformy klientských aplikací, jako je například React.js.

Verze postavená na WebAssembly je zaměřena pouze na tvorbu klientské části aplikace. Webové stránky vytvořené na této platformě jsou prakticky .NET programy, které si klient s prvním požadavkem stáhne a spustí je u sebe v prohlížeči pomocí formátu WebAssembly. Jako výhodu lze uvést, že kód stránky lze psát přímo v jednom z jazyků .NET a lze se tak vyhnout JavaScriptu. Nevýhodou je nutnost přenášet velké množství dat ke klientovi, kde jsou zároveň kladeny vyšší nároky na jeho hardware.

Blazor Server je více podobný DotVVM, protože také generuje HTML stránky na serveru a jde také o full-stack řešení. Tyto stránky komunikují se serverem pomocí technologie SignalIR. Ta je z principu náročnější na spolehlivost spojení než klasické REST dotazy, protože vyžaduje nepřerušené připojení k serveru. Na druhou stranu poskytuje možnost vyvolávat aktualizace v klientské aplikaci ze strany serveru [4] [15].

2.3.5 React

Je vysoce populární platforma pro tvorbu klientských aplikací [2] od společnosti Meta (dříve Facebook). Existují dvě verze - React.js a React Native. JS verze je

prakticky JavaScript knihovna a je určena pro tvorbu webových aplikací. Native verze je určena pro tvorbu nativních mobilních aplikací na platformy Android a iOS. Pro obě verze se píše kód ve stejném stylu, takže je teoreticky mezi těmito verzemi přenositelný. Následující popis vychází hlavně z verze React.js.

Práce v React je založena na tvorbě komponent, ze kterých se následně stránky poskládají. Kód se píše ve specializovaném formátu JSX kombinující psaní HTML a JavaScriptu v jednom. Existuje také možnost využívat dříve zmíněný TypeScript a psát kód ve formátu TSX. Díky těmto kombinovaným formátům je možné psát komponenty tak, že lze ihned sledovat jak se postupně skládá struktura naší webové stránky a není před námi skryta žádná logika. Malou nevýhodou tohoto řešení je nutnost před použitím soubory JSX/TSX zkompilovat na čistý JavaScript. K tomu se pak využívá například nástroj Node.js [6].

Jedna ze základních funkcí Reactu je zabudovaná podpora Hooků, které umožňují tvořit vysoce responzivní uživatelské rozhraní, kde každá část stránky se může nezávisle aktualizovat a měnit. Této úrovně responzivity by bylo v čistém JavaScriptu těžké dosáhnout a vyžadovalo by pravděpodobně velké množství kódu. Stejně tak při použití platformy vytvářející stránku na serveru (DotVVM), ve kterých ale zpravidla lze používat komponenty tvořené v JavaScriptu (takže i v Reactu).

React díky své vysoké popularitě také těží z velkého množství veřejně dostupných knihoven a komponent. Díky tomu lze vytvořit funkční stránky během relativně krátkého času. Zároveň je ale nutné kontrolovat původ knihovny - existuje zde spousta knihoven, které vznikly jako projekt jednotlivce a nejsou dlouhodobě udržovány, nebo se jedná o nekvalitní kód. Nutno podotknout, že tento problém se týká i .NET světa, kde ale lze narazit na méně takových knihoven. Pravděpodobně je to ovlivněno strmější učební křivkou C# oproti JS a vyšší celkovou popularitou JS.

Další teoretickou nevýhodou Reactu jsou výrazné změny jeho návrhu v čase a příliš permisivní povaha JS vývoje. Díky tomu lze většinu věcí napsat mnoha různými způsoby, ale ve výsledku jde z hlediska funkcí o stejný kód. Tento zmatek lze pak nejvíce vnímat při učení vývoje aplikací na této platformě. Začínající vývojář může být zmaten, které konstrukce a principy má vlastně používat, protože každý zdroj mu prezentuje zcela odlišné způsoby poplatné času vzniku daného zdroje. Je možné si představit, že bude obtížné v takovýchto projektech udržovat čistý kód. U komerčních projektů tak bude pravděpodobně nutné více dbát na konkrétní vývojové standardy a kontrolu kódu.

2.4 Návrhové vzory

V této sekci budou popsány návrhové vzory MVC a MVVM, jakožto vzory uplatňující se při tvorbě aplikaci, které nějakým způsobem interagují s uživatelem. Dále bude popsán Dependency injection, což je důležitý návrhový vzor, využitý v návrhu nového systému vizualizace. Všechny zmíněné vzory budou popsány hlavně z pohledu technologií Microsoft, jelikož na nich bude nový systém postaven.

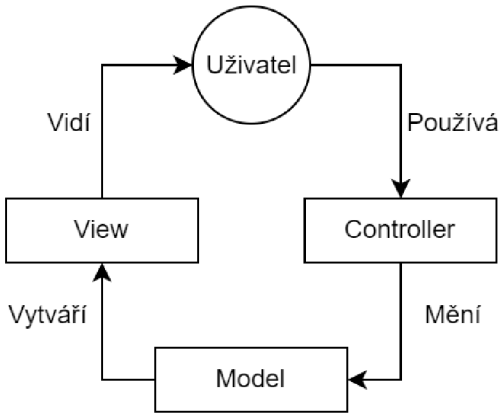
2.4.1 MVC

Původ architektury Model-View-Controller sahá až do začátku 80. let, kde tvořil základ pro první uživatelské rozhraní aplikací [12] a od té doby se používal v mnoho významných platformách. Z těch co jsou dnes stále relevantní můžeme zmínit například Microsoft Foundation Classes (MFC) a ASP.NET MVC.

Tento vzor rozděluje části aplikace s uživatelským rozhraním na Model, View a Controller. Primárním cílem je vytvořit pro uživatele a programátory nějaké zjednodušení, když je nutné pracovat se složitým datovým modelem (Model) [12]. Uživateli jsou prezentována data pomocí View, pokud je potřebuje nějakým způsobem měnit, používá Controller. Každá tato část má svoji určenou zodpovědnost [10]:

- `Model` – zajišťuje veškerou logiku dat v aplikaci, také například načítání a ukládání dat do databáze.
- `View` – popisuje jak bude vypadat uživatelské rozhraní, a prezentuje data. Je vytvářen přímo z Modelu.
- `Controller` – reaguje na požadavky od uživatele a nějakým způsobem upravuje Model. Může také obsahovat logiku pro výběr, jaký Model bude dále použit pro vytvoření View.

Vztah mezi částmi aplikace dle MVC vzoru lze vidět na obrázku 2.4.



Obr. 2.4: Diagram MVC návrhového vzoru

2.4.2 MVVM

Původ této architektury je úzce spjat s ekosystémem .NET, protože byl poprvé použit u technologie WPF a dodnes je hojně používán u platforem spjatých s firmou Microsoft a světem .NET. Jedná se prakticky o evoluci MVC, která se více hodí pro moderní aplikace. Kromě již zmíněného WPF ho lze použít například v Xamarin, MAUI a DotVVM. Jeho dlouhodobá podpora od roku 2005 je důkazem toho, že se osvědčil při tvorbě aplikací [17].

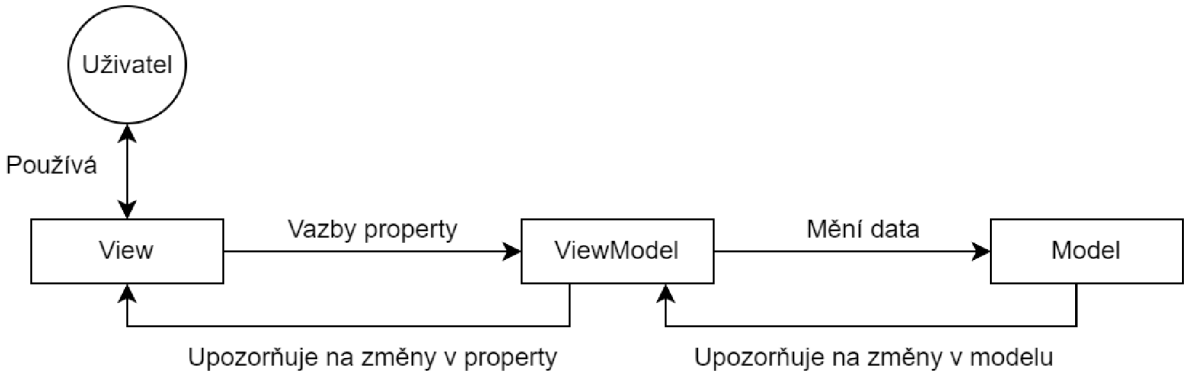
Zkratka doslova znamená Model-View-ViewModel, kdy je podobně jak u MVC struktura aplikace rozdělená na tři části a každá část má přesně určenou zodpovědnost. Zde ale mají jednotlivé části jiný význam [3]:

- `Model` – zajišťuje pouze veškerou logiku dat v aplikaci, také například načítání a ukládání dat do databáze.
- `View` – popisuje jak bude vypadat uživatelské rozhraní, definuje v něm různé uživatelské prvky. Veškeré vazby UI prvků na data nebo události jsou realizovány jako textové odkazy na názvy property a commandů ve ViewModelu. Tyto odkazy se nazývají vazby, neboli binding. View je realizován například pomocí XAML nebo HTML.
- `ViewModel` – spojuje vrstvy View a Model, vystavuje property a commandy, ke kterým se vážou jednotlivé prvky ve View. Obsahuje prakticky veškerou logiku kolem obsluhy UI.

Díky automatickému propojení přes vazby (binding) mezi View a ViewModel vrstvou není nutné psát repetetivní kód pro přiřazování a čtení dat z/do jednotlivých UI prvků. Lze tak například jednoduše používat stejnou hodnotu na více místech.

Data se prezentují/čtou pomocí property ve ViewModelu. Tento způsob umožňuje tvorbu přehledného kódu pro složitější aplikace. Na druhou stranu je složitý pro použití v jednoduchých aplikacích.

Vztah mezi vrstvami MVVM pak možná nejlépe znázorňuje následující diagram na obrázku 2.5. Z tohoto obrázků by mělo být také zřejmé, že mezi vrstvami View a Model není žádná vazba a mohou tak být vyvíjeny zcela nezávisle.



Obr. 2.5: Diagram MVVM návrhového vzoru [3]

Stejně jako u MVC, toto rozvrstvení také teoreticky umožňuje snadnější rozdělení rolí mezi programátory u větších projektů, kdy například vývojáři uživatelského rozhraní nemusí zasahovat do kódu věnující se zpracování dat z databáze. Model-View-ViewModel je pak někdy označován jako Model-View-Binder, například u KnockoutJS [16].

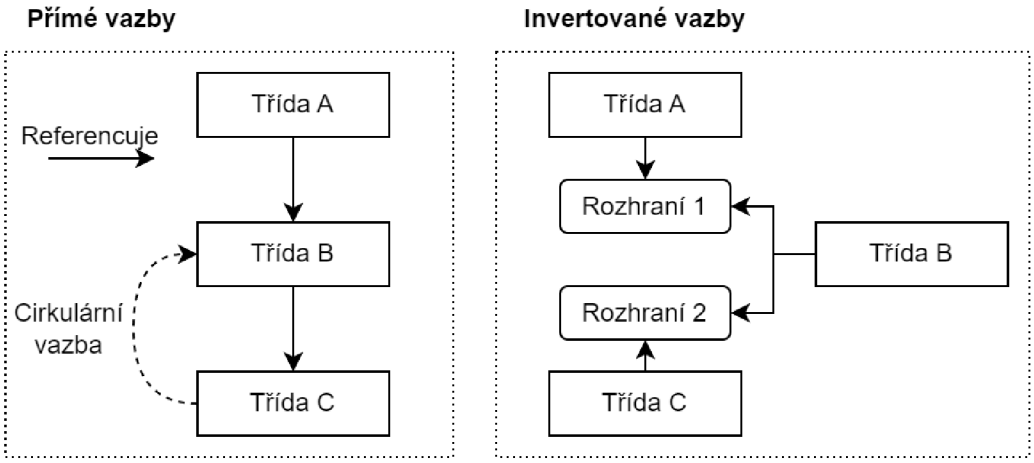
Na závěr k MVC a MVVM návrhovým vzorům je nutné poznamenat, že jde opravdu jen o **vzory**. Jejich význam si jednotlivé platformy a vývojáři vykládají různě, existuje mnoho ne-jednoznačností a drobných rozdílů mezi platformami. Vzory se mohou také volně prolínat a některé platformy mohou umožňovat použití více vzorů zároveň. Slouží tedy spíše jako vodítko, jak bude strukturován vývoj aplikace na konkrétní platformě.

2.4.3 Dependency injection

Na úvod bude vhodné uvést, co označujeme jako vazbu mezi třídami. Jako vazbu lze označit případ, když třída **A** odkazuje na metodu nebo property třídy **B** a spoléhá se na to, že metoda nebo property v třídě **B** existuje. V jazyce C# lze dokonce tvořit tzv. cirkulární vazby, kdy uvnitř metody třídy **B** zároveň voláme zpět metody ze třídy **A**.

Z pohledu .NET je Dependency injection (DI) návrhový vzor pro snadnější správu vazeb mezi třídami. Umožňuje dosáhnout takzvané invertované vazby mezi

třídami, kdy jsou vazby realizovány pomocí společných rozhraní a ne přímo mezi třídami [4]. Tato abstrakce umožňuje tvořit kód jako množinu samostatných komponent, které nemají mezi sebou přímé vazby a lze je tedy snadněji vyvíjet a testovat [18] [11]. Porovnání přímé a invertované vazby lze vidět na obrázku 2.6. Rozhraní zde pak může tvořit C# rozhraní, nebo například nějaká jednodušší třída zprostředkovávající komunikaci.



Obr. 2.6: Diagram přímých a invertovaných vazeb [18]

Dependency injection je v .NET používán tak, že jsou společné rozhraní definovány na jediném místě v aplikaci a následně šířeny do tříd v jejich konstruktorech. Zde můžeme řídit životní cyklus jednotlivých objektů a zaměňovat jejich konkrétní implementace. Tyto objekty pak lze nazývat jako **služby**.

Pokud budeme chtít uvést konkrétní příklad pro ASP.NET Core a DotVVM - služby jsou registrovány při startu aplikace a je definován druh jejich životního cyklu. V rámci požadavku jsou vytvářeny instance ViewModel tříd, které ve svém konstruktoru mohou obsahovat některé z těchto služeb. Systém DI pak automaticky konstruktoru poskytne požadovanou službu, bez ohledu na to, jak hluboko se třída ve struktuře programu nalézá.

Protože jsou tyto služby definovány na jediném místě a zároveň skryté za C# rozhraní, lze je pro účely (jednotkového) testování libovolně zaměňovat. Podmínkou je samozřejmě implementace stejného C# rozhraní.

V ASP.NET Core jsou definovány tyto druhy služeb, které se liší v jejich životním cyklu v rámci HTTP požadavku [13]:

- Transient - každý požadavek na tuto službu vytvoří nový objekt

[4] Nazýváno také jako Inversion of Control (IoC)

- Scoped - požadavek na tuto službu vytvoří nový objekt pouze jednou v rámci HTTP požadavku
- Singleton - existuje pouze jedna instance této služby pro celý běh aplikace

2.5 Designové principy

V této sekci budou zmíněny některé důležité designové principy pro tvorbu dobře spravovatelného kódu, následně tedy i celého softwarového řešení. Následující podsekce vycházejí především z [18] a [5]. Mezi tyto principy lze zařadit i inverzní vazby, které byly popsány již v 2.4.3.

2.5.1 Oddělení zájmů

Za základní princip lze označit Oddělení zájmů (Separation of concerns). Ten říká, že části aplikace by měly být jednoznačně odděleny podle jejich funkce. Je zřejmé, že například kód obsluhující uživatelské rozhraní by měl být oddělen od kódu obsluhující komunikaci s databází. Pro dobrou spravovatelnost by ale měly být i tyto části rozděleny na menší pod-části, které se věnují pouze konkrétní oblasti. Musíme zajistit to, aby každá oblast měla ideálně jen **jednu odpovědnost**.

Princip jedné odpovědnosti (Single responsibility) obecně říká, že objekt musí mít pouze jednu odpovědnost a jeden důvod měnit svůj stav. Svůj stav pak musí měnit pouze v případě, že se změnil způsob, jakým bude realizovat svoji jednu odpovědnost.

Z pohledu obecného návrhu aplikace - uživatelské rozhraní se může skládat z modulárních komponent, kód pro prezentaci dat může být oddělen od kódu zpracovávající vstupy od uživatele. Logika v pozadí aplikace může být tvořena pomocí menších výkonných (business) tříd, které mají méně vazeb a bude možné je používat a testovat samostatně. Místo velkých monolitických tříd vznikne více jednoduchých tříd, každá zajišťující pouze jednu funkci. Pokud se přesuneme hlouběji, každá metoda definována v těchto třídách by měla realizovat pouze jednu funkci.

Tuto myšlenku lze rozšířit i na celé systémy, kdy jednotlivé funkce velkého systému realizujeme pomocí nezávislých mikroslužeb. Pokud potřebujeme v systému další funkce, vytvoříme další službu.

2.5.2 Zabalení

Pro úspěšné oddělení zájmů můžeme využít princip Zabalení (Encapsulation). Z pohledu celé aplikace je dle tohoto principu vhodné oddělovat jednotlivé úrovně tak, aby byly propojeny přes co nejjednodušší rozhraní. Z pohledu konkrétního kódu by třídy neměly poskytovat svému okolí funkce, property a proměnné, které jsou určené

pouze pro jejich interní logiku. Vývojář by měl dbát na to, aby veřejné rozhraní dané třídy prezentovalo pouze věci související s její celkovou veřejně prezentovanou funkcí. V C# je to prakticky realizováno pomocí modifikátorů přístupnosti, využití C# rozhraní a správným nastavením get/set u property.

Výhoda z hlediska spravovatelnosti je zřejmá, protože při prohlížení kódu nejsme zahlcování nadbytečně prezentovanými vlastnostmi prohlížené třídy. Při úpravě kódu takovéto třídy pak můžeme mít jasný přehled, kdy nějakým způsobem ovlivníme veřejný projev (návratové hodnoty, hodnoty v property) dané třídy. Použití rozhraní pak vytváří modulárnější řešení, kdy je možné některé části kódu lehce nahrazovat a testovat.

2.5.3 Neopakování se

Části kódu by se neměly zbytečně opakovat, protože není možné efektivně spravovat všechny jeho výskyty. Opakující se části by měly být odděleny do samostatné jednotky (třídy/metody) tak, aby bylo možné je spravovat na jednom místě. Pokud vyžadují některá místa nějaké specifické chování, je možné využít například dědičnost tříd a factory návrhový vzor. V některých případech může být opakování výhodné, pokud by došlo k výraznému zesložitění vazeb a porušení jiných důležitějších principů.

2.5.4 Persistence ignorance

Tento pojem se používá u aplikací obsahující přístup k trvalému (persistentnímu) úložišti dat. Toto úložiště pak může být nějaký databázový systém, nebo i například soubor XML uložený na disku. Označuje třídy, které by se měly používat uvnitř aplikace pro přenos dat, vyjímaje načítání a ukládání do výše zmíněného úložiště. Tyto třídy by měly být navrženy zcela nezávisle na technologii úložiště a měly by být co nejjednodušší. Díky rozvrstvení na třídy používané pro komunikaci s úložištěm a na třídy používané uvnitř aplikace, nejsme svázáni konkrétní technologií pro ukládání dat. Tyto třídy někdy bývají označovány jako POCO - Plain Old CLR Object.

3 Nový systém pro vizualizaci

V této kapitole bude popsán návrh nového systému pro vizualizaci, jeho architektura a základní principy. V zadání této práce je zmíněno, že první krok by měl být návrh softwarových modulů pro vizualizaci. Nejdříve je ale nutné vytvořit samotnou webovou aplikaci a vytvořit základ pro implementaci těchto softwarových modulů.

3.1 Obecný popis

Nový systém byl vytvořen zcela od začátku, jako zcela nový projekt. Využívá nejnovější dostupnou verzi DotVVM Framework, postavenou již na moderním ASP.NET Core jako svůj základ.

Základní myšlenky, na které byl kladen důraz v celém návrhu a realizaci, je modularita a dobrá spravovatelnost kódu. Budeme se tedy snažit se držet designových principů, které byly popsány v kapitole 2.5. Aplikace bude tvořena z více malých částí tak, aby každá část měla pouze jednu zodpovědnost, třídy budou co nejvíce zabalené a přístup k datům bude rozvrstvený. Zároveň se budeme snažit, aby názvy tříd a rozhraní byly sebe-vysvětlující a vznikal tak sebe-dokumentující se kód. Pokud se povede tyto cíle udržet napříč celým projektem, měl by být do budoucna vysoce spravovatelný [5].

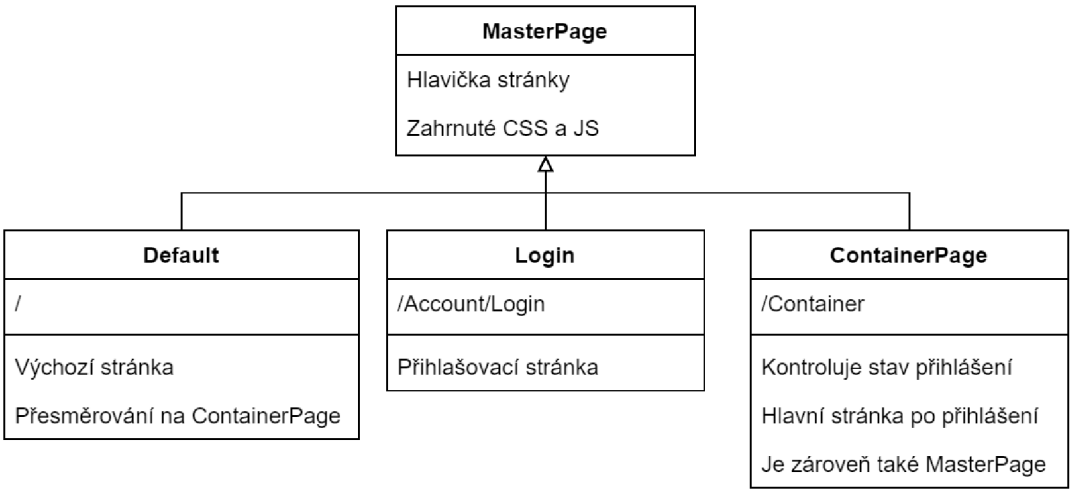
Vizualizace bude používat jako úložiště dat MySQL databázi. Načítání datových bodů pro vizualizaci bude používat stejný datový model, který využívá i starý systém vizualizace. Nebude ale využívat stejné tabulky pro ukládání nastavení systému. Vzniknou nové tabulky podle architektury nového systému. Problematika načítání dat je velice klíčová a je jí věnována sekce 3.3.1.

Z hlediska uživatelského rozhraní byl jako základní vzhled aplikace implementován balíček Bootstrap 5 Admin Dashboard z webu Bootstrapious. Byla použita pouze volně přístupná verze zdarma. Tento balíček je soubor CSS stylů, které nám umožňují rychle stylovat HTML elementy a vytvářet tak uživatelsky přívětivou stránku. Zároveň byla použita nadstavba Bootstrap for DotVVM obsahující množinu pokročilejších komponent k použití v naší webové aplikaci. V aplikaci se dále vyskytuje mnoho ikon a jako jejich zdroj byla zvolena knihovna Material Icons od Google. Všechny soubory knihoven a ikon byly staženy jako soubor pro offline použití. Nebude tedy nutné, aby se web dotazoval na externí servery pro správné zobrazení. Použití těchto knihoven by nám teoreticky mělo umožnit se více soustředit na samotnou logiku v aplikaci, místo jejího vzhledu.

3.2 Struktura webové aplikace

Doporučená struktura v DotVVM je následující. Nejdříve se definuje MasterPage, tvořící základ všech zobrazovaných stránek. Ten uvnitř sebe zobrazuje další obsah (Content) dle cesty v navigaci (URL). MasterPage pak zároveň definuje všechny prvky HTML, které budou mezi těmito obsahy sdíleny. To může být například navigační menu, nebo definice použitého CSS stylu a JavaScriptu. Logika MasterPage a Content je pak stejná i u ViewModelů, kdy třídy jednotlivých Contentů dědí z třídy ViewModelu svého MasterPage. Je tedy možné i v této části implementovat různou společnou logiku.

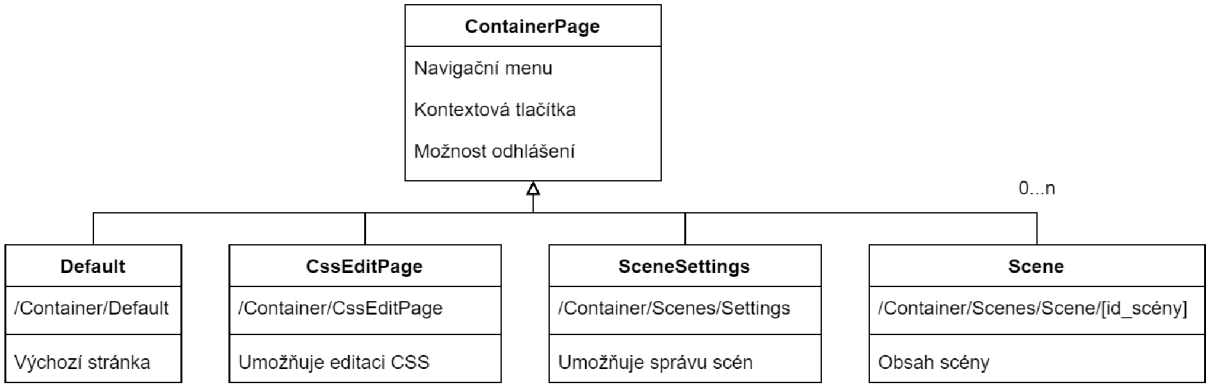
V navigaci systému pro vizualizaci byla zvolena následující struktura navigace zobrazená na obrázku 3.1.



Obr. 3.1: Diagram struktury základu stránky

V diagramu je možné vidět, že základní struktura umožňuje pouze navigaci mezi přihlašovací stránkou a ContainerPage, což je další MasterPage. Ten obsahuje další obsah týkající se vizualizace. Tato struktura umožní jednoduše implementovat do webu vizualizace stránky, které nepotřebují zobrazovat prvky v ContainerPage.

Pokud se přesuneme o úroveň níže, do již zmíněného ContainerPage, můžeme vidět následující strukturu.



Obr. 3.2: Diagram struktury hlavní stránky vizualizace

Zde si můžeme povšimnout, že stránek s obsahem scén může být 0 až n. Tyto stránky jsou generovány podle definovaných scén v databázi.

3.2.1 MasterPage

Obsahuje HTML hlavičku a definice zahrnutých CSS a JavaScript souborů. Tato část je pak společná pro všechny další stránky. Je vhodné zde také zmínit stránku Default, která působí jen jako zástupný prvek při prvním otevření vizualizace. Zde lze zvolit, na jakou stránku bude uživatel přesměrován z výchozí URL. V tomto případě dojde vždy k přesměrování na stránku ContainerPage.

3.2.2 Přihlašovací stránka

Přihlašovací stránka je zcela základní, jakou lze vidět u mnoha jiných webových stránek. Je vyžádáno pouze jméno a heslo od uživatele. Při úspěšném přihlášení je uživatel přesměrován na ContainerPage, odkud se může navigovat na stránky scén. U systému je použito přihlašování na základě cookies, takže je možné v prohlížeči stránku různě zavírat/otevírat bez nutnosti dalšího přihlašování. Správu uživatelů zajišťuje externí systém, takže není nutné zde řešit problematiku registrací a zapomenutých hesel. Na obrázku 3.3 je vidět příklad zprávy při neplatných přihlašovacích údajích.

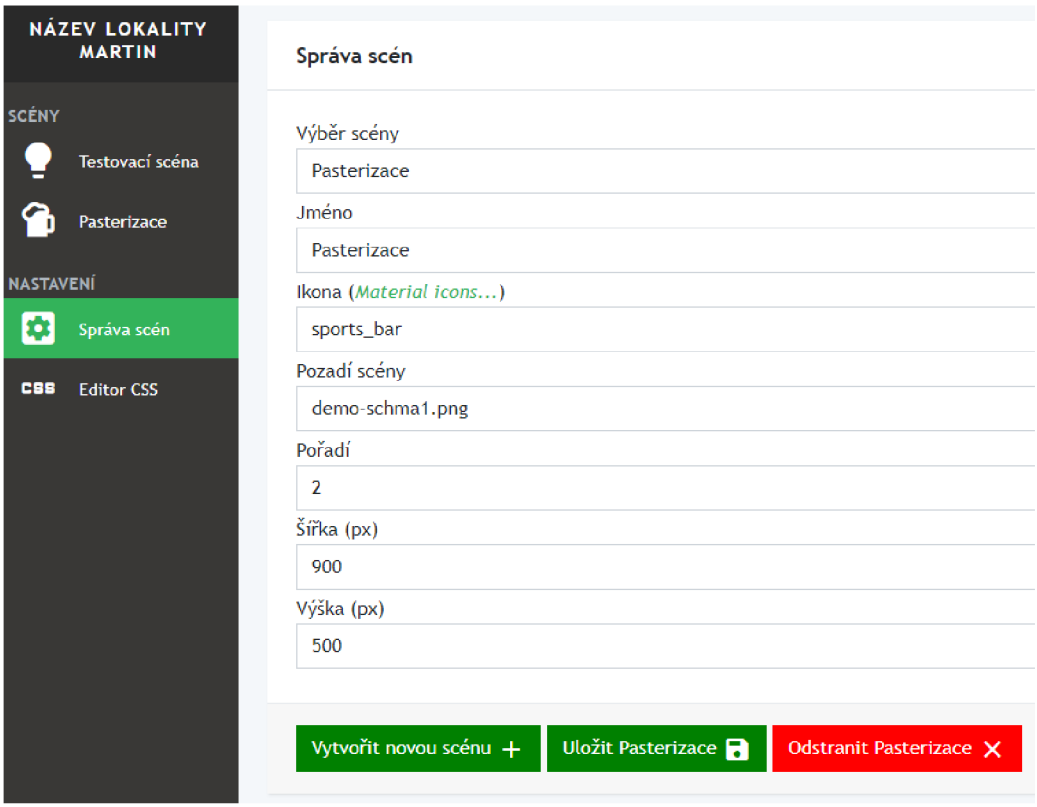
Obr. 3.3: Ukázka přihlašovací obrazovky v nové vizualizaci

3.2.3 ContainerPage

Je také stránka typu MasterPage a obsahuje další Content týkající se vizualizace. Obsahuje prvky navigace společné pro všechny obsažené stránky, jako navigační menu, kontextuální tlačítka a tlačítko pro odhlášení. Umožňuje přepínání mezi scénami, otevření správy scén a otevření editoru CSS. Stránka je dělaná responzivně, takže navigační prvky by měly být dostupné i na mobilních zařízeních.

3.2.4 Správa scén

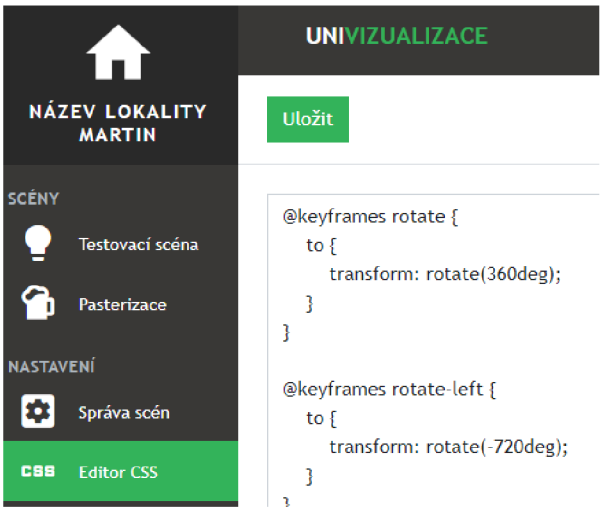
Na této stránce je možné vytvářet, mazat a editovat scény. Při vytváření je jediné kritérium, aby název scény nebyl prázdný. Při mazání je uživateli zobrazen dialog, ve kterém musí explicitně zadat název mazané scény pro potvrzení smazání. U každé scény lze pak definovat, jakou bude mít ikonu v menu a v jakém pořadí se bude v menu zobrazovat. Lze také nastavit obrázek, který bude zobrazován jako pozadí scény. Tento obrázek se musí nacházet ve vyhrazeném adresáři webu backgrounds. Ukázka správy scény je na obrázku 3.4.



Obr. 3.4: Ukázka správy scén v nové vizualizaci

3.2.5 Editor CSS

Tato stránka umožňuje za běhu aplikace měnit globální CSS styly. Tyto styly pak mohou být použity uvnitř modulů na scéně. Soubor editovaný uživatelem je zcela oddělen od CSS důležitých k fungování aplikace, takže by jeho editace za běhu aplikace měla být bezpečná. Jak lze vidět na obrázku 3.5, jde o jednoduchý textový editor. Po stisku tlačítka Uložit dojde k uložení na server a znovu-načtení stylů prohlížečem.



Obr. 3.5: Ukázka editoru CSS

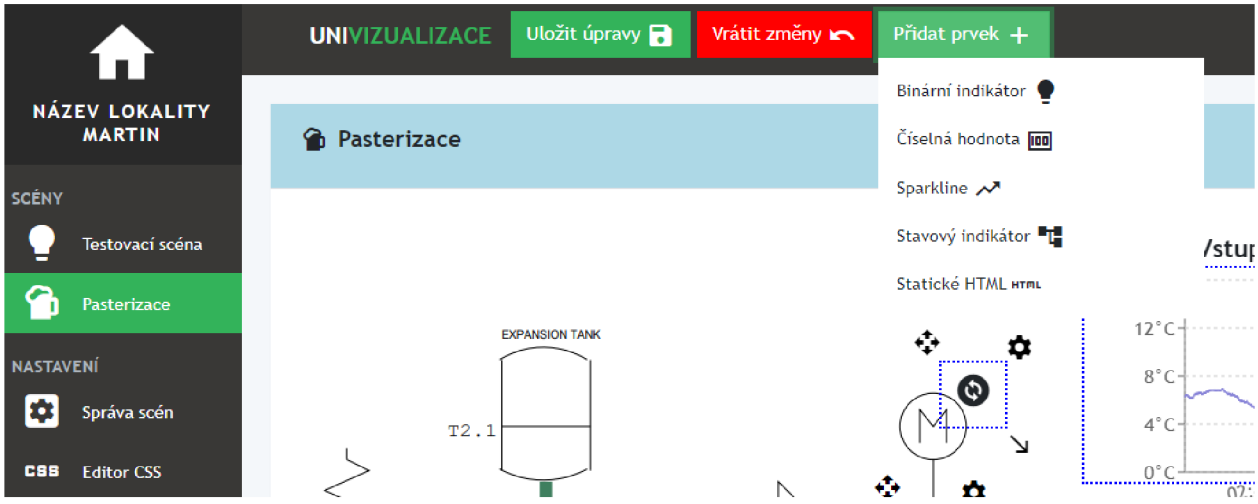
3.2.6 Stránka scény

Tyto stránky jsou jádro systému pro vizualizaci. Každá scéna představuje jednotlivý sledovaný proces, nebo jeho část. Pro zajištění univerzálnosti zde byl zaveden systém dlaždic, představující instance jednotlivých softwarových modulů. Popis dlaždic je v samostatné kapitole 3.2.7. Při zobrazení scény se nabízejí kontextová tlačítka na vrchní straně stránky. Přes ty je možné přepnout scénu do editačního režimu, uložit změny, případně změny vrátit do původního stavu. V editačním režimu je zobrazeno tlačítko pro přidání dlaždice.

Scéna má vždy definovanou minimální šířku a výšku. Pokud definujeme jako pozadí scény obrázek s většími rozměry, přizpůsobí se velikost scény obrázku. Pro lepší ergonomii je možné scénu myší chytnout a posouvat ji v rámci stránky. Pokud kdekoliv scéna přesahuje okraje stránky, zobrazí se scrollbar.

Stránka zajišťuje přesouvání a změnu velikosti dlaždic. Tyto akce se zachytávají do mřížky, jejíž rozlišení můžeme definovat od 1 do 25 pixelů. K zajištění těchto funkcí byl vyvinut JavaScriptový modul, který manipuluje přímo s ViewModelem na straně klienta a mění jeho hodnoty.[1]. Tento modul sleduje pozici kurzoru na scéně a počítá nové hodnoty pro dlaždice a scénu i v závislosti na pozici scrollování scény. Změnu velikosti a pozice dlaždice lze ukončit také stiskem tlačítek Esc a Enter na klávesnici.

[1]Manipulace dat ViewModelu na klientovi je nativně podporovaná funkce DotVVM [14].



Obr. 3.6: Ukázka stránky scény v režimu editace

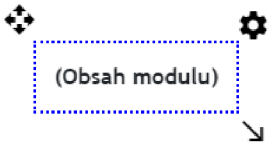
3.2.7 Dlaždice

Dlaždice jsou komponenty DotVVM, které je možné rozmisťovat na scéně a jejichž obsah tvoří jednotlivé softwarové moduly. Jejich hlavní funkce je tvorba společného rozhraní mezi scénou a moduly, zároveň však mají několik vlastních funkcí:

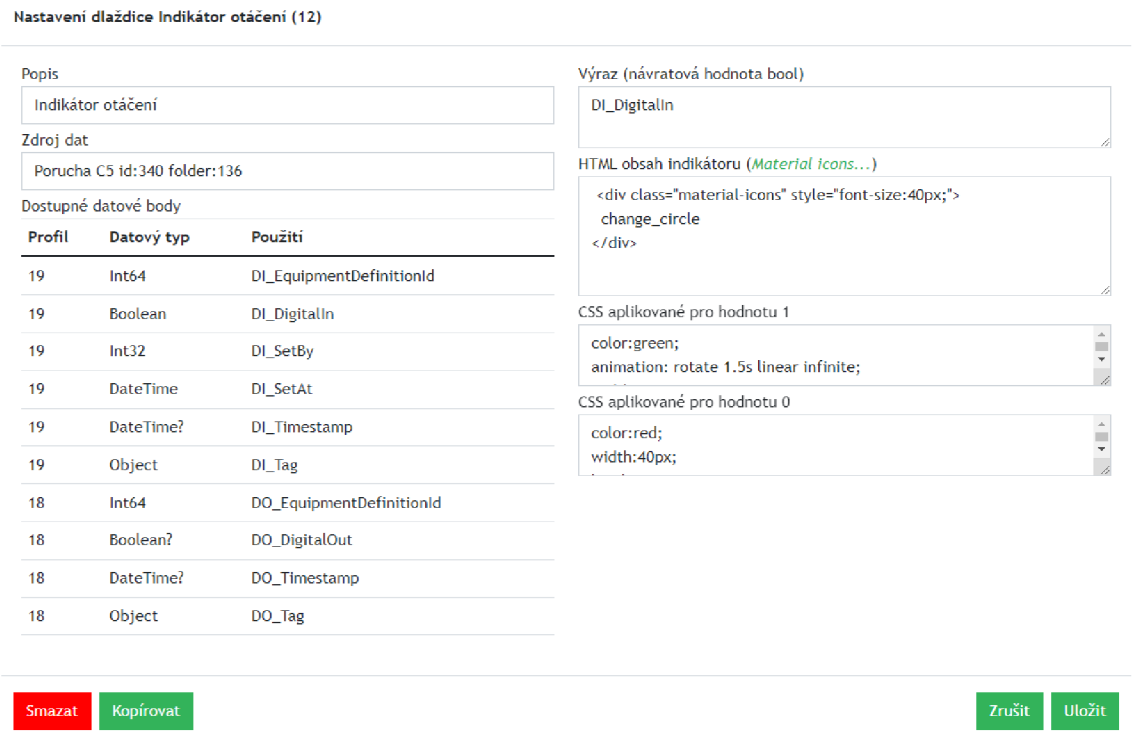
- Změna pozice a rozměrů na scéně
- Společné uživatelské rozhraní bez ohledu na obsah dlaždice - okraj zobrazovaný v editačním režimu, tlačítka pro změnu pozice a rozměrů, tlačítko nastavení
- Zpracovávání záznamu dlaždice z databáze
- Zobrazování uživatelského rozhraní specifického pro konkrétní modul
- Zobrazování chyby v modulu
- Vytvoření ViewModelu pro konkrétní modul
- Kopírování a mazání dlaždic na scéně

Dlaždici je možné vidět na obrázku 3.7. Pomocí šipek vlevo nahoře lze dlaždici přemisťovat, pomocí šipky vpravo dole lze měnit její velikost. Uvnitř ohraničení je pak obsah pocházející z použitého modulu. Pomocí tlačítka s ozubeným kolem lze otevřít nastavení dlaždice, které lze vidět na obrázku 3.8. Toto nastavení se otevře jako modální okno rozdělené na společnou část (vlevo) a část specifickou pro daný modul (vpravo). Přes dialog nastavení lze také dlaždici zkopírovat nebo smazat.

Díky tomuto rozvrstvení je možné tvrdit, že scéna nemá vazbu na softwarové moduly a její implementace je na modulech zcela nezávislá. Zabývá se pouze správou dlaždic, které jsou vždy stejné. Stejně tak dlaždice mají minimální vazbu na samotné



Obr. 3.7: Ukázka dlaždice



Obr. 3.8: Ukázka nastavení dlaždice

moduly a pouze se zabývají vytvářením jejich instancí. Veškeré další vazby jsou zajištěny pomocí pevně definovaného rozhraní. To bude konkrétněji vysvětleno v sekci 3.5.1.

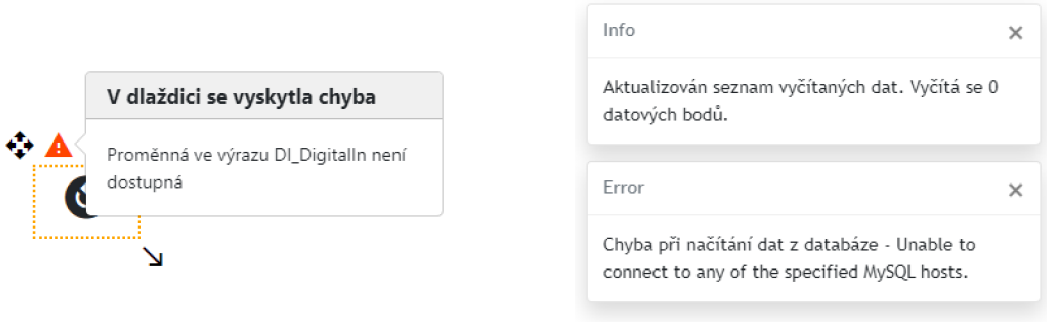
3.2.8 Notifikace o chybách

Během běžného fungování vizualizace lze očekávat, že budou vznikat různé chyby. Ty mohou pocházet například z načítání a zobrazovaní dat v softwarových modulech, nebo při nastavování dlaždice. Proto byl vytvořen jednotný systém pro logování chyb a jejich zobrazování v aplikaci.

Sbírání a zobrazování globálních chyb, které se nevážou pouze k jedné dlaždici, je zajišťováno pomocí služby označené jako agregátor chyb v 3.4.2. Službu lze na

libovolném místě zavolat a předat do ní položku Error (chybu) nebo Info (notifikaci) s unikátním klíčem. Tento klíč je krátký textový řetězec a zajišťuje, aby se stejná chyba zobrazovala pouze jednou. Globální chyby se zobrazují jako vyskakovací okénka na pravé straně stránky scény, které se skládají pod sebe podle času přidání položky. Lze je ručně zavřít, nebo se po 3 sekundách zavřou automaticky. Příklad těchto notifikací lze vidět na pravé straně obrázku 3.9.

Pokud se chyba týká pouze konkrétní dlaždice/modulu, je u komponenty na scéně zobrazena ikona varovného trojúhelníku. Po najetí nad ikonu se zobrazí malé překrytí s textem chyby. Logika přidávání chyb je stejná jak u globálních chyb, jen v tomhle případě musí modul ručně nastavit, že tato konkrétní chyba je již vyřešena. Chybu odstraní znovu pomocí textového klíče. Příklad interní chyby dlaždice lze vidět na obrázku 3.9 na levé straně.



Obr. 3.9: Ukázka zobrazování globálních chyb a interních chyb dlaždice

3.3 Načítání a zpracování dat

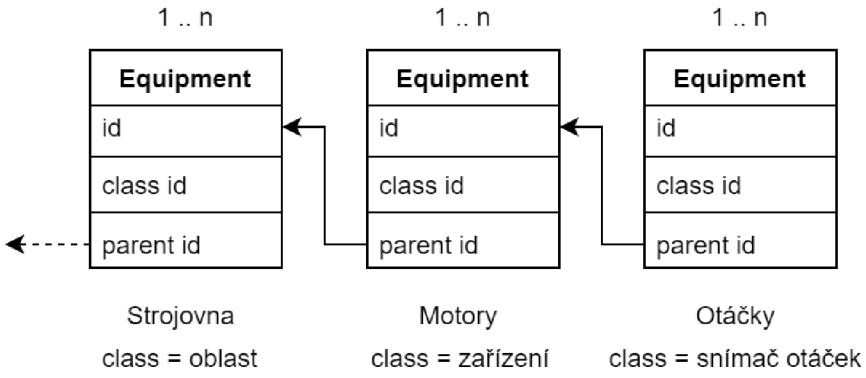
V této sekci bude popsán datový model použitý jako zdroj dat pro prvky ve vizualizaci. Bude vysvětleno jeho celkové schéma a problematika univerzálního načítání dat z tabulky s libovolnou strukturou. Následně bude představeno řešení pro načítání těchto dat a možnost flexibilního použití v softwarových modulech.

3.3.1 Datový model

Vizualizace může plnit svůj účel pouze tehdy, když nám bude zobrazovat stav (data) nějakého sledovaného systému. Pro získání těchto dat využívá nový systém stejný datový model jako jeho předchůdce a je také napojen na MySQL databázi. Data do této databáze poskytuje externí systém, který je získává od sledovaných zařízení, například z PLC rozmístěných napříč výrobním procesem. Stejně tak konfiguraci

datových bodů a jejich struktury provádí externí systém. Vizualizace se bude na tyto data pouze napojovat a zobrazovat je.

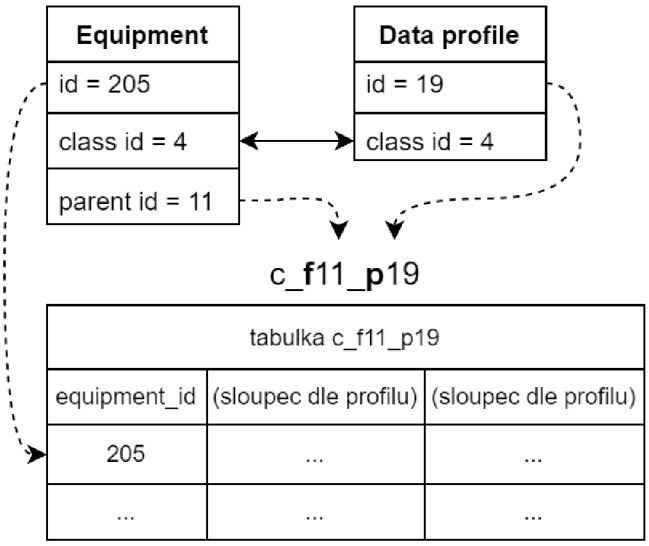
Abychom mohli popsat načítání dat, je nutné nejdříve popsat strukturu datového modelu. Model je tvořen vysoce flexibilním způsobem, který je ale díky tomu více komplikovaný. Data se vyskytují v rozdílných tabulkách, definovaných pomocí stromové struktury. Ta může mít teoreticky téměř nekonečnou hloubku. Na obrázku 3.10 je ukázka této struktury pro snímač otáček motoru. Databázové entity **Equipment** jsou vedle sebe umístěny v jedné tabulce a navzájem na sebe odkazují pomocí parent_id ve vztahu 1 až n. Tím vytváří pomyslnou stromovou strukturu, protože každý element může mít nekonečno podřízených elementů. Equipment má vždy přiřazený class_id, čímž je definováno o jaký **typ objektu** se jedná. Může to být například obecná složka, skupina zařízení, instance zařízení až po konkrétní snímač.



Obr. 3.10: Entity typu Equipment

Pro každý typ objektu (class_id) může být definováno téměř nekonečno **datových profilů**. Ty nám popisují, jaká data jsou dostupná pro daný typ objektu. Každý datový profil definuje struktury tabulky, která bude pro daný typ zařízení existovat. Pokud tuto strukturu projdeme, získáme názvy tabulek, kde se budou vyskytovat data daných Equipmentů. Získání názvu zdrojové tabulky je popsáno na obrázku 3.11.

Tyto data jsou tím co chceme primárně zobrazovat ve vizualizaci. Hodnoty v tabulce s předponou **c__** (current) jsou pouze aktuální hodnoty dané entity Equipment a nelze zde dohledat historické hodnoty. Pro práci s historickými daty existují tabulky s předponou **h__** (history), kde jsou ukládány hodnoty daného zařízení v čase. Tyto tabulky mají stejnou strukturu dle daného profilu, ale jsou vyhrazeny pouze pro jeden Equipment. Pro zařízení s equipment_id 205 z obrázku 3.11 by se taková tabulka jmenovala h__e205__b0000__p18. Místo povinného sloupce equipment_id jsou zde sloupce id a timestamp, kdy timestamp je sloupec typu DateTime



Obr. 3.11: Přístup k datům na základě profilu

a obsahuje čas navzorkování daného řádku tabulky.

3.3.2 Napojení aplikace na datový model

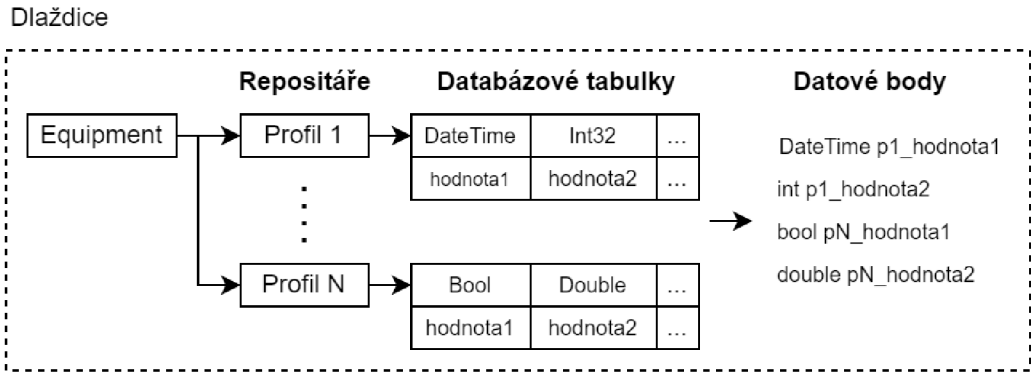
V zájmu zachování kompatibility se starší verzí vizualizace bylo rozhodnuto, že bude použitý stejný způsob přístupu k databázi. Jedná se o na míru vytvořené třídy zprostředkovávající načítání a ukládání dat do databáze s využitím takzvaného Repository návrhového vzoru. Umožňují specifikovat kritéria, která jsou ekvivalentní k podmínkám při dotazu v MySQL. Tato kritéria pak lze uplatňovat při načítání a mazání dat. Dále umožňuje nastavovat řazení a stránkování.

Třídy repositáře nám vrací přímo naplněné POCO objekty se strukturou odpovídající čtené tabulce. Pro zápis do tabulky se vkládají stejné objekty. Příklad takového objektu lze vidět na výpisku 3.2. Repositáře zároveň umí automaticky načítat a mazat objekty, které jsou podřízené danému objektu (například Scéně jsou podřízené objekty Dlaždice). Toto chování lze konfigurovat, nebo vypnout úplně. Pro každý datový profil je definována jedna třída repositáře.

Tyto třídy jsou generovány pomocí externího programu, kde jsou definovány vazby mezi tabulkami. Poté jsou ručně vloženy do kódu. Lze tedy očekávat, že s úpravami v tabulkách budou tyto části kódu přepisovány jako celek. Zároveň je nutné čtení dat přes třídy repositářů udělat univerzálně, aby bylo možné použít libovolný datový model.

V novém systému proto vzniklo následující řešení. Bylo rozhodnuto, že každá instance dlaždice vizualizace bude napojená na jeden Equipment. Ten může mít

přiřazený libovolný počet datových profilů, kdy každý profil odpovídá typu databázové tabulky (a repositáře). Každý sloupec této tabulky je **datový bod**, který chceme zobrazovat ve vizualizaci. Tento koncept je zobrazen na obrázku 3.12. Do názvu datových bodů je vložen i název datového profilu, díky tomu je napříč aplikací unikátní.



Obr. 3.12: Datové body jako zdroj dat dlaždice

Při prvním dotazu na dostupné datové body je pomocí reflexe[2] prohledán kód projektu systému vizualizace a jsou hledány všechny definované repositáře a typy objektů, které využívají. Property těchto objektů pak odpovídají sloupcům tabulek a jsou to tedy námi hledané datové body. Následně je provedena filtrace a jsou zachovány pouze datové body profilů, jejichž tabulky reálně v databázi existují. Tento proces je proveden pouze jednou za běh aplikace.

Všechny dotazy na data uvnitř vizualizace jsou prováděny s parametry id Equipmentu a názvu datového bodu. Takový požadavek nám jednoznačně definuje, jakou třídu repositáře je nutné použít a jakou konkrétní tabulku v databázi používáme.

Pro potřeby automatického načítání tříd repositářů a jejich datových bodů bylo nutné udělat drobné úpravy v kódu těchto tříd. Rozšíření lze vidět ve výpisku 3.1. Repositáře musí implementovat rozhraní IRepository, který vrací načítaná data jako generické objekty. Použitím bázové třídy BaseAutoRep<T, K> je toto rozhraní již implementováno. Zároveň musí obsahovat konstanty ProfileId a Description. ProfileId je v každé třídě již definované, ale nebylo přístupné jako veřejná konstanta. Description je textová hodnota přidávaná do názvu datového bodu. Tento text by měl ideálně popisovat přímo význam datového profilu, například digitální vstupy mohou mít předponu DI. V poslední řadě musí obsahovat konstantu Category rozlišující repositáře pro aktuální a historická data.

[2]Nástroj pro prohledávání a manipulaci s .NET kódem za běhu aplikace

Výpis 3.1: Příklad rozšíření třídy repositáře

```
public class HistorySensorAORep : BaseAutoRep<HistorySensorAOItm,
    HistorySensorAOItm>
{
    //id profilu
    public const int ProfileId = 14;

    //popis ve výrazech
    public const string Description = "HistoryAO";

    //kategorie aktuální/historická data
    public const ValueSourceCategory Category =
        ValueSourceCategory.History;

    ...
    //ostatní automaticky generované části
}
```

Objekty vrácené z datových repositářů **aktuálních** dat musí implementovat rozhraní IEquipmentDataPoint, definující property EquipmentDefinitionId. Tato property se bude u tabulek pro aktuální data vždy vyskytovat a jde tedy jen o malou úpravu.

Výpis 3.2: Příklad rozšíření třídy vracející se z repositáře

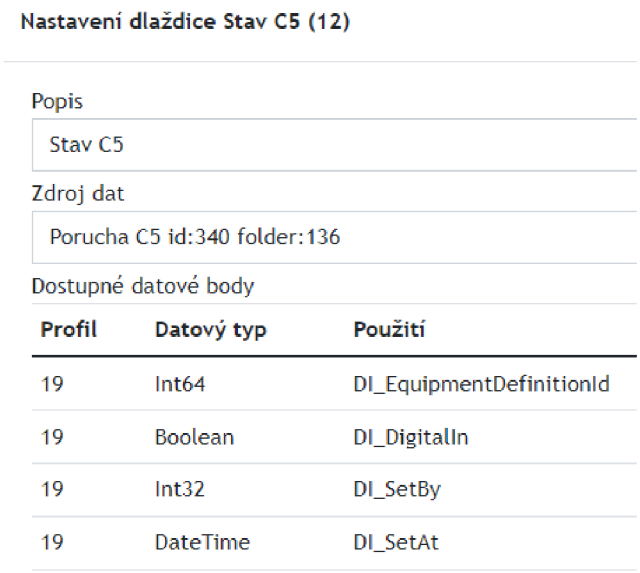
```
public class EngineAOItm : BaseItem, IEquipmentDataPoint
{
  public long EquipmentDefinitionId { get; set; }
  public float AnalogOut { get; set; }
  public DateTime? Timestamp { get; set; }

  ...
  //ostatní automaticky generované části
}
```

Všechny změny udržují tyto třídy zpětně kompatibilní. V budoucnu při nasazení této aplikace by pravděpodobně byly přidány do externího programu pro jejich generování.

3.3.3 Využití datových bodů

V nastavení dlaždice nyní můžeme vybrat zdroj dat (Equipment) a následně se nám jeho dostupná data vypíšou jako seznam datových bodů a jejich C# datových typů. Příklad výběru zařízení a jeho datových bodů lze vidět na obrázcích 3.13 a 3.8. Tato část nastavení je společná pro všechny dlaždice.

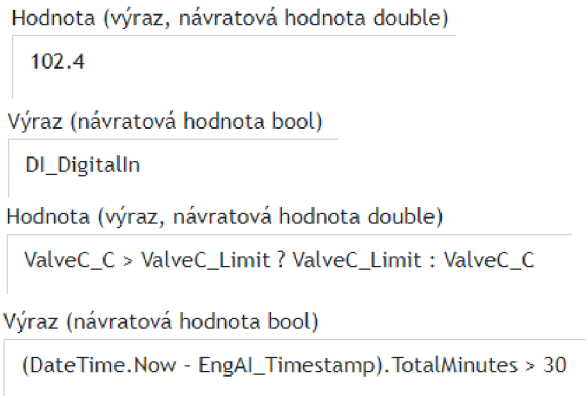


Obr. 3.13: Výběr datových bodů

Tyto datové body jsou používány dvojím způsobem, v závislosti na tom, jestli datový bod obsahuje aktuální nebo historickou hodnotu. Modul v dlaždici předává informaci, jaký typ hodnot je schopen zpracovávat a podle toho se dostupné datové body filtrují. Nejprve si popíšeme zpracovávání aktuálních hodnot, používaných hlavně v rámci výrazů.

Pro zobrazování hodnot a vyhodnocování stavů uvnitř softwarových modulů byl vytvořen systém **výrazů**. Tyto výrazy jsou jednořádkový C# kód, ve kterém lze používat datové body jako jednotlivé proměnné. Návratová hodnota je výsledek tohoto výrazu. Lze tak v rámci možností C# s datovými body provádět libovolné výpočty a transformace - vkládání podmínek, bitové operace a podobné. Pokud se podíváme na obrázek 3.14, můžeme používat například následující výrazy:

- `1. příklad` - konstantní hodnota
- `2. příklad` - přímé zobrazení hodnoty z databáze (datového bodu)
- `3. příklad` - podmínka omezení hodnoty do určitého limitu
- `4. příklad` - test na uběhlou dobu od posledního navzorkování hodnoty v databázi



Obr. 3.14: Příklady využití výrazů

Každý softwarový modul může využívat libovolné množství výrazů a je až na vývojáři modulu, jak s hodnotami bude nakládat. Správa a vykonávání výrazů je řešena centrálně přes společné rozhraní dlaždic, takže jejich použití je pro vývojáře modulu velice jednoduché. Vše kolem výrazů je zajišťováno službami, které jsou popsány v 3.15. Samotné vyhodnocování výrazů zajišťuje externí knihovna DynamicExpresso.

V softwarovém modulu se lze odkazovat i přímo na aktuální hodnotu datového bodu a použití výrazů je pouze jednou z možností jak tento systém používat.

Pokud je potřeba v dlaždici používat historické hodnoty, není možné je používat ve výrazech, ale je nutné se na ně odkazovat přímo. Historické datové body jsou pak načítány zcela odděleně od aktuálních, pomocí jiné služby. Důvodem oddělení je fakt, že každá načtená hodnota by generovala vlastní výraz a zahltila by systém vyhodnocování výrazů. Přepočet pomocí výrazu by v těchto dlaždicích měl být možný, ale je potřeba ho dělat zvlášť jen v rámci daného modulu.

3.3.4 Aktualizace prvků na scéně

Vizualizace musí zobrazovat data sledovaného systému v reálném čase, takže musí docházet k periodickému načítání dat a aktualizaci uživatelského rozhraní. Periodická aktualizace je iniciovaná ze strany klienta (prohlížeče) pomocí časovače v JS. V rámci každé periody obnovy je realizována následující sekvence:

- JS časovač iniciuje aktualizaci stránky Scény.
- Ze všech dlaždic/softwarových modulů jsou sesbírány použité výrazy a použité datové body.

- Datové body jsou seskupeny takovým způsobem, aby každý objekt byl čten z databáze pouze jednou.
- Jsou načtena data z databáze do datových bodů (pouze aktuální hodnoty).
- Pro každou dlaždici na scéně je volána metoda Refresh, ve které modul začne vykonávat výrazy pomocí centralizované služby. Pomocí výsledků výrazů aktualizuje svůj vzhled.

Načítání dat do datových bodů probíhá znovu s použitím reflexe, kdy jsou postupně vytvářeny instance repositářů a dle požadavků na datové body jsou vyčítány tabulky z databáze. Následně jsou s pomocí reflexe přenášeny hodnoty z objektů repositáře do hodnoty datového bodu, podle jeho definovaného jména.

Celé řešení používá extensivně reflexi při každé aktualizaci, takže lze očekávat, že bude výpočetně náročnější. Aby byl zmírněn dopad na výkon, bylo implementováno již zmíněné seskupování požadavků na datové body. Služba pro vykonávání výrazů pak dokáže cachovat totožné výrazy a výrazy, jejichž hodnota se nemohla mezi aktualizacemi změnit, protože hodnoty jejich vstupních proměnných se také nezměnily. Míra použití reflexe je tak snížena na nutné minimum pro zajištění této funkce.

U modulů využívajících historická data nedochází k žádnému seskupování požadavků, ale data jsou vyčítána až dle potřeby v posledním kroku aktualizace. Načítání je řízeno vyhrazenou službou a dochází zde ke cachování historických dat. Nová data z databáze jsou načítána vždy až po uplynutí definované periody obnovy.

U obou služeb obsahující cachované hodnoty dochází jednou za čas k úklidu, kdy jsou již nepoužívané záznamy mazány. Kontrola probíhá jednou za hodinu a mažou se záznamy starší než dvě hodiny.

3.4 Přenos dat uvnitř systému

V této sekci bude vysvětleno, jakým způsobem se předávají závislosti a data mezi třídami uvnitř systému. Zároveň bude představeno řešení, pomocí kterého je možné některá data přechovávat v paměti serveru v rámci přihlášení uživatele.

3.4.1 Služby

Většina dat v systému je poskytována pomocí služeb. Instance těchto služeb a jejich závislosti jsou pak vytvářeny pomocí Dependency injection. To částečně zjednodušuje návrh, protože nemusíme věnovat téměř žádné úsilí předávání závislostí a dat mezi třídami. Popis tohoto návrhového vzoru a jeho výhody byly již popsány v 2.4.3.

V aplikaci existují dvě úrovně služeb, zajištěné dvěma různými nástroji pro realizaci DI. První úroveň lze označit jako **globální služby**. Tyto služby jsou registrovány při startu aplikace a jsou zajišťovány přímo pomocí ASP.NET Core. Při vytváření třídy ViewModelu jsou automaticky platformou vkládány do konstruktoru, odkud je lze dále používat. Jedna z globálních služeb pak tvoří adaptér pro získání LoginScoped služeb. Ty budou detailně popsány v 3.4.2. Všechny služby existují vždy pouze na straně serveru a v rámci svého definovaného životního cyklu. Mezi globální služby patří:

- Repositář databáze pro konfigurační data vizualizace
- Zpracování příkazů z dlaždic
- Poskytovatel LoginScoped služeb

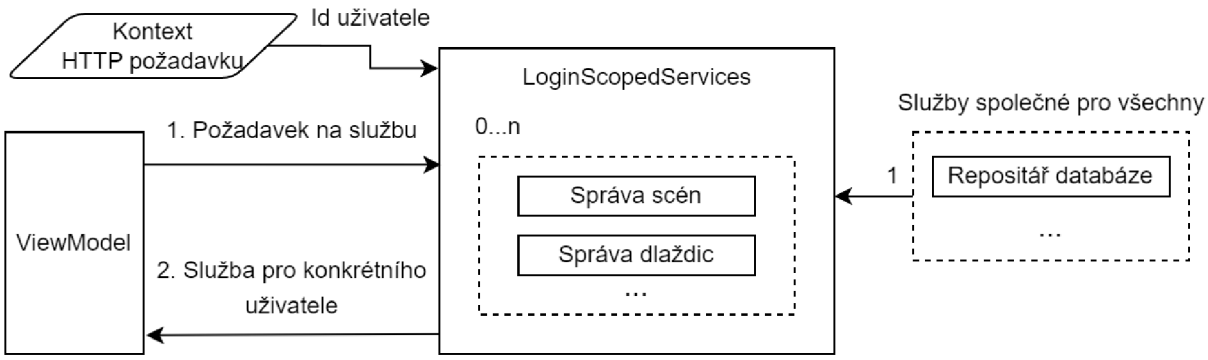
Celá struktura šíření závislostí/dat je vrstvená a každá služba má pouze svoji jednu odpovědnost. Když se budou moduly uvnitř dlaždic odkazovat pouze na služby, nebudou mít žádnou přímou vazbu na ViewModely stránek a bude možné je vyvíjet zcela odděleně. Použití služeb realizuje designové principy popsané v 2.5, konkrétně Oddělení zájmů a Persistance ignorance.

3.4.2 Služby existující v rámci přihlášení uživatele

Pro některé části aplikace bylo nutné vytvořit způsob pro uchovávání dat uživatele mezi HTTP požadavky. Tento problém řeší koncept LoginScopedServices, tedy služeb existujících pouze v rámci přihlášení uživatele. LoginScoped služby jsou poskytovány pomocí globální služby z vyšší vrstvy DI, která existuje pouze v jedné instanci v rámci běhu aplikace. Funguje jako takový centrální poskytovatel balíčků služeb[3], na který se odkazují všichni, kdo chtějí LoginScoped služby použít. Technicky je toto zajištěno pomocí další knihovny pro DI - Autofac.

Této službě je vždy předáno Id uživatele a typ požadované služby. Poskytoval zkontroluje, jestli už existuje balíček pro tohoto uživatele. Pokud neexistuje, vytvoří se nový balíček, pokud ano, vrátí již existující. Při odhlášení uživatele balíček služeb zaniká. Proces získávání služby pak popisuje obrázek 3.15.

[3]Technicky správný termín je ILifetimeScope.



Obr. 3.15: Diagram funkce LoginScopedServices

Mezi LoginScoped služby patří:

- Správa scény a dlaždic - nutné pro editaci a možnost vrácení změn
- Načítání dostupných Equipmentů a datových bodů - nutné držet odděleně, protože různí uživatelé mohou mít různá oprávnění k databázi
- Načítání dat z databáze do datových bodů pro aktuální hodnoty
- Načítání dat z databáze pro historické hodnoty
- Vyhodnocování a cachování výrazů
- Agregátor chyb v aplikaci

Toto řešení ale vytváří nový problém, kdy by bylo vhodné ošetřit stav, že uživatel zavře stránku a neodhlásí se. Jeho služby budou v tomto případě dále zabírat prostředky na serveru, i když už nemají žádný význam. Vhodné řešení by bylo dát každému balíčku datum expirace, které by se prodlužovalo s každým dotazem na data od uživatele. U systému vizualizace lze ale předpokládat, že počet uživatelů nebude vysoký a množství zabraných prostředků tak nebude významné.

3.5 Softwarové moduly

V této sekci budou popsány softwarové moduly, jakožto samostatné vizuální prvky vizualizace, které nám budou ukazovat data ze sledovaného technologického procesu. V prvé řadě bude popsáno, jak funguje společné rozhraní s dlaždicí a se zbytkem aplikace. Následně bude popsáno, co obnáší vývoj nového modulu a jeho přidání do aplikace. Poté budou postupně popsány moduly vytvořené v rámci této práce, které mohou poskytovat základ pro prvotní implementaci tohoto systému.

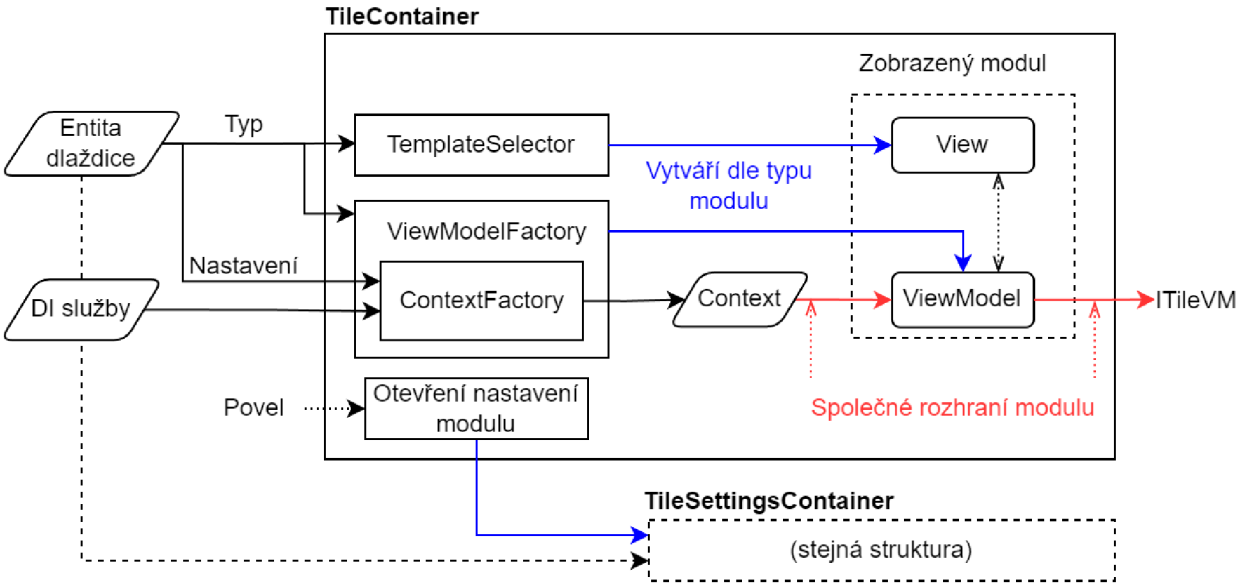
3.5.1 Společné rozhraní

Tok dat a způsob hostování softwarových modulů je ukázán na obrázku 3.16. Toto schéma je částečně zjednodušeno a View/ViewModel vrstva je znázorněna vedle sebe. Celý proces vytváření modulu začíná entitou dlaždice v databázi. Zde jsou kromě obecných nastavení dlaždice (poloha, navázaný Equipment id, atd.) také nastavení modulu (serializovaný JSON) a typ modulu (název třídy ViewModel). V systému existuje obecný objekt TileContainer, který na základě nastavení dlaždice vytváří instanci View a ViewModelu odpovídajícího modulu. Tomu pak předává všechna potřebná data (nastavení) a služby DI, zabalené do objektu TileContext.

Dlaždice se dále na vytvořený ViewModel odkazuje pouze pomocí společného rozhraní ITileVM, který musí dlaždice implementovat. Lze tedy říci, že TileContext a ITileVM jsou jediná místa tvořící rozhraní mezi modulem a zbytkem aplikace. ITileVM obsahuje pouze metodu Update(), která se volá při aktualizaci dlaždice.

Tato struktura je celá zopakována při otevírání nastavení dlaždice. TileContainer vytváří objekt TileSettingsContainer, kde ale hostujeme View a ViewModel **nastavení** modulu. Hostovaný ViewModel zde musí implementovat rozhraní ITileSettingsVM a zpravidla mu není nutné předávat objekt TileContext, protože v nastavení není důvod ho používat. Přes rozhraní se předává uložené nastavení a datové body. ViewModel nastavení pak musí zpět poskytnout seznam použitých výrazů a objekt nastavení k uložení do databáze.

Pro ViewModely modulu a nastavení modulu existují také jednoduché bázové třídy, kde jsou implementovány jednoduché metody pro načítání a ukládání nastavení do dlaždice. Tyto třídy není nutné používat, slouží pouze k zjednodušení vývoje modulů. Použití bázové třídy vytváří určitou vazbu mezi samotnými moduly, takže by neměly být rozšiřovány o složitější logiku.



Obr. 3.16: Hostování softwarového modulu v dlaždici

Z důvodu omezení DotVVM nelze přidávat za běhu aplikace nové softwarové moduly. Před spuštěním aplikace je nutné definovat všechny existující typy modulů v TemplateSelector a ViewModelFactory. Díky tomuto omezení také nelze realizovat načítání modulů jako externí .dll za běhu aplikace.

3.5.2 Vývoj softwarového modulu

Tato část se bude zabývat problematikou vývoje nového softwarového modulu. Může tedy sloužit jako návod pro budoucí vývoj modulů pro tuto aplikaci. Zároveň by měla doplňovat popis struktury dlaždice a aktualizaci scény v 3.5.1 a 3.3.4.

Vývoj nového modulu by měl probíhat dle následující souslednosti:

- Je nutné definovat typ modulu v databázi. V tabulce scene_tile_types definujeme sloupec [assembly], jehož hodnota by měla odpovídat plnému názvu třídy View nového modulu. Můžeme zde také definovat ve sloupci [default_module_settings] výchozí nastavení, které se aplikuje při vytvoření nové instance dlaždice.

id	name	assembly	default_module_settings
1	Binární indikátor	Container.Scenes.Tiles.BinaryIndicator	{"IsTrueExpression":null,"HtmlContent":"<div class...

Obr. 3.17: Definice typu modulu v tabulce scene_tile_types

- Poté je potřeba rozšířit enum TileContentType a přidat záznam do Dictionary v DataStorageService. Ten překládá název definovaný v databázi na konkrétní položku enumu.

Výpis 3.3: Definice TileContentType

```
{ "Container.Scenes.Tiles.BinaryIndicator",
    TileContentType.BinaryIndicator }
```

- Definujeme POCO objekt obsahující nastavení nového modulu do složky Model/Tiles. Nastavení je vhodné definovat jako třídu i pokud se chystáme ukládat jednoduchou položku (řetězec, číslo). To nám umožní jednoduše v budoucnu nastavení rozšiřovat.

Výpis 3.4: POCO třída nastavení modulu

```
namespace UniVizualizace.Model.Tiles
{
    public class BinaryIndicatorSettings
    {
        public string IsTrueExpression { get; set; }

        public string HtmlContent { get; set; }

        ... //další property nastavení modulu
    }
}
```

- Vytvoříme ViewModel modulu a jeho nastavení. Ty by se měly nacházet v adresáři ViewModels/Container/Scenes/Tiles. Nastavení je pak ještě v podadresáři /Settings. ViewModel třídy musí implementovat rozhraní ITileVM a ITileSettingsVM. Jejich implementaci si můžeme zjednodušit použitím bázových tříd BaseTileVM a BaseTileSettingsVM. Tyto bázové třídy jsou generické a jejich parametr je typ POCO objektu nastavení. Využití bázových tříd nám vyřeší ukládání a načítání nastavení, na které se pak můžeme jen jednoduše odkazovat. ViewModel modulu a jeho nastavení by měl vypadat podobně jak na výpisech 3.5 a 3.6.

Výpis 3.5: Příklad třídy ViewModel modulu

```
namespace UniVizualizace.ViewModels.Container.Scenes.Tiles
{
    public class BinaryIndicatorVM :
        BaseTileVM<BinaryIndicatorSettings>
    {
        public string HtmlContent { get; private set; }

        public BinaryIndicatorVM(TileContext context) : base(context)
        {
            HtmlContent = Settings?.HtmlContent;
            //bázová třída automaticky načte a vytvoří objekt Settings
            //typu definovaném v BaseTileVM<T>
        }
    }
}
```

Výpis 3.6: Příklad třídy ViewModel nastavení modulu

```
namespace UniVizualizace.ViewModels.Container.Scenes.Tiles.Settings
{
    public class BinaryIndicatorSettingsVM :
        BaseTileSettingsVM<BinaryIndicatorSettings>
    {
        public override List<string> GetUsedExpressions()
        {
            return new List<string>
            {
                Settings.IsTrueExpression
            };
        }
    }
}
```

V tomto případě je nutné zajistit navrácení všech použitých výrazů v metodě GetUsedExpressions().

- Přidáme do třídy TileViewModelFactory property obsahující ViewModel modulu a jeho nastavení. Třída obsahující nastavení musí mít atribut [Bind(Direction.Both)]. Dále rozšíříme metody metody Build a BuildSettings, kde podle vybraného enumu vytvoříme patřičný ViewModel. Nakonec musíme tento objekt přiřadit

do pomocných property ShownTile a ShownTileSettings.

Výpis 3.7: Vytvoření definice v TileViewModelFactory

```
namespace UniVizualizace.ViewModels.Container.Scenes.Components
{
    public class TileViewModelFactory
    {
        [Bind(Direction.Both)]
        public BinaryIndicatorSettingsVM BinaryIndicatorSett {...}
        public BinaryIndicatorVM BinaryIndicator {...}

        internal ITileVM ShownTile {...}
        internal ITileSettingsVM ShownTileSettings {...}
        // všechny property jsou { get; private set; }

        private readonly TileContextFactory contextFactory;
        ...
        //přiřazeno přes konstruktor

        public void Build(SceneTile tile)
        {
            switch (tile.TileType)
            {
                case TileContentType.BinaryIndicator:
                    BinaryIndicator = new
                        BinaryIndicatorVM(contextFactory.Create(tile));
                    ShownTile = BinaryIndicator;
                    break;
            }
        }
        ... //obdobným způsobem je realizovaná metoda BuildSettings
    }
}
```

- Vytvoříme View modulu a jeho nastavení. Jejich umístění by mělo být totožné s VM, ale cesta začíná složkou Views. Vytváříme zde DotVVM Control, takže soubory končící příponou .dotcontrol. Zde můžeme používat všechny dostupné možnosti DotVVM co se týkají těchto Controlů. Na začátek souboru přidáme direktivu @viewModel odkazující na ViewModel tohoto modulu. Příklad View nastavení lze vidět na výpisu 3.8. Zde díky použití společného rozhraní a bázové třídy nemusíme řešit načítání a ukládání nastavení a pouze rovnou editujeme

objekt Settings.

Výpis 3.8: View nastavení modulu

```
@viewModel ...(namespace VM)... .BinaryIndicatorSettingsVM,
    UniVizualizace
<div>
    Výraz (návratová hodnota bool)
    <dot:TextBox Text="{value: Settings.IsTrueExpression}" />
</div>
```

- Přidáme do SceneTileContainer.dotcontrol další prvek TemplateChoice, který odkazuje na dříve přidanou položku enumu TileContentType a jehož obsah je View modulu definovaný v předchozím kroku. Jeden prvek TemplateChoice lze vidět na výpisku 3.9. Položka enumu je specifikována pod Key a uvnitř prvku je náš dříve vytvořený View. S ViewModelem ho provážeme pomocí nastavení DataContext, kdy všechny definované ViewModely jsou dostupné pod property Implementation.

Výpis 3.9: Template selector

```
<dc:TemplateSelector SelectedKeyBinding="{value: TileContentType}">
            <dc:TemplateChoice Key="BinaryIndicator">
                <t:BinaryIndicatorTile
                    DataContext="{value:
                        Implementation.BinaryIndicator}" />
            </dc:TemplateChoice>
</dc:TemplateSelector>
```

- Obdobně přidáme do SceneTileSettingsContainer.dotcontrol další prvek TemplateChoice, který odkazuje na dříve přidanou položku enumu TileContentType a jehož obsah je View nastavení modulu.
- Nyní by měl být nový modul dostupný k přidání na scénu. Pokud mu chceme v menu přidat ikonu, je nutné ji definovat ve SceneVM v kolekci tileTypesIcons.

Výpis 3.10: Definice ikony modulu v menu

```
{ TileContentType.BinaryIndicator, "lightbulb" }
```

Následný postup se liší podle toho, jestli námi vyvíjený modul bude pracovat s aktuálními nebo historickými daty. V obou případech implementujeme logiku do metody Update() ve ViewModel modulu, která je volána s každou periodou obnovy.

- `Aktuální data` - zobrazování dat pomocí výrazů. Pokud byly použity nějaké výrazy (a byly správně vráceny z nastavení modulu), jsou v tuto chvíli již připravena všechna potřebná data a je nutné pouze výrazy vykonat a změnit nějakým způsobem stav modulu.
  Příklad vyhodnocování výrazu lze vidět na výpisku 3.11. Zde použijeme třídu ExpressionEvaluator a voláme vždy metodu Evaluate. Té předáme daný výraz a přetypujeme návratovou hodnotu na náš požadovaný typ. Ten musí být nullable, protože při neplatném výrazu bude návratová hodnota null. V případě, že je null, se automaticky vytvoří chyba v dané dlaždici. Tuto metodu můžeme volat vícekrát bez omezení, nedochází již ke čtení z databáze a výrazy jsou cachovány. V příkladu zároveň vidíme ošetření chyby, kdy uživatel nenastavil daný modul správně.

  Výpis 3.11: Aktualizace aktuální hodnoty v modulu

```
public override void Update()
{
    if (string.IsNullOrEmpty(Settings?.IsTrueExpression))
    {
        context.Tile.AddError("tile-settings", "Dlaždici chybí
            nastavení");
        return; //výraz nebyl definován
    }

    var evaluator = new ExpressionEvaluator(context);
    var isValueTrue = evaluator.Evaluate(Settings.IsTrueExpression)
        as bool?;

    if (isValueTrue == true)
    {
        ... //výsledek není null ani false
                //změníme stav UI modulu
    }

    context.Tile.RemoveError("tile-settings");
    //došli jsme až sem, takže nastavení bylo validní
}
```

- `Historická data` - zde se již nepoužívají výrazy, ale přímo odkazy na datové body. Vytvoří se požadavek na načtení historických dat, který je předán vy-

hrazené službě. Pokud uběhla perioda obnovy zobrazované části dat, dochází zde ke čtení z databáze. Příklad použití lze vidět na výpisku 3.12. Načtená historická data se nám vrací jako kolekce párů čas-hodnota.

Výpis 3.12: Aktualizace historických hodnot v modulu

```
public override void Update()
{
    ... //kontrola validního nastavení

    var timeThreshold =
        DateTime.Now.AddMinutes(-Settings.ShownMinutes);
    var request = new HistoryDataRequest
    {
        TileId = context.Tile.Id,
        EquipmentId = context.Tile.EquipmentId.Value,
        FromTime = timeThreshold,
        ValueSourceReference = Settings.SelectedValueSourceReference,
        RefreshInterval = Settings.RefreshPeriod,
    };

    var historyData =
        context.EquipmentHistoryData.LoadHistoryData(request);
    //data je nyní možné zobrazit
}
```

3.5.3 Binární indikátor

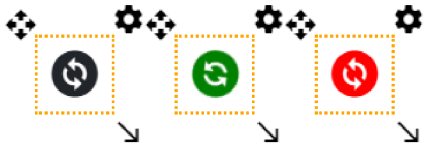
Tento indikátor umožňuje definovat HTML obsah, který se bude uvnitř něj zobrazovat. Poté se definují dva CSS styly, první je aplikován při hodnotě True(1) a druhý je aplikován při hodnotě False(0). Pokud nelze načíst hodnotu, není aplikován ani jeden ze stylů a jedná se tedy o mezistav - neznámá hodnota. Binární hodnota je vyhodnocována jako výsledek výrazu s návratovou hodnotou bool.

Jedná o velmi flexibilní prvek, díky možnosti specifikovat téměř libovolné HTML a CSS. Zároveň implementace jednoduchých ikon není příliš složitá, díky použití knihovny Material Icons. Pro obsah ikony lze definovat jednoduché HTML, kde obsahem prvku je název ikony.

Výpis 3.13: Zobrazení material-icons v HTML

```
<div class="material-icons">
  change_circle
</div>
```

Poté lze pomocí jednoduchého CSS color:(hodnota) měnit barvu indikátoru při změně binární hodnoty. Na obrázku 3.18 lze vidět příklad jednoduchého indikátoru. Tento indikátor ve stavu 0 zčervená, ve stavu 1 zezelená a začne se otáčet (pomocí CSS). Pokud hodnota nebyla načtena, nebo se vyskytla chyba, je indikátor černý.

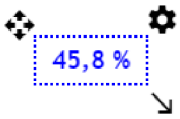


Obr. 3.18: Ukázka modulu Binární indikátor

3.5.4 Numerický indikátor

Numerický indikátor slouží k zobrazení číselné hodnoty. Tato hodnota je brána z výrazu, jehož návratovou hodnotou má být double. Očekává se tedy číslo s desetinnou čárkou, ale i výrazy s návratovou hodnotou integer (bez desetinné čárky) jsou brány jako validní. Dále je možné definovat C# formát pro zobrazované číslo. Tím lze definovat počet desetinných míst, nebo například zobrazovat s hodnotou jednotku. Nakonec jsou zde výrazy pro limity, kde se definuje maximum a minimum. S těmito hodnotami je zobrazované číslo porovnáváno a při překročení limitu se změní jeho barva. Při překročení maxima je aplikován CSS styl numeric-indicator-overlimit a ve výchozím nastavení hodnota zčervená. Při nedosažení minima je aplikován CSS styl numeric-indicator-underlimit a ve výchozím nastavení je hodnota modrá. Tyto styly jsou poté dostupné v uživatelsky editovatelném globálním CSS souboru. Pro možnosti dalšího přizpůsobení je zde možné nastavit vlastní CSS styl pro celý obsah této dlaždice.

Zde je vhodné připomenout, že ve všech výrazech je možné hodnoty upravovat a přepočítávat. Mohou být tedy výsledkem vzorce, do kterého vstupuje více proměnných, nebo zde může být hodnota upravována konstantami. Jako výraz lze také používat samostatné číslo, které pak představuje konstantní číselnou hodnotu a výraz je vykonán pouze jednou v rámci přihlášení uživatele.



Obr. 3.19: Ukázka modulu Graf historických hodnot

3.5.5 Vícestavový indikátor

Tento modul umožňuje zobrazovat neomezené množství stavů na základě definovaných výrazů. Uvnitř nastavení modulu je možné definovat (teoreticky) nekonečně dlouhý seznam dvojic výraz-stav, kdy stav vždy definuje aktuálně zobrazované HTML v modulu a výraz má vracet bool. Seznam je realizován ergonomicky, kromě přidávání a mazání položek lze samostatně editovat jednotlivé řádky a měnit jejich pořadí.

Aktualizace dlaždice probíhá tak, že jsou výrazy v seznamu postupně vyhodnocovány a cyklus se přeruší když výraz vrátil hodnotu True(1). Následně je aplikováno HTML přiřazené k tomuto výrazu.

Příklad využití tohoto indikátoru může být například pro entitu Motor ze starého systému vizualizace. Zde by bylo možné definovat vizuální styly pro všechny provozní stavy - běží, neběží, porucha, přehřátí, překročení motohodin. Na obrázku 3.20 je možné vidět příklad nastavení tohoto modulu s jednoduchými výrazy.

Podmínka	Zobrazené HTML	
(DateTime.Now - EngAI_Timestamp).TotalMinutes > 30	Ztráta komunikace	
EngAI_AnalogIn =< 2	Hodnota pod limitem	
EngAI_AnalogIN > 2 && EngAI_AnalogIN < 10	Hodnota OK	
EngAI_AnalogIN >= 10	Překročení hodnoty	

Přidat stav

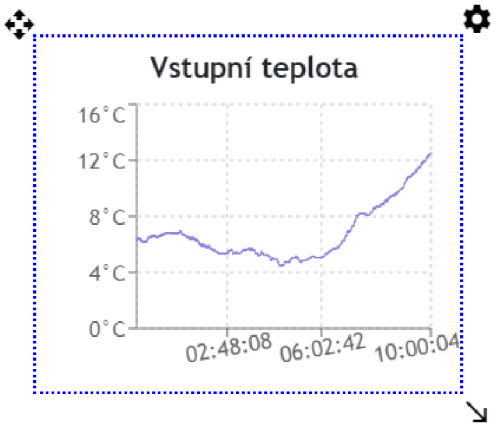
Obr. 3.20: Ukázka modulu Vícestavový indikátor

3.5.6 Graf historických hodnot

Modul grafu historických hodnot umožňuje jednoduchý náhled na vývoj hodnoty v čase. V jeho nastavení nejsou použity žádné výrazy, ale odkazujeme se přímo na hodnotu. Jsou zde také nabízeny pouze zařízení a profily historických hodnot. V grafu lze nastavit jeho popisek a jednotky na osách.

U historických dat je nutné specifikovat i periodu obnovy, protože je není vhodné načítat při každé aktualizaci scény. Pro načítání je použita služba pro načítání historických dat, která obsahuje interní cache požadavků a provede aktualizaci dat pouze při změně požadavku, nebo uplynutí periody obnovy.

Protože je graf složitější webová komponenta, byla pro jeho zobrazení zvolena knihovna ReCharts. Tato knihovna je postavená na komponentách React.js, takže ji bylo potřeba implementovat jako React modul do DotVVM. Tento modul je tak zároveň ukázkou, že struktura aplikace umožňuje implementaci těchto externích JS komponent.



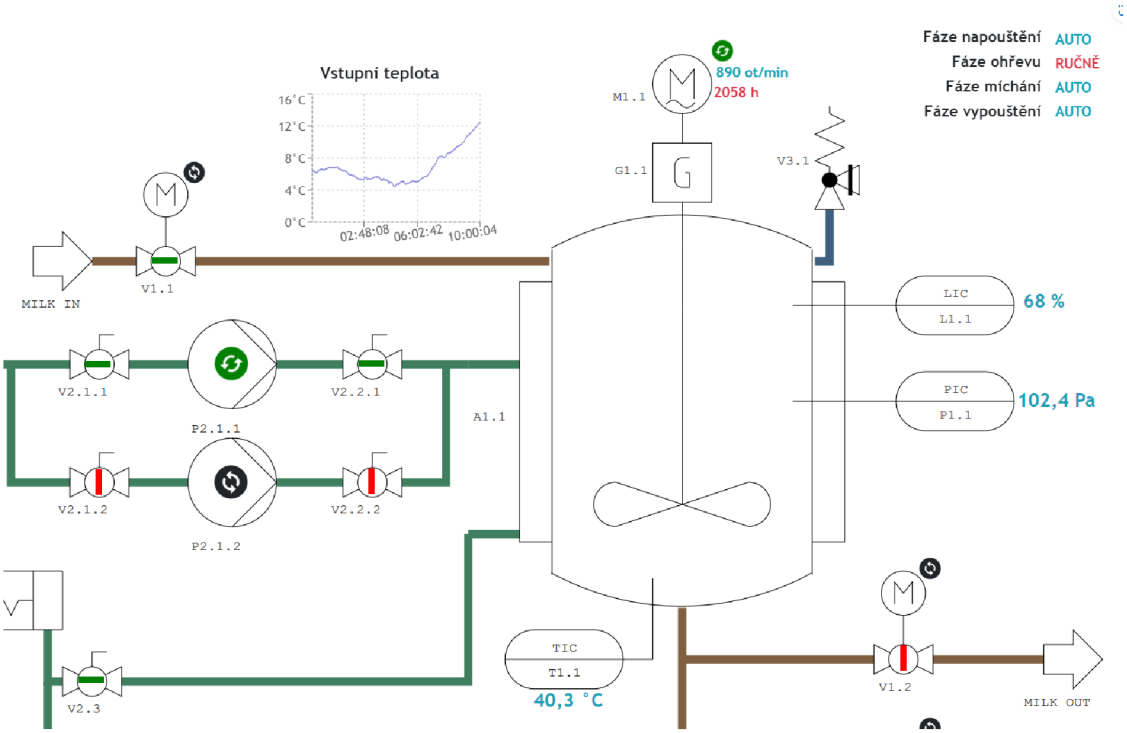
Obr. 3.21: Ukázka modulu Graf historických hodnot

3.5.7 Statický HTML obsah

Tento modul je asi nejjednodušší příklad modulu dlaždice pro tuto aplikaci. V nastavení lze pouze nastavit část HTML, která se poté zobrazuje jako obsah této dlaždice. Žádné jiné funkce zde nejsou implementovány. Může sloužit pro realizaci různých textových popisků, nebo dodatečných grafických elementů na scéně.

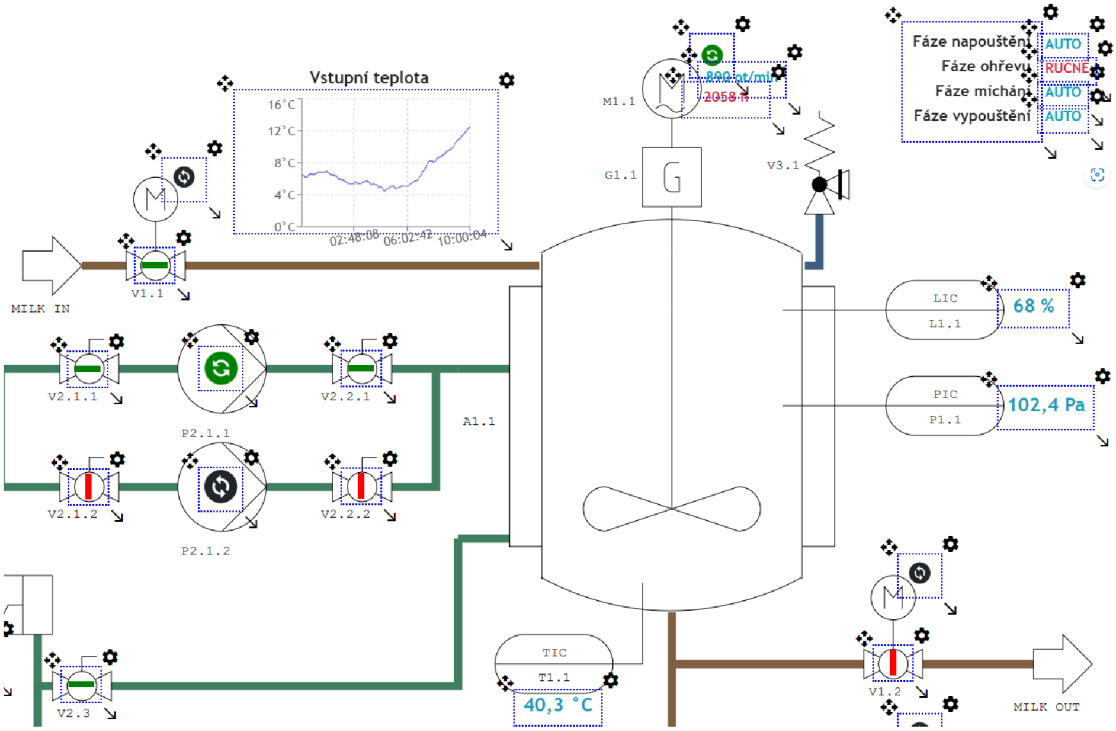
3.6 Nasazení aplikace

Pro vyzkoušení funkcí vizualizace vznikla vzorová implementace úlohy pasterizace mléka z předmětu Automatizace procesů (MPC-AUP). Na obrázku 3.22 lze vidět část jednoduchého P&ID schéma úlohy pasterizace. Ve schématu jsou rozmístěny například indikátory ukazující stav ventilů, čerpadel a fází řízení. Jsou zde pak také indikovány různé veličiny jako tlak, teplota a hladina v pasterizačním tanku. Jako příklad zobrazení historických dat je zde možné vidět průběh vstupní teploty pasterizovaného média. Byly zde použity všechny vzniklé softwarové moduly. Tato scéna vizualizace nebyla z časových důvodů nakonec napojena na reálná data. Všechna vizualizovaná data byla nasimulována.



Obr. 3.22: Příklad nasazení vizualizace na úloze pasterizace

Jak pak taková scéna vypadá v editačním režimu lze vidět na obrázku 3.23. Zde můžeme sledovat, které prvky jsou tvořeny obrázkem pozadí scény a které prvky jsou námi rozmístěné softwarové moduly.



Obr. 3.23: Příklad nasazení vizualizace na úloze pasterizace - editační režim

Závěr

V rámci této diplomové práce byl proveden rozbor stávajícího systému a byly identifikovány některé aspekty, na které se bylo vhodné zaměřit při tvorbě nové aplikace. Těmito aspekty byla vyšší modularita a spravovatelnost, zvláště týkající se napojení na datový model, správy scén, realizací a umisťování prvků na scéně.

Následně proběhl rozbor technologií, návrhových vzorů a designových principů pro tvorbu webových aplikací. Na začátek byly popsány pojmy .NET a ASP.NET, poté byla popsána platforma DotVVM Framework, která byla použita v novém i starém systému. Vedle toho byly uvedeny i další technologie související s DotVVM, ať už jako konkurenti, nebo jako předchůdci. Na konec byly popsány některé důležité designové principy a návrhové vzory pro tvorbu dobře spravovatelných aplikací.

Vznikla zcela nová DotVVM webová aplikace umožňující vizualizaci stavu systému v reálném čase a zobrazování historických dat. Aplikace umožňuje definovat jednotlivé vizualizační prvky jako samostatné softwarové moduly, které lze vyvíjet a spravovat zcela odděleně od zbytku aplikace. Zároveň umožňuje tyto prvky efektivně napojit na zcela libovolný datový model pomocí systému datových bodů a výrazů. Nová aplikace tvoří zcela funkční systém vizualizace a zajišťuje navíc funkce jako správu a editaci vizualizačních scén, systém přihlašování a systém notifikací pro uživatele.

Celé řešení je stavěno na zmíněných designových principech a s využitím několika vrstev Dependency injection. Díky tomu by mělo být dostatečně modulární a spravovatelné pro budoucí vývoj a nasazení pro sledování reálných systémů.

Vzniklo několik softwarových modulů realizující základní vizualizační prvky, jako binární, numerický a stavový indikátor. Pro vizualizaci historických dat vznikl modul grafu, který zároveň slouží jako příklad integrace React komponenty s DotVVM.

Pro ukázku funkce celé aplikace byla implementována vzorová scéna vizualizace realizující pohled na laboratorní úlohu Pasterizace z předmětu MPC-AUP.

Literatura

[1] DotVVM | GitHub.
URL https://github.com/riganti/dotvvm

[2] Stack Overflow Developer Survey 2022.
URL https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022

[3] Britch, D.; Schonning, N.; Dunn, C.; aj.: The Model-View-ViewModel Pattern - Xamarin.
URL https://learn.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

[4] Holec, M.: Porovnani Blazor Server a Blazor WebAssembly.
URL https://www.miroslavholec.cz/blog/pribeh-blazor-server-web-assembly

[5] Martin, R. C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, ISBN 9780132350884.

[6] Meta: React - Documentation.
URL https://react.dev/learn

[7] Microsoft: Choose an ASP.NET Core UI.
URL https://learn.microsoft.com/en-us/aspnet/core/tutorials/choose-web-ui

[8] Microsoft: .NET (and .NET Core) - introduction and overview.
URL https://learn.microsoft.com/en-us/dotnet/core/introduction

[9] Microsoft: .NET Framework - Lifecycle.
URL https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-framework

[10] Microsoft: Overview of ASP.NET Core MVC. 2022.
URL https://learn.microsoft.com/en-us/aspnet/core/mvc/overview

[11] Microsoft: Dependency injection in .NET. Březen 2023.
URL https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

[12] Reenskaug, T.: MVC XEROX PARC 1978-79.
URL https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html

[13] Rick-Anderson: Dependency injection in ASP.NET Core. Leden 2023.
URL https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

[14] Riganti: DotVVM | Documentation.
URL https://www.dotvvm.com/docs

[15] Riganti: DotVVM | FAQ.
URL https://www.dotvvm.com/faq

[16] Sanderson, S.: KnockoutJS Documentation.
URL https://knockoutjs.com/documentation/observables.html

[17] Smith, J.: Patterns - WPF Apps With The Model-View-ViewModel Design Pattern.
URL https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[18] Smith, S.: *Architecting Modern Web Applications with ASP.NET Core and Azure*. Microsoft Developer Division, .NET, and Visual Studio product teams, v6.0 vydání, 2022.
URL https://aka.ms/webappebook

[19] Toub, S.: Performance Improvements in .NET 6.
URL https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/

Seznam symbolů a zkratek

C# jednoduchý, moderní, mnohoúčelový a objektově orientovaný programovací jazyk vyvinutý firmou Microsoft v roce 2000

JIT Just In Time - pojem popisující vykonávání činnosti až když je právě potřeba

HTML HyperText Markup Language - značkovací jazyk používaný pro definici struktury webové stránky

CSS Cascading Style Sheets - jazyk používaný pro definici stylů používaných ve webové stránce

DOM Document Object Model - API zpřístupňující strukturu webové stránky jako logický strom

API Application Programming Interface - rozhraní komunikace mezi počítačovými programy

JSON JavaScript Object Notation - jednoduchý formát pro přenos datových objektů ve formě textu

REST Representational State Transfer - styl tvorby jednotného API pomocí URL cest, standardizovaných dotazů a objektů pro přenos dat

URL Uniform Resource Locator - definice adresy pomocí lomítek a parametrů, známá také jako webová adresa

POCO Plain Old CLR Object - označení pro jednoduché objekty (třídy) používané v aplikacích .NET

UI User interface - uživatelské rozhraní

JS Zkratka pro JavaScript

TS Zkratka pro TypeScript

DI Zkratka pro Dependency Injection

Seznam příloh

A Repositář projektu aplikace

Projekt aplikace je přiložen jako elektronická příloha. Jedná se projekt ASP.NET Core vyžadující pro spuštění Visual Studio 2022 a novější, s nainstalovanými balíčky pro ASP.NET Core a DotVVM. Řešení je nutné otevřít pomocí souboru UniVizualizace.sln. Otevře se nám řešení s několika projekty.

- `UniDataLayer` - obsahuje třídy repositářů pro přístup k databázi
- `UniDomain` - obsahuje generované POCO třídy, se kterými pracují repositáře
- `UniVizualizace` - projekt nové aplikace vizualizace