FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Validátor regulárních výrazů realizovaný jako Webassembly
Student:	Filip Figuli
Vedoucí:	Ing. Pavel Štěpán
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce zimního semestru 2022/2023

Pokyny pro vypracování

Cílem této práce je vyvinout webovou aplikaci se snadnou instalací. Aplikace má sloužit k vyhodnocení a následnému zobrazení regulárních výrazů s možností práce v offline režimu. Výsledný produkt bude dodán ve formě OCI kontejneru pro jednoduché nasazení v intranetovém prostředí.

- porovnejte výhody webových aplikací pracujících v offline režimu – zpracování citlivých údajů, dostupnost aplikace bez nutnosti instalace, snadný upgrade na novou verzi.
- seznamte s webovou technologií Webassembly
- seznamte se s .NET 5.0 BLazor framework
- seznamte se s technologií s technologií Docker
- implementujte stand-alone webovou aplikaci v technologii Blazor pro vyhodnocení a zobrazení regulárních výrazů
- dodejte aplikaci ve formě kontejneru (OCI)

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 14. září 2021 v Praze.

Bakalárska práca

VALIDÁTOR REGULÁRNÍCH VÝRAZŮ REALIZOVANÝ JAKO WEBASSEMBLY

Filip Figuli

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedúci: Ing. Pavel Štěpán
10. mája 2022

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2022 Filip Figuli. Všetky práva vyhradené..
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu: Figuli Filip. *Validátor regulárních výrazů realizovaný jako Webassembly.* Bakalárska práca. České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Obsah

Poďakovanie vi

Vyhlásenie vii

Abstrakt viii

Zoznam skratiek x

1 Úvod 1

2 Ciele práce 3
2.1 Štruktúra cieľov 3

3 Teoretická časť 5
3.1 WebAssembly 5
3.1.1 Príchod na trh 5
3.1.2 WebAssembly vs JavaScript 5
3.1.3 Kúzlo WebAssembly 6
3.2 ASP.NET Core 7
3.2.1 História 7
3.2.2 ASP.NET MVC 8
3.2.3 ASP.NET WebAPI 8
3.3 Blazor Framework 9
3.3.1 Progressive Web Apps 9
3.3.2 Blazor komponenty 9
3.3.3 SignalR 10
3.3.4 Blazor Server 11
3.3.5 Blazor WebAssembly 12
3.3.6 Blazor Hybrid 12
3.3.7 JavaScriptové knižnice 13
3.3.8 Závislosti 15
3.3.9 Singleton pattern 15
3.3.10 Notifikácie 15
3.4 Softvér na správu kontajnerov 16
3.4.1 Kontajnery 16
3.4.2 Nasadenie 17

3.4.3 Docker . . . 17
3.4.4 Kubernetes . . . 17
3.5 Databáza . . . 18
3.5.1 SQL vs NoSQL databázy . . . 18
3.5.2 Entity Framework . . . 19
3.5.3 Výber databázy . . . 20
3.6 Alternatívne riešenia . . . 20
3.6.1 Google Web Toolkit . . . 20
3.6.2 Angular . . . 21
3.6.3 React . . . 21

4 Praktická časť 23
4.1 Validátor regulárnych výrazov . . . 23
4.1.1 Vytváranie stromovej štruktúry . . . 23
4.1.2 Vykreslovanie . . . 25
4.1.3 Notifikácie . . . 26
4.1.4 Servisná vrstva . . . 28
4.2 Server . . . 29
4.2.1 Identity Server . . . 29
4.2.2 Repository-Service Pattern . . . 30
4.2.3 Testovanie . . . 31
4.2.4 Nasadenie . . . 31
4.2.5 Dokumentácia a inštalácia . . . 32

5 Zhodnotenie 35
5.1 Miera naplnenia našich cieľov . . . 35
5.2 Vývoj v Angular frameworku . . . 35
5.3 Vízia do budúcnosti . . . 36

6 Záver 37

Obsah priloženého média 41

Zoznam obrázkov

3.1 Štruktúra WebAssembly v prehliadači 7
3.2 Životný cyklus Blazor komponentu 10
3.3 Responzívny dizajn 14

4.1 Reprezentácia výstupu zhôd z metódy Match 24
4.2 UML diagram databázy 30
4.3 Náhlad dokumentácie vytvorenej programom Doxygen 32
4.4 Náhlad súboru s inštrukciami pre inštaláciu 33

Zoznam výpisov kódu

3.1 Singleton Trieda 16
4.1 Funkcia AddGroup 24
4.2 Funkcia ToHtml 26
4.3 Implementácia zobrazovania a následného mazania notifikácií 28

V prvom rade by som chcel poďakovať vedúcemu bakalárskej práce pánovi Ing. Pavlovi Štěpánovi za jeho trpezlivosť, ochotu a cenné rady. Ďalej by som chcel poďakovať môjmu otcovi. Bez neho by som nikdy nemal príležitosť študovať na tejto vysokej škole a táto práca by nemohla vzniknúť. Veľké ďakujem patrí aj mojim blízkym za povzbudivé slová, pomoc a podporu.

Vyhlásenie

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 10. mája 2022 ....

Abstrakt

Informačné systémy narástli na popularite a je po nich všeobecne dnes najväčší dopyt. Ich vývoj však nieje jednoduchý a často obnáša celý tým programátorov, ktorý sa špecifikújú na časti takéhoto systému od vizuálneho rozhrania klienta pre jednotlivé platformy, až po server s databázou na správu informácií.

Mojim primárnym cieľom je vytvoriť kompletný základ pre vývoj informačného systému a následne demonštrovať jednoduchou aplikáciou na validáciu regulárnych výrazov výhody, ktoré priniesol prelom vo svete webových technológií nazývaný WebAssembly. WebAssembly poskytuje elegantné riešenie, ktoré umožní pomocou jedného jazyka vývoj softvéru vrátane užívateľského rozhrania a zároveň multifunkčného serveru pre uchovávanie a spracovanie dát. Tento technologický pokrok výrazne uľahčuje vývoj tým, že nevyžaduje veľkú škálu rôznych technológií ako súčasný mainstream a je dnes plnohodnotne podporovaný každým bežným webovým prehliadačom.

Pre implementáciu som si zvolil .NET framework v jazyku C#. .NET poskytuje novú technológiu nazývanú Blazor, vďaka ktorej môžeme vytvárať standalone WebAssembly aplikácie. Napriek tomu, že takáto aplikácia je distribuovaná formou webstránky, kód je kompilovaný priamo u klienta, čo nás odbremení od potreby byť neustále v spojení so serverom a teda získame schopnosť práce v offline režime a to skoro natívnou rýchlosťou.

Aby sme zachovali jednotnosť jazykov pri vývoji serveru využijeme framework ASP.NET Core.

Výsledný softvér je schopný pracovať v dvoch režimoch. Offline režim demonštruje diskrétne spracovanie regulárnych výrazov bez strachu o to, že sa naše citlivé informácie dostanú do nesprávnych rúk a online režim umožnuje jednoduchú prácu s dátami za pomoci REST API z ASP.NET Core serveru. Vďaka built in CSS knižniciam je táto aplikácia responzízvna a vie sa prispôsobiť prakticky každému zariadeniu.

Prínosom je kvalitný základ pre veľkú škálu rôznych informačných systémov. Implementácia bola pomerne rýchla a jednoduchá vďaka dobrej dokumentácií. Nevyžadovala expertízu v obore okrem znalosti jazyka a základných programovacích konceptov. Vďaka veľkej modularite sa dá softvér ľahko rozšíriť podľa potreby.

Klíčová slova WebAssembly, C#, .NET, Blazor Framework, ASP.NET Core Framework, REST API, CSS

Abstract

Information systems grew in popularity and there is a great demand for them. However their development is complicated and usually takes several skilled developers to perform. Each of them focus on their own expertise ranging from platform specific user interfaces to development of server with database to store data.

My primary goal is to create a foundation for development of such information system. By creating a simple application for validation of regular expressions I want to demonstrate the advantages that WebAssembly brought with it by providing a briliant solution to create with a single programming language software that could create user interface as well as necessary server side to calculate and persist data. Such software would not only run with nearly native speed at client side but also would work on suitable platform of any choice. This technological progress singinficantly improves the development by not requiring a big scale of technologies as the current mainstream. It is already completely supported by majority of common web browsers.

I chose .NET framework with language C# for the implementation. .NET provides new technology called Blazor which has the ability to create standalone WebAssembly application. Nevertheless this application is being distributed in a form of a website, the code is compiled directly in the client device. Such feature allows us to enter offline mode without direct connection with a server. Its import to mention that the efficiency can compete with a native application.

To keep unification in programming langauges we will also use a part of .NET called ASP.NET Core to implement server.

Our final product is able to work in two modes. Offline mode provides discrete calculations of regular expressions without the fear of our sensitive data falling in to the wrong hands. Online mod on the other hand demonstrate easy processing of data with the help of a REST API provided by ASP.NET Core server. Included CSS libraries allow our application to be responsive, out of the box adjusting to any desired device.

The main benefit of our application is solid foundation for virtually any information system. The implementation it self was very fluent and simple thanks to good documentation. It does not require any in depth expertise except for the basic knowledge of C# language and programming concepts. Thanks to its high modularity it can be easily extended according to ones needs.

Keywords WebAssembly, C#, .NET, Blazor Framework, ASP.NET Core Framework, REST API, CSS

Zoznam skratiek

API	Application Programming Interface
GWT	Google Web Toolkit
WSL	Windows Subsystem pre Linux
EF	Entity Framework
HTTP	Hypertext Transfer Protocol
REST	Representational state transfer
W3C	The World Wide Web Consortium
MVC	Model-View-Controller
DOM	Document Object Model
WPF	Windows Presentation Foundation
SQL	Structured Query Language
JSON	JavaScript Object Notation
OCI	Open Container Initiative
CLR	Common Language Runtime
HTML	HyperText Markup Language
PWA	Progressive Web Apps
LINQ	Language-Integrated Query
CSS	Cascading Style Sheets
CRUD	Create, Read, Update, Delete
ORM	Object-relational mapping
UML	Unified Modeling Language

Kapitola 1
Úvod

V dnešnej dobe je čím ďalej, tým väčší dopyt po informačných systémoch, ktoré poskytnú užívateľom interaktívnu aplikáciu s možnosťou spravovať a ukladať rôzne dáta. Je už zabehnutou praxou pred vývojom zvoliť rad rôznych nástrojov, ktorými pokryjeme jednotlivé disciplíny, či už vizuálne spracovanie, biznis logiku, alebo perzistenciu informácií. To však odmysliac si samotné nástroje vyžaduje širokú škálu skúsených programátorov so špecializáciou na danú problematiku a výrazne komplikuje celý proces vývoja. Primárnou príčinou toho to problému je veľká škála zariadení, na ktorých sa snažíme náš softvér poskytovať.

Príchod webových aplikácií priniesol revolúciu v možnostiach ako poskytovať softvér viac menej na ľubovolnom zariadení na ktorom sa dá zmysluplne používať. So šikovnými nástrojmi, ako napríklad Angular, alebo React, sme dokonca schopní plnohodnotne nahradiť natívne aplikácie do takej miery, že bežný užívateľ nerozpozná rozdiel. Neustále je vyžadovaný rad rôznych technológií od JavaScriptu pre vizuálne prevedenie a taktiež často používanú Javu na restový server a biznis logiku.

Nedávno na trh prišla nová vízia. Spraviť jeden webový štandard, ktorý umožní spustiť kód napísaný v rôznych jazykoch priamo na webstránke skoro natívnou rýchlosťou čo doteraz nebolo možné. Je tento nový prístup prelomom vo svete informačných systémov a webstránok, alebo len ďalším neúspešným pokusom, ako bol Google Web Toolkit (GWT)?

V mojej bakalárskej práci sa budem snažiť implementovať riešenie, ktoré bude tvoriť základ pre informačný systém a demonštrovať rôznu základnú funkcionalitu na ukážkovom programe. Táto aplikácia bude vyhodnocovať regulárne výrazy. Jednou z hlavných vlastností, ktoré takýto program vie poskytovať bude práca v offline režime, čo je v prípade regulárnych výrazov veľmi kľúčový aspekt práve kvôli zmluve o mlčanlivosti. Vo svete programovania je takýto dodatok v pracovnej zmluve viac ako bežný, čo často znemožňuje pohodlné používanie výrazov priamo na citlivé dáta v komerčných aplikáciach, kde nevieme garantovať diskrétnosť.

Regulárne výrazy ako ukážkový program som zvolil najmä pre viditeľnú výhodu

v standalone prevedení, ktoré však stále vysoko benefituje pridanou schopnosťou ukladania výrazov, či textov do databázy na neskoršie použitie. Moja práca sa však nezaoberá priamo implementovaním algoritmu, ktorý by tieto výrazy analyzoval a aplikoval. Na túto funkcionalitu bola použitá už existujúce knižnica.

Kapitola 2
Ciele práce

Primárnym cieľom práce je vytvoriť standalone progresívnu webovú aplikáciu schopnú pracovať v aj v režime offline. Pre ukladanie a spracovanie dát je nutné vytvoriť serverovú aplikáciu, ktorá zabezpečí komunikáciu medzi perzistentným médiom a klientom. Bezpečnosť dát bude realizovaná užívateľskými účtami a autorizáciou.

Úlohou klientskej aplikácie bude demonštrovať využitie komplexného jazyku, v našom prípade C#, a to priamo v browseri za pomoci WebAssembly. Zároveň musí poskytovať možnosť upravovať dáta a komunikovať so serverom čo však bude umožnené len pre registrovaných užívateľov a ich vlastné dáta. Pre ľahkú distribúciu a prácu v offline režime bude vytvorená vo forme progresívnej webovej aplikácie.

Serverová aplikácia musí tiež svoju implementáciu realizovať v jazyku C#. Pre uľahčenie budúceho vývoja chceme zaručiť čo najväčšiu modularitu, takže bude nasledovať Repository-Service pattern. Zároveň potrebujeme zabezpečiť autorizáciu užívateľov v prípade, že klientská aplikácia je v online móde.

Tieto aplikácie chceme distribuovať v ľahko nasaditeľnom a upgradovateľnom formáte a to vo formáte OCI kontajneru a naviac aj s perzistentným médiom.

2.1 Štruktúra cieľov

- Implementácia klienta
  - Komponenty a triedy potrebné na funkcionalitu
  - Ukážkový validátor regulárnych výrazov
  - Rozhranie pre vkladanie notifikácií
  - Servisná vrstva pre komunikáciu so serverom
- Implementácia serveru

  - Pridanie nevyhnutných prvkov pre autorizáciu
  - Vytvorenie jednotlivých tried reprezentovaných v databáze
  - Vytvorenie Repository-Servis patternu na prístup k databáze
  - Vytýčenie kontrolerov pre komunikáciu so serverom
- Integračné testy
- Nasadenie aplikácie
  - Vytvorenie kontajneru pre PostreSQL databázu
  - Publikácie serveru do súboru Dockerfile a následné vytvorenie kontajnera aplikácie
  - Po úspešnom spustení aplikácie stiahnutie standalone verzie klienta

Kapitola 3
Teoretická časť

Táto kapitola nám priblíži všetky dôležité technológie a postupy, ktoré nám umožnili zrealizovať našu víziu.

3.1 WebAssembly

Táto sekcia sa bude venovať technológií, ktorá priniesla revolúciu vo vývoji softvéru. V našom prípade nám WebAssembly priamo zabezpečuje .NET runtime, v ktorom sa vykonáva naša klientská aplikácia.

3.1.1 Príchod na trh

Od roku 2019 WebAssembly bolo oficiálne pridané organizáciou W3C medzi webove štandardy[1]. Nový nízkoúrovňový programovací jazyk prichádza s hlavnými výhodami ako je napríklad kompaktný binárny formát a takmer zhodná efektivita ako nátivná aplikácia. Primárna myšlienka za týmto nápadom spočíva v umožnení ďaleko komplexnejších a silnejších jazykov, ako napríklad C/C++ či C#, priamo fungovať v prehliadači, čo doteraz, až na pár neúspešných pokusov, ako bolo GWT nebolo možné[2].

W3C, alebo tiež The World Wide Web Consortium[3], je medzinárodná komunita organizácií, ktoré spolupracujú na tvorbe webových štandardov. Ich schválenie je prísľubom svetlej budúcnosti pre WebAssembly otvárajúc dvere do prakticky každého webového prehliadača.

3.1.2 WebAssembly vs JavaScript

Je potrebné vysvetliť, že technológia, ktorá umožňuje komplexný prístup a prácu priamo v prehliadači už dávno existuje. JavaScript[4] bol pôvodne vyvinutý ako skriptovací jazyk a to len za 10 dní v roku 1995. Napriek tomu, že ako taký nebol nikdy predurčený sa stať primárnym jazykom webstránok, nabral pomerne rýchlo

na popularite a dnes je jeden z najpoužívanejších programovacích jazykov na svete. Pochopiteľne jeho pôvodný úmysel dnes prináša mnoho problémov. Nízka efektivita, objemné knižnice, komplikovaná kompilácia a veľa iných nedostatkov, viedlo k vzniku WebAssembly. Cieľom WebAssembly nikdy nebolo nahradiť JavaScript, práve naopak úzko spolupracujú a je vitálnou súčasťou celého životného cyklu aplikácie. Kým JavaScript umožňuje WebAssembly fungovať tým, že spustí jeho modul a prepojí funkcionalitu s okolitým svetom. WebAssembly svojou efektivitou a kompaktnosťou dokáže vykonávať komplexné algoritmy v prijateľnom čase s vysokou účinnosťou.

3.1.3 Kúzlo WebAssembly

Webstránky sa skladajú z dvoch častí viz obrázok 3.1. Virtuálnym strojom, ktorý zabezpečuje beh našej aplikácie a vykonáva väčšinou JavaScriptový kód. Tento kód dokážeme ovládať pomocou programovacieho prostredia aplikácie, API čím získava naša aplikácia kontrolu nad funkcionalitou prehliadača[5].

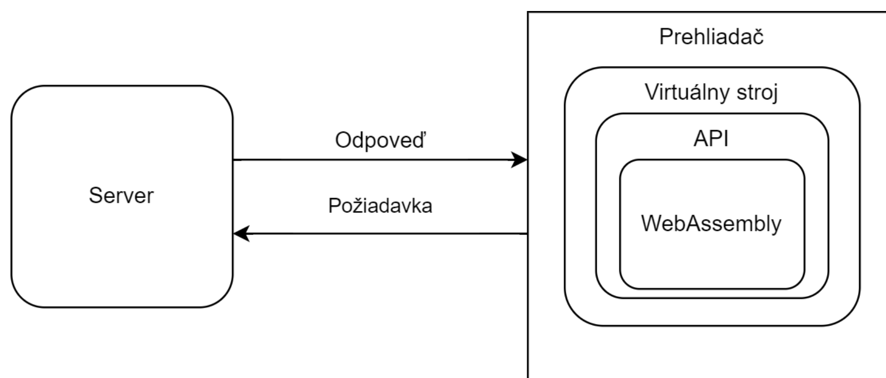
WebAssembly aplikácia sa skladá z niekoľkých kľúčových súčastí. Jednou z nich je modul, ktorý je kompilovaný už priamo v prehliadači do binárneho formátu, ďalej vyrovnávajúca pamäť s ktorou modul smie pracovať s nizkoúrovňovými inštrukciami a tabuľkou odkazov. Inštancia takéhoto modulu spája a umožňuje prístup ku všetkým týmto súčastiam. JavaScript vďaka svojmu API dokáže vytvoriť inštanciu, cez ktorú môže využívať jednotlivé súčasti a sprístupňovať ich priamo programátorovi ako JavaScriptové funkcie.

WebAssembly sa dá vyvíjať rôznymi spôsobmi. Je reprezentovaný aj kódovou formou, ktorá umožňuj programátorovi písať priamo zdrojový kód. To v prípade takto nízko-úrovňového jazyka nie je vôbec ľahké a ani zďaleka efektívne, kvôli tomu existuje alternatíva s názvom AssemblyScript. Takýto script sa námatkovo podobá na JavaScript resp. jeho zjednodušenú verziu TypeScript. Kód je následne kompilovaný priamo do binárneho formátu, ktorý už vie prehliadač vykonávať.

To však okrem efektivity neprináša žiadnu marginálnu výhodu. Hlavná sila WebAssembly prichádza so schopnosťou preniesť vysoko-úrovňový programovací jazyk a jeho potencial do prehliadača, pričom aplikácia neutrpí na funkčnosti, alebo rýchlosti. Pre takýto vývoj máme dve možnosti. Môžeme našu aplikáciu napísanú v inom jazyku skompilovať priamo do kódu WebAssembly, ktorý je následne ako modul načítaný JavaScriptom a cez jeho API komunikuje so zvyškom webstránky. Takto skompilovaný kód však nie je priamo binárna reprezentácia strojového kódu pre špecifický systém, naopak nám môže námatkovo pripomínať ByteCode pre Javu, alebo Common Itermediate Language pre C#. WebAssembly v takomto formáte je následne spracovávané virtuálnym strojom v prehliadači, ktorá ďalej kód interpretuje.

Na takéto účely už existuje mnoho kompilátorov pre často používané jazyky ako napríklad C, alebo C++. Alternatívne napríklad .NET využíva skompilovanú verziu svôjho virtuálnej stroja, ktorý vykonáva C# kód priamo v prehliadači. Takýto

virtuálny stroj, v našom prípade Common Runtime Language[6], je nevyhnutnou súčasťou životného cyklu celej aplikácie. Okrem vyhodnocovania samotného kódu má na starosti veľa iných vitálnych funkcií. Týmto spôsobom WebAssembly zabezpečí vykonávanie nášho C# kódu, zatiaľ čo JavaScript odkomunikuje potrebné prepojenie s prehliadačom ako takým.



Obr. 3.1 Štruktúra WebAssembly v prehliadači

3.2 ASP.NET Core

ASP.NET predstavuje riešenie Microsfotu na tvorbu webstránok za pomoci .NET Frameworku. Dlho sa vyvíjalo a prešlo mnohými zmenami, kým sa dostalo do stavu v akom ho dnes používame v našej aplikácii. V tejto sekcií si ucelene vysvetlíme ako dnes funguje a čo tomu predchádzalo.

3.2.1 História

Aby sme mohli plnohodnotne pochopiť ako zmenil príchod Blazor Frameworku prístup Microsoftu k danej problematike musíme si najskôr vysvetliť čo novým riešeniam predchádzalo, a ako sa web v ich podaní vyvíjal predtým ako sa dostal do stavu v akom ho dnes využívame v našej práci. Microsoft prišiel s vlastným riešením pre tvorbu webstránok a nazval ho ASP.NET[7]. Veľkú škálu rôznych aplikácií pokryli dvoma rôznymi verziami ASP.NET MVC a ASP.NET WebAPI. Žiaľ ani jedna z týchto technológií nebola perfektná, keďže MVC v modernej dobe webových aplikácií pri svojej potrebe obnovovať webstránku po každej interakcií posielalo vždy nanovo vygenerované HTML súbory, čo výrazne zvyšovalo množstvo dát tečúcich medzi serverom a klientom. Takto realizované riešenie nebudilo tak plynulý dojem, ako konkurenčné aplikácie napríklad v Angular Frameworku firmy Google. WebAPI na druhej strane trpelo nutnosťou pre využitie viacerých technológií, ktoré následne trpeli ich vzájomnou nekompatibilitou. Napríklad entity realizované v perzistentnej vrstve WebAPI nemohli byť použité v aplikácií priamo,

ale museli sa serializovať na ekvivalentné triedy v danej technológií a mnoho iných komplikácií.

3.2.2 ASP.NET MVC

ASP.NET MVC využíva C# kód na produkovanie takzvaných komponentov. Tieto komponenty sa skladajú z dvoch súborov. Jeden reprezentujúci finálny dokument v HTML a druhý pridávajúci pomocou C# kódu rôznu funkcionalitu našej HTML stránke. Klient si najskôr vypýta webstránku ako takú, a potom posiela naspäť celý HTML formulár so zmenami. Server ich následne spracuje a podľa implementácie zareaguje na rôzne zmeny, napríklad kliknutie tlačítka či vyplnenie formulára. Následne vygeneruje opäť celý dokument, už upravený, ktorý posiela naspäť klientovi, kde je následne nanovo vykreslený do prehliadača. Microsoft časom zmenil renderovaciu technológiu a ASP.NET Web Forms na takzvané Razor Pages. Razor nepriniesol výraznu zmenu vo funkcionalite, ale zjednodušil spôsob akým sa komponenty programovali, a tým prioritne uľahčil programovanie ako také, ale aj skrátil čas potrebný na pochopenie tejto technológie. Jedna komponenta či stránka sú reprezentované jedným súborom, ktorý v sebe zapúzdri implementáciu spolu s HTML reprezentáciou komponentu. Tým dosiahli zamedzeniu opakovania zbytočného kódu a zlepšili organizáciu v projekte čo umožnilo zrýchlenie vývoja. Tento prístup neskôr adaptoval Blazor Framework vo forme Blazor Serveru, ktorému sa budeme bližšie venovať v nasledujúcej sekcii[7].

3.2.3 ASP.NET WebAPI

ASP.NET WebAPI poskytuje aplikačné programovacie rozhranie. Inými slovami je to aplikácia, ktorá je pripravená prijímať, v tomto prípade HTTP požiadavky, a adekvátne odpovedať vo formáte, ktorý je vopred stanovený. WebAPI je realizovaná vo forme serveru, ktorý vystaví na príslušné URL adresy takzvané kontrolery. Každý takýto kontroler má fixne danú štruktúru a formát dát, ktoré je schopný prečítať a odozvdať zvyšku aplikácie na spracovanie. Tento prístup však neprináša žiadne vlastné užívateľské rozhranie, takže je závislý na kompatibilnej implementácií klientskej aplikácie. Riešenie problému s nekompatibilitou prinieslo až WebAssembly a jeho adaptácia vo forme Blazor WebAssembly aplikácie. Bližšie si ju vysvetlíme v nasledujúcej sekcii[7].

3.3 Blazor Framework

S príchodom nových technológií ako WebAssembly či SignalR prišla aj adaptácia v prevedení nového ASP.NET Core formou Blazor Frameworku. Úlohou tejto sekcie bude ukázať ako Blazor Framework priniesol riešenia, s ktorými sa potýkal ASP.NET počas svojej existencie a vysvetlíme si ako jednotlivé verzie tohoto frameworku fungujú.

3.3.1 Progressive Web Apps

Progresívna webová aplikácia je aplikácia bežiaca v prehliadači[8]. Na rozdiel od klasickej webstránky však disponuje vlastnosťami ako natívné aplikácie. Prehliadač funguje v pozadí ako sprostredkovateľ, ale aplikácia samotná budí dojem bežnej aplikácie. Dá sa spustiť priamo zo zariadenia a podporuje aj prácu v offline režime. Veľkou výhodou je podpora rôznych operačných systémov a jednoduché nasadenie, keďže užívateľovi stačí otvoriť aplikáciu v prehliadači a následne si ju môže nainštalovať do svôjho zariadenia.

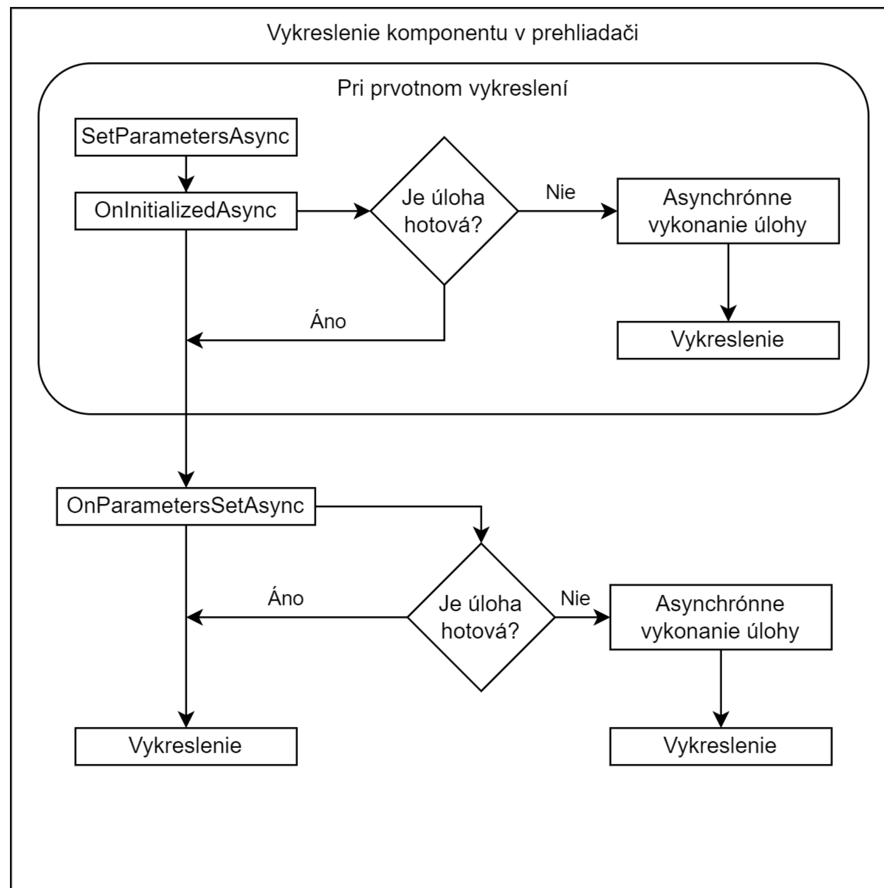
Takto nainštalovaná aplikácia má prístup k súborom na disku, pripojenému hardvéru a v podstate rovnakú škálu možností aké by mala čisto nativná aplikácia. S príchodom WebAssembly sa dnes darí progresívnym webovým aplikáciam konkurovať v efektivite nátivnym aplikáciam. Tento fakt je veľmi kľúčový, lebo umožňuje realizovať aj komplikované algoritmy a funkcionality priamo u klienta. Dobrým príkladom je svetoznámy softvér AutoCAD, ktorého tridsaťročný zdrojový kód vývojári dokázali spustiť v prehliadači[9].

Vizuálna časť aplikácie je realizovaná formou HTML a CSS. Vďaka knižniciam, ako napríklad Bootstrap, vieme vytvárať responzívny dizajn a tým dosiahneme želaného vzhľadu na prakticky ľubovoľnom rozlíšení.

3.3.2 Blazor komponenty

Základný princíp Blazor Frameworku[10] spočíva vo vytváraní komponentov a ich následnému zobrazovaniu do HTML ekvivalentnej reprezentácie. Napriek tomu, že to na prvý pohľad pripomína už spomínané Razor Pages, životný cyklus komponentov viz obrázok 3.2 a ich spracovanie na serveri je rozdielne. Nedochádza tu ku klasickým požiadavkám na aké sme v HTTP zvyknutí. Klientská aplikácia oboznamuje server o zmene stavu a server na to adekvátne reaguje. Blazor zachováva razor formát súborov, takže súbory si držia svoju HTML reprezentáciu aj s ich vlastnou implementáciu v razor objekte. Tento objekt sa neskôr vyrenderuje do výsledného Document Objet Model (DOM). Predtým ako ho uvidíme zobrazený v prehliadači však musí vykonať sadu metód, ktoré zabezpečia načítanie parametrov, inicializáciu a renderovanie samotného dokumentu[11]. Dokument po vykonaní zmien u klienta a oboznámení serveru o zmene je odoslaný naspäť, kde je porovnaný s poslednou vytvorenou verziou toho istého objektu, ktorý server na-

posledy odoslal. Tým server získava zmeny vykonané klientom a adekvátne na ne reaguje. Po vykonaní potrebných operácií server opakovane renderuje objekt avšak neposiela celý dokument naspäť, ale len útržok v ktorom prišlo ku zmene voči poslednej odozve od klienta. U klienta je JavaScriptové rozhranie, ktoré vie takýto útržok prijať a za behu aplikácie ho aplikovať. Týmto Blazor budí podobný plynulý dojem ako spomínaný Angular bez potreby renderovanie celého webu nanovo pri každej zmene.



Obr. 3.2 Životný cyklus Blazor komponentu

3.3.3 SignalR

Každý sa určite už stretol s protokolom HTTP, napríklad keď píšeme adresu webstránky v prehliadači. Tento protokol nám umožňuje komunikovať medzi stránkou a serverom na princípe požiadavka a odpoveď. Za bežných okolností si server neudržuje informácie o predošlej komunikácií s klientom, snaží sa len vyriešiť aktuálnu požiadavku a na tú odosiela konkrétnu odpoveď. Nevýhodou tejto komunikácie je, že nie je plynulá. Všetky dáta, ktoré klient obdrží sú len dôsledkom

požiadavky, teda server svojvoľne nič neodosiela. Komunikácia vždy vzniká a zaniká jednou požiadavkou.

Riešenie na tento problém prináša knižnica SignalR[12] pomocou niekoľkých technológií. WebSocket, Server Sent Events, Forever Frame a Long Polling. Postupne sa pokúša využiť každú z nich. Prvý je WebSocket. V prípade, že nie je podporovaný na klientovi, alebo serveri otestuje či je dostupné Server Sent Events. Ak ani ten nie je dostupný, obráti sa na Forvere Frame a napokon vždy môže použiť Long Polling.

WebSocket je protokol, ktorý vytvára trvalý komunikačný kanál. Komunikácia prebieha v oboch smeroch a to v rovnakom čase. Stále využíva HTTP pre zahájenie komunikácie, ale už viac nezaniká po odoslaní odozvy zo serveru.

Server Sent Events fungujú na princípe, kedy sa klient ohlási serveru pomocou HTTP a vysvetlí mu akým spôsobom je pripravený spracovávať dáta. Server potom môže bez limitácií posielať čo je potrebné klientovi.

Forever Frame využíva HTML element s názvom iframe. Umožňuje nám vytvoriť spojenie so serverom, kde iframe je ako vstupná brána, ktorú môže ovládať server priamo a dodávať jej dáta bez toho aby klient musel posielať požiadavky.

Long Polling je technika, ktorá pracuje na princípe HTTP požiadavka a odpoveď. Rozdiel je v tom, že server miesto instantnej odozvy predĺži čas, dokiaľ odpoveď posiela dovtedy kým sa pripravia všetky potrebné dáta.

SignalR využíva v základe Blazor Server, avšak vieme ho využiť aj v našej Blazor WebAssembly aplikácií. Jeho jedna zaujímavá vlastnosť je, že si server vie udržiavať skupinu takýchto SignalR konekcií a zdieľať im spoločné dáta. Klienti si začnú odoberať dáta od konkrétnej skupiny SignalR konekcií a rovno ich spracovávať hneď ako sa zmenia.

3.3.4 Blazor Server

Blazor Server reprezentuje ASP.NET MVC prístup. Aplikácia stále žije svojím životom na serveri, ale spojenie s klientom udržuje spojenie cez SignalR. To prináša niekoľko výhod. Množstvo dát u klienta je výrazne menšie, aplikácia žijúca na serveri má k dispozícií kompletné celý .NET, vývoj je výrazne uľahčený možnosťou ladenia priamo kódu aplikácie, samotný kód sa nedostáva ku klientovi a hlavne asynchrónne spracovanie požiadaviek a teda plynulosť aplikácie bez potreby neustáleho obnovovania webstránky. Za to však platíme nemalú cenu formou zvýšenej latencie, veľkej záťaži na server pri viacerých užívateľoch a prichádzame o schopnosť aplikáciu dodávať aj vo forme offline režimu.

Server za pomoci spomínaného SignalR umožňuje stabilnú asynchrónnu komunikáciu s klientom. Každá, hoci malá zmena je odpropagovaná na server, kde sa následne spracuje a podľa životného cyklu komponentu odošle naspäť potrebné informácie[10].

3.3.5 Blazor WebAssembly

Teraz keď už máme predstavu čo je WebAssembly a rozumieme úplne základným princípom ASP.NET Core poďme sa bližšie pozrieť ako nám to celé poprepája Blazor WebAssembly. Jeho úlohou je vytvoriť C# aplikáciu, ktorá bude vďaka CLR schopná plnohodnotne fungovať v prehliadači. K tomuto využíva práve druhý vyššie spomínaný spôsob využitia, kedy sa celý runtime skompiluje do binárneho prevedenia a následne vykonáva C# kód v interpretovanej forme.

Aj v tomto prípade sa životný cyklus komponentov veľmi nemení avšak už nemáme server, ktorý by vykonával potrebné zmeny v dokumente. Tento krát nám úlohu serveru zabezpečuje priamo aplikácia, ktorá vďaka WebAssembly vie plnohodnotne fungovať priamo u klienta. Tento prístup až na jednoduché aplikácie sám o sebe nemôže plnohodnotne fungovať a preto prichádza na scénu už spomínaný ASP.NET WebAPI, ktoré v novom prevedení ASP.NET Core sa vie vyvíjať zarovno s Blazor WebAssembly aplikáciou. Klientská aplikácia sa teda podľa potreby obracia na vytýčené kontrolery našej serverovej API, ktorá nám v tomto prípade poskytuje prepojenie medzi aplikáciou a perzistentným médiom resp. databázou.

Takéto riešenie až na občasné požiadavky voči serveru pracuje úplne samostatne u klienta. Hlavnou výhodou je okrem instantnej odozvy bez latencie aj práca v offline režime a zároveň možnosť distribúcie výpočtovej záťaže medzi serverom a klientom. Pre predstavu ak potrebujeme zrátať malú matematickú rovnicu pre náš server to nie je veľká záťaž tak ako ani pre klienta, avšak ak bude veľké množstvo klientov chcieť od serveru zrátať veľké množstvo malých operácií, dokážu ho výrazne spomaliť zahltením. V prípade Blazor WebAssembly aplikácie máme možnosť rozdistribuovať záťaž medzi server a klienta tak, aby časté jednoduché operácie mohol vykonať sám klient a server využiť len na špecifické komplikované úlohy, ktoré sám nie je schopný vykonať. Taktiež, takýto prístup pri ktorom máme väčšinu implementácie priamo u klienta umožňuje vytvoriť takzvanú progresívnu webovú aplikáciu, ktorú je možné spúšťať v offline režime. Takéto prevedenie je možné sledovať v implementácií tejto bakalárskej práce, kde validovanie regulárnych výrazov je vykonávané u klienta a plnohodnotne funguje aj v prípade nedostupnosti servera[10].

3.3.6 Blazor Hybrid

Blazor v jeho hybridnej forme je zatiaľ len ukážka konceptu a teda nie finálny produkt, nie je vhodný pre produkciu. Stále si však myslím, že si zaslúži svoju osobitnú časť ako jeden z hlavných pilierov, na ktorom Blazor stojí.

Tak ako zvyšné režimy hostovania aj Hybrid využíva už veľa krát spomínaného prístupu Blazoru k jeho komponentom a ich životným cyklom. V tomto prípade však nehrá žiadnu rolu server a ani aplikácia vykonávaná vo WebAssembly CLR. Za týchto okolností je implementácia realizovaná ako natívna aplikácia. Pôvodný princíp razor komponentov zostáva stále zachovaný vďaka vstavanému webovému

zobrazeniu, ktoré bez pomoci prehliadača vie zobrazovať HTML dokument. Navyše je možné použiť existujúcu technológiu Windows Presentation Foundation (WPF) s novým kontrolom BlazorWebView, ktorý vie vykresliť Blazor komponenty v časti aplikácie zatiaľ čo zvyšok beží nativne vo WPF. Spracovanie zmeny stavu komponentov a celú zvyšnú logiku vieme v tomto prípade vykonávať priamo v systéme bez potreby dodatočného serveru či CLR[10].

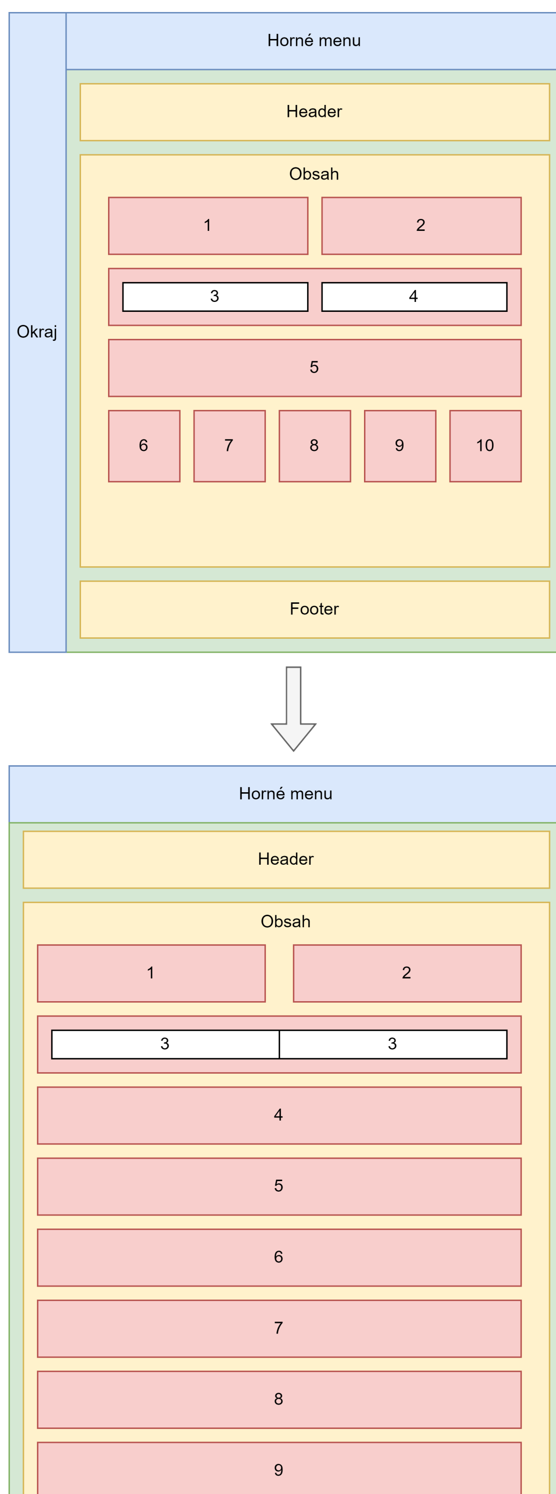
3.3.7 JavaScriptové knižnice

WebAssembly priamo komunikuje s JavaScriptom čo znamená, že pri využívaní Blazor Frameworku máme k dispozícii ľubovoľné JavaScriptové knižnice. Či už populárny Bootstrap na tvorbu responzívneho dizajnu, alebo dnes priam nevyhnutné JQuery na dodatočné manipulovanie s dokumentom.

Bootstrap je knižnica tvorená CSS a JavaScriptom. Jej cieľom je umožniť vytvárať responzívny dizajn webstránky. Takýto dizajn na základne istých pravidiel sa snaží prispôsobiť rôznym rozlíšeniam ako sú veľké stolné monitory, alebo aj malému displeju mobilného zariadenia. Princíp Bootstrapu je založený na pomyselnej, mriežke do ktorej jednotlivé elementy na obrazovke ukladáme. Keď sa obrazovka zmenší Bootstrap za pomoci JavaScriptu a CSS dokáže prispôsobiť aktuálne rozloženie mriežky prípadne skryť, alebo zobraziť konkrétne elementy[13]. Bootstrap prichádza už ako súčasť Blazor projektu. V prípade, že využívame na vývoj VisualStudio máme k dispozícií aj nápovedy vo forme dopĺňania textu3.3.

JQuery na druhej strane je malá jednoduchá knižnica, ktoré nám dovoľuje manipulovať s dokumentom, či už vykonávaním zmien nad jeho elementmi, alebo pridávaním nových elementov.

Existuje mnoho iných veľmi užitočných JavaSCriptových knižníc, ktoré sa nám môžu hodiť v budúcnosti. Stále sa však snažíme nechať väčšinu náročných algoritmov vykonávať vo WebAssembly pre čo najlepší výkon.



Obr. 3.3 Responzívny dizajn

3.3.8 Závislosti

Vitálnou súčasťou každého zložitejšieho softvéru sú závislosti. Jedná sa o dizajnovací postup, pri ktorom sa snažíme jednotlivé časti programu rozdeliť do logických celkov. Každá trieda by mala vykonávať len jednu konkrétnu úlohu. Ak na splnenie svojej úlohy potrebuje inú triedu vtedy sa jedná o takzvané vkladanie závislostí.

V prípade Blazoru si musíme uvedomiť ako je štrukturovaná hierarchia jednotlivých komponentov. Všetky komponenty sú vytvorené kaskádne počínajúc `App.razor`. Aby sme nemuseli do každého komponentu osobitne cez konštruktor pridávať závilosti potrebné pre jeho potomkov Blazor nám dáva k dispozicií vstavanú funkcionalitu, ktorá je realizovaná v `Program.cs`. `WebAssemblyHostBuilder` postupne naplníme jednotlivými singleton inštanciami a bez žiadnych obmedzení ich môžeme referencovať už priamo v kóde komponentov[14].

Takto rozdelený softvér má niekoľko výhod. Každý logický celok vieme v rámci jeho mandátu rozširovať bez toho aby to malo vplyv na chod programu. Tiež známe ako voľne spojene. Zároveň nám to uľahčuje testovanie. Jednotlivé závislosti vieme takzvane napadobniť vytvorením špecialnej prázdnej inštancie, ktorá len sa len tvári, že robí to čo robí. To vie byť veľmi užitočné napríklad, keď checeme testovať prácu s databázou, ale pochopiteľne nechceme počas testovania modifikovať dáta v reálnej databáze[15].

3.3.9 Singleton pattern

Signleton pattern sa často používa v prípade závislostí. Chceme docieliť aby danú funkcionalitu poskytovala len jedna inštancia triedy, ktorú budú navzájom zdielať všetky objekty.

Vytvorením súkromného konštruktora pre triedu vieme zabezpečiť jedinečnosť inštancie. Takto jediný spôsob ako môžeme inštanciu vytvárať je cez statický atribút reprezentujúci samotnú triedu. Tento atribút pri zavolaní skontroluje či inštancia existuje, alebo treba vytvoriť novú3.1.

V Blazore sa singleton triedy delia na tri typy singleton, scoped a transient. Trieda vo forme singleton sa vytvára pri prvotnom zapnutí aplikácie a existuje počas celého behu. Scoped vzniká pri každej požiadavke. Kým sa požiadavka spracuváva vždy máme k dispozicií tú istú konkrétnu inštanciu. Transient naopak vždy zabezpečí novú inštanciu pri každom volaní triedy čiže objekty takúto inštanciu nezdielajú.

Všetky takto vytvorené inštancie vieme ako závislosti sprístupniť celému projektu v súbore `Program.cs`[14].

3.3.10 Notifikácie

Výpis kódu 3.1 Singleton Trieda

```
public sealed class ColorPalette
{
    private static ColorPalette current = null;
    private ColorPalette() { }
    public static ColorPalette Current
    {
        get
        {
            if (current == null)
            {
                current = new ColorPalette();
            }
            return current;
        }
    }
}
```

3.4 Softvér na správu kontajnerov

V tejto sekcii si vysvetlíme prečo je výhodné využívať technológiu kontajnerov, ako táto technológia funguje a prečo sme ju využili v našej implementácií.

3.4.1 Kontajnery

S technológiou pripomínajúcou kontajnery sme sa už v tejto práci stretli. Spomínali sme virtuálny stroj JavaScriptu, ktorý pracuje v prehliadači a zabezpečuje vykonávanie JavaScriptových príkazov. Kontajnery majú však oveľa komplexnejšiu víziu. Ich cieľom je vytvoriť niekoľko priamo nezávislých súčasti systému, ktoré prerozdelíme na jednotlivé kontajnery. Tieto jednotlivé časti nesú so sebou len procesy a samotné závislosti bez ktorých by nemohli samostatne fungovať. Dokážu spolu komunikovať a zdieľať rôzne zdroje v rámci lokálnej siete, ktorú softvér na správu kontajnerov, v našom prípade Docker, vytvorí a poskytuje. Takto vytvorenému kontajneru je pomerne jednoduché poskytnúť rovnaké podmienky aké mal počas vývoja. To následne robí nasadenie, ale aj prípadne vylepšovanie veľmi jednoduché.

Kontainer, ako taký, je len inštancia vývojárom vytvoreného obrazu. Obraz si so sebou nesie všetky potrebné informácie v rátane kódu samotného a nevyhnutných knižníc pre beh programu. Vytvorený kontajner sám seba vidí ako osobitný systém s vlastným systémom súborov čo je výsledkom izolácie o ktorú sa kontajnery snažia. To nám umožňuje napríklad spustiť viacero inštancií toho istého softvéru naraz a prerozdeľovať požiadavky medzi jednotlivé kontajnery podľa záťaže. Inštancia využíva na svoje fungovanie priamo jadro operačného systému v našom prípade Linux Kernel. Pozitívnou skutočnosťou je úplná podpora Windows Subsystem pre Linux (WSL), ktorá nám dovolí spustiť Docker,

vytvárať obrazy a inštanciu aj priamo vo Windows systéme, ktorý odporúčam pre vývoj takejto aplikácie kvôli priamej kompatibilite s .NET[16].

3.4.2 Nasadenie

Aby sme mohli našu aplikáciu nasadiť, stačí mať funkčný softvér na správu kontajnerov na hosťovacom stroji a skompilovaný obraz z jednotlivých samostatných častí našej aplikácie. Tieto časti môžu napríklad byť API server, aplikácia a databáza. Zásluhou rozkúskovanému softvéru vieme nasadiť jednotlivé časti do kontajnerov a umožniť im komunikáciu v rámci ich vlastnej siete. To nám prináša niekoľko výhod ako napríklad ľahké vylepšenie konkrétnej časti tým, že len vytvoríme nový kontajner bez toho aby si zvyšok systému uvedomil, že došlo ku zmene, alebo zaistenie plnej funkcionality, ktorú si môžeme otestovať ešte počas vývoja na svojom lokálnom softvéri na správu kontajnerov.

3.4.3 Docker

Súčasťou dodaného softvéru, ktorý je výsledkom tejto práce je obraz vo forme Docker-image. Dôvod prečo som zvolil Docker ako manažér našich kontajnerov je primárne priama podpora integrovaného vývojového prostredia Visual Studio, ktoré vie vytvoriť Docker-file slúžiaci ako šablóna pre vytvorenie obrazu z našej aplikácie.

Zároveň ľahká inštalácia či už na platforme windows vďaka spomínanému WSL, alebo na Linuxe, ktorý s najväčšou pravdepodobnosťou by bol aj systém na ktorom by ste aplikáciu chceli eventuálne hosťovať. To mi umožňuje dosiahnuť celý proces vývoja až po samotné dodanie aplikácie s istotou, že dodaná aplikácia nebude mať problém fungovať v prostredí v ktorom ju klient nasadí[16].

3.4.4 Kubernetes

Príchod Dockera a jemu podobnému softvéru priniesol novú myšlienku ako realizovať rôzne komplikované softvérové riešenia. Štandardný systém vývoja aplikácií spočíval v jednej veľkej, ktorá mala na starosti všetku funkcionalitu. Kontajnery nám umožnili rozložiť softvér na viacero samostatných modulov, ktoré vieme ľahko nasadiť. To nám dovoľuje podľa potreby distribuovať záťaž medzi rôzne inštancie našej aplikácie. Takéto softvérové riešenie prináša napríklad firma Google a nazýva sa Kubernetes.

Predstavme si, že máme časť implementácie zodpovednej za prijímanie emailov a iná naopak ich spracováva. V jeden moment nám príde veľké množstvo dát, ktoré nestíhame prijímať. Vtedy Kubernetes rozpozná vyššiu záťaž na konkrétnom module a z obrazu vytvorí ďalšie inštancie, ktoré nám pomôžu ich prijať. Keď

nápor emailov opadne Kubernetes tieto inštancie rozloží a naopak zdvihne viacej inštancií modulu, ktorý pomôže emaily zase spracovať.

Aby Kubernetes mal nad celou hierarchiou našej aplikácie kontrolu využíva takzvané reverzné proxy. Reverzné proxy je server, ktorý slúži ako vstupná brána do lokálnej siete aplikácie. Všetky prichádzajúce požiadavky sú posielané priamo na tento server a ten sa následne rozhoduje kam ďalej ich treba poslať na spracovanie. Takto zapuzdrený softvér disponuje viacerými výhodami ako je bezpečnosť, spoľahlivosť a práve rozdistribuovanie práce do špecifických odvetví systému[17].

3.5 Databáza

Nasledujúce sekcia vysvetlí pojem databáza a odôvodní výber databázy pre našu aplikáciu.

3.5.1 SQL vs NoSQL databázy

Pre chod našej aplikácie, ale aj ľubovolného informačného systému potrebujeme perzistentné médium, kde budeme odkladať dôležité informácie. Softvér, ktorý nám toto umožní sa nazýva databáza. Databáza je štruktúra informácií, ktorá nám pomáha s dátami manipulovať a organizovať ich. Dnes sa najčastejšie stretávame s dvoma typmi databáz.

Relačná databáza organizuje dáta do takzvaných entít na základne vzťahov medzi nimi. Jednotlivé entity sú reprezentované formou tabuliek, kde každý stĺpec prezentuje jeden atribút a riadok jeden zápis. Každá takáto entita musí obsahovať jeden unikátny identifikátor tiež známy ako primárny kľúč. Primárny kľúč vieme potom použiť v nasledovnej tabuľke aby sme povedali, že konkrétna entita patrí inej entite. Vtedy má označenie cudzí kľúč. Relačná databáza má svoje dáta uložené na disku a pristupuje k nim na to určený server, ktorý s nimi vie manipulovať. Serveru môžeme dávať rôzne požiadavky pomocou programovacieho jazyka nazývaného Structured Query Language (SQL), ktorý je dnes už zaužívaným štandardom. Je to vysokoúrovňový programovací jazyk, čo nám umožňuje jednoduchými príkazmi vykonať zložité operácie, ktoré by za bežných okolností procedurálnym jazykom zabrali aj stovky riadkov kódu. Výhodou relačnej databázy je množstvo dát s ktorými vie pracovať a to do akej miery vieme špecifikovať jednotlivé vzťahy medzi nimi. Fixná štruktúra nám pomáha ku stabilite systému. Tým pádom vieme čo presne vchádza do databázy aj to čo z nej môžeme očakávať naspäť. S tým však prichádza rovnako aj nevýhoda vo forme komplikovaného návrhu a malej flexibility. V momente, keď v databáze máme už produkčné dáta je veľmi ťažké robiť zmeny bez toho aby sme o ne prišli, alebo ich museli migrovať. Relačnú databázu väčšinou rozširujem vertikálne to znamená pridaním fyzických zdrojov do existujúceho stroja miesto pridávania ďalších strojov.

Alternatívne sa môžeme stretnúť s NoSQL databázami, ktoré dáta odkladajú

do špecifických štruktúr čím získavajú rôzne benefity. Dokumentová databáza využíva miesto tabuliek JavaScript Object Notation (JSON) formát v binárnej podobe. JSON umožňuje zapísať jednotlivé atribúty triedy aj s ich hodnotami. Ďalšie spôsoby odkládania dát sú aj napríklad formou slovníkov, grafov, alebo dynamických tabuliek. Výhodou takejto databázy je predovšetkým veľká flexibilita a za istých okolností vysoká efektivita. Dokáže si usporiadávať súbory podľa toho ako často prichádzajú na ne požiadavky a je horizontálne rozšíriteľná. To znamená, že môžeme ľahko pridať ďalší stroj, ktorý bude disponovať ďalším miestom a sám si ho môže riadiť. Táto databáza však v našej aplikácií nie je použiteľná, kvôli chýbajúcej kompatibilite s Entity Frameworkom o ktorom sa dočítate nasledujúcej sekcií[18].

3.5.2 Entity Framework

Získanie informácií z databázy obnáša niekoľko krokov. Musíme inicializovať spojenie s databázou, poslať potrebné SQL požiadavky a transformovať dáta z databázy do objektov. Aby sa mohli programátori sústrediť na dôležitejšiu vyššiu úroveň abstrakcie vznikla nová programovacia technika s názvom Object-relational mapping skrátene ORM. ORM poskytuje spôsob akým konvertovať dáta medzi typmi v databáze a objektami v kóde.

.NET nám formou princípu ORM poskytuje takzvaný Entity Frameworku (EF)[19]. EF sa skladá z niekoľkých častí. Komunikáciu s ním nám umožňuje API, ktorá zapuzdruje tri hlavné časti. Konceptuálny model reprezentujúci naše triedy, storage model postavený na základe použitého databázové schéma a mapovanie medzi týmito dvoma modelmi. Mapovanie obsahuje práve informácie ako transformovať správne naše triedy do ekvivalentných entít v databáze a naopak. Inštancie našich tried sa držia v takzvanom databázovom kontexte. Táto trieda nám umožňuje povedať, ktoré triedy chceme aby boli reprezentované v databáze a neskôr nám umožňuje nad nimi vykonávať potrebné operácie. Všetky vykonané zmeny sa dejú lokálne na úrovni kontextu a k uloženiu do databázy prebehne až po zavolaní metódy `SaveChangesAsync`, ktorá kontext implementuje. Naopak, keď chceme data získať, kontext nám poskytuje radu podporných metód a zároveň nám umožňuje používať Language-Integračné Query LINQ na miesto písania SQL požiadaviek.

Veľkou výhodou EF je aj jeho podpora na rôznych platformách nie len Windows a zároveň sa dá aplikovať na rôzne databázy tým, že vývojárom umožňuje implementovať databázový poskytovateľa, ktorý špecifikuje predovšetkým akým spôsobom sa transformujú dátové typy C# tried na typy danej databázy. Funkcionalita prichádza so vstavaným riadením transakcií a dočasným odkladaním opakovaných požiadaviek do lokálnej pamäte, aby sa zbytočne nevykonávali rovnaké operácie nad databázou.

3.5.3 Výber databázy

EF podporuje radu rôznych databáz, ktoré implementujú vyššie spomínaného databázového poskytovateľa. Najznámejšie z nich sú napríklad SQL Server, MySQL, SQLite a PostgreSQL no každý prináša svoje výhody.

Prirodzene by sme vybrali SQL Server s jeho vstavanou funkcionalitou priamo v EF, nie je pre nás úplne ideálny primárne z dôvodu, že sa jedná o platený softvér. Poskytuje aj verziu SQL Server Express tá je však určená predovšetkým pre jednoduchý a malý softvér čo vzhľadom k tomu, že chceme poskytnúť základ pre ľubovolný informačný systém nie je pre nás riešením.

MySQL je momentálne open source projekt, ktorý v roku 2010 kúpila firma Oracle, takže majú v rukách všetky zmeny, ktoré v budúcnosti prídu do MySQL v vrátane jeho licencie. Nemáme záruku, že nová verzia už nepríde pod zmenenou licenciou a nedonúti nás eventuálne prejsť na konkurenciu, čo bude obnášať veľa komplikovaných migrácií a zmenu v nasadení[20].

Tu prichádza na scénu PostgreSQL. Tiež open source projekt, ktorý je komunitným dielom a teda nemá vlastníka, ktorý by mohol jednohlasne prísť a povedať, že odteraz sa bude platiť[21]. Vďaka tomu môžeme s kľudným svedomím vyvíjať náš softvér s víziou do budúcna. Zároveň to pre Postgres znamená veľké množstvo vývojárov, ktorý sa na jeho rozsahu podieľajú. Dnes už predstavuje plnohodnotné riešenie pre celú škálu rôznych softvérov s mnohými funkcionami, ktoré konkurenčné databázy zadarmo neposkytujú takže nám bude fungovať aj napríklad Kubernetes.

SQLite[22] predstavuje svoju vlastnú kategóriu. Nie je reprezentovaný ako klasický databázový server, s ktorým komunikujeme, keď potrebujeme dáta, ale formou súborov. Toto je dosiahnuté tým, že jednotlivé SQL požiadavky sa vďaka SQLite knižniciam prekladajú na niekoľko malých operácií operačného systému. Nevytvára sa žiadny nový proces, všetko je spracovávané so zdrojmi, ktorými aplikácia disponuje. Takýto prístup nám poskytuje hneď niekoľko výhod avšak rozhodne nie je určený pre žiadne veľké a náročné aplikácie. Môžeme sa zamyslieť nad spôsobom ako ho využiť napríklad ako dočasné úložisko pre dáta v dobe, keď naša aplikácia je offline a pracuje len so svojimi lokálnymi dátami.

3.6 Alternatívne riešenia

Webové aplikácie sa stali populárnymi ďaleko skôr ako Blazor WebAssembly vzniklo. V tejto sekcií sa pozrieme na alternatívne riešenia, ktorými by sme vedeli dosiahnuť podobných výsledkov.

3.6.1 Google Web Toolkit

Google Web Toolkit (GWT) bol jeden z prvých pokusov vytvárať interaktívne webové aplikácie. Tento framework využíva jazyk Java, ktorý neskôr kompiluje

rovno do čistého JavaScriptu. Dnes sa už GWT považuje za zastarané a jeho vývoj posledné roky veľmi klesol najmä kvôli malému záujmu vývojárov. Napriek tomu stále existuje niekoľko rôznych softvérov, ktoré sú stále udržiavané a ďalej vyvíjané v GWT[2].

3.6.2 Angular

Angular[23] je front-endový framework, vyrobený firmou Google, určený primárne na vytváranie užívateľského rozhrania. Podobne ako Blazor funguje na princípe komponentov, ktoré sa dokážu počas života aplikácie pridávať, odoberať a rôzne meniť. Tento framework využíva JavaScript pre svoje účely. Pôvodne existoval takzvaný Angular JS, ktorý sa vyvíjal priamo v JavaScripte. Neskôr bol nahradený Angular X, ktorý JavaScript vymenil za TypeScript.

TypeScript je programovací jazyk, ktorý sa transkompiluje na JavaScript. Jeho úlohou je uľahčiť vývoj JavaScriptových aplikácií. Dovolí priamo vpisovať čistý JavaScriptový kód a prináša so sebou rôzne funkcionality ako podpora statických typov, objektov a uľahčenie ladenia softvéru počas vývoja.

Hlavnou výhodou vývoja v Angulare je jeho fixná štruktúra, ktorá je efektívna pri vytváraní jednotlivých komponentov a implementovanie funkcionality do nich. Rieši za nás veľa problémov ako napríklad dodávanie závislostí pre jednotlivé časti aplikácie. Obsahuje aj množstvo užitočných knižníc, ktoré nám uľahčia vývoj.

Dnes je Angular jeden z najpopulárnejších technológií na vyvíjanie webových aplikácií a vytvára skvelú alternatívu pre Blazor.

3.6.3 React

React je obzvlášť populárna[24] JavaScript knižnica, ktorá sa snaží o podobný prístup k užívateľskému rozhraniu ako Angular, či Blazor formou komponentov. Pre porovnanie s Angularom sa však jedná len o knižnicu to znamená, že neprichádza s celým radom rôznych funkcionalít na výrazne urýchlenie vývoja, ale so spôsobom ako priamo v JavaScripte vytvárať komponenty a usmerňovať ich dáta v reálnom čase. Existuje jeho obdoba React Native, ktorá sa však využíva ako nástroj na tvorbu nátivných aplikácií pre rôzne platformy.

Na získanie funkcionality podobnej Angularu, môžeme využiť NextJS framework, ktorý vytvára React aplikácie. Narozdiel od klasického Reactu, webstránka je vygenerovaná rovno na serveri a klient dostáva už hotovú stránku.

Kapitola 4
Praktická časť

Už sme sa zoznámili s potrebnými pojmami a technológiami použitými v tejto práci. Teraz si ukážeme ako sa dá toto všetko využiť na tvorbu jednoduchého informačného systému a s akými problémami sme sa počas vývoja stretli.

4.1 Validátor regulárnych výrazov

4.1.1 Vytváranie stromovej štruktúry

Na implementáciu validátoru sme využili pár základných princípov Blazoru a časti .NET knižnice nazývanej Regex. Tu prichádzame k prvému problému. Výstup metódy `Match` je zoznam zhôd zoradený zľava doprava viz obrázok 4.1 avšak my ich potrebujeme zoradiť do stromovej štruktúry pre čitateľnejšie zobrazenie.

Takýto zoznam prináša jednu veľkú nevýhodu a tou je, že nevieme povedať či sa daná zhoda nachádza v tej naľavo od nej, alebo nie. Môžeme pri každom prvku zobrať predchádzajúci a zistiť či sa v jeho reťazci nenachádza podreťazec aktuálnej zhody. To však zlyhá v prípade, že dve menšie sa nachádzajú v jednej veľkej, ale nie v sebe samej navzájom. Tým pádom tretia zhoda v poradí nebude súčasťou tej predošlej, ale tej ešte predtým. Teda môžeme implementovať algoritmus tak, že v prípade, že sa nenachádza v susediacej zhode pokúsime sa zistiť či nie je súčasťou predošlej. Hneď si však môžeme všimnúť problém pri vstupe v ktorom dve susediace budú zdieľať rovnaký reťazec avšak nebudú zapuzdrené v rovnakej zhode nad nimi. Riešenie vyžaduje použitie indikátoru či sa snažíme znovu zaradiť zhodu s rovnakým reťazcom na miesto, kde sa už nachádza predošlá. Takýto algoritmus začína výrazne naberať na komplexitivite, aj keď si odmyslíme ako náročné je prechádzať reťazec a hľadať v ňom podreťazce. Ak by sme chceli niečo podobné aplikovať na veľké množstvo textu náš program by trval neprístupne dlho a zbytočne by vyťažoval klientom, ktorý nemusí nevyhnutne byť výkonný stroj.

Naše riešenie prichádza z jednej dôležitej informácie, ktorú si každá zhoda nesie vo vstupnom liste pri vytváraní so sebou. Tou je index na ktorom začína.

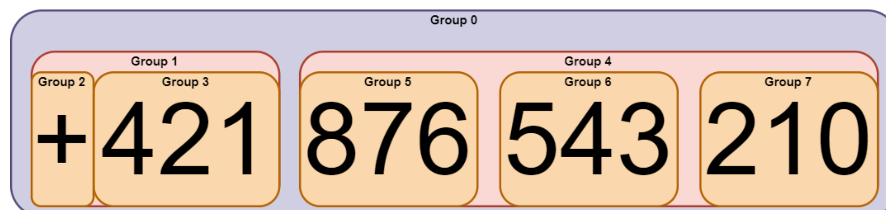
Ak si zoberieme počiatočný index vieme ľahko povedať či sa začína v rozhraní indexu predchádzajúcej zhody a dĺžky reťazca, ktorý v sebe má. Najskôr si vytvoríme prvú triedu, ktorá si v sebe bude držať všetky naše výsledky a následne rekurzívne aplikujeme funkciu `AddGroup` viz kód 4.1, ktorá preiteruje svojich potomkov. Ak nastane prípad, že sa zhoda, ktorú sa snažíme pridať na základe indexu, už nachádza v jednom z potomkov rekurzívne zopakujeme rovnaký proces. V konečnom dôsledku vždy nastane stav, kedy už nevieme nájsť dieťa do ktorého by sme mohli našu novú zhodu zaradiť a tak ju pridáme ako nový list do stromovej štruktúry na posledný prvok v ktorom bola funkcia `AddGroup` zavolaná.

Výpis kódu 4.1 Funkcia AddGroup

```
public void AddGroup(ExpressionGroup newGroup, int depth = 0)
{
  this.Depth = depth;
  bool createNewGroup = true;

  foreach(ExpressionGroup childGroup in ExpressionGroups)
  {
    if(childGroup.ContainsGroup(newGroup))
    {
      childGroup.AddGroup(newGroup, this.Depth + 1);
      createNewGroup = false;
      break;
    }
  }

  if(createNewGroup)
  {
    newGroup.Depth = this.Depth + 1;
    ExpressionGroups.Add(newGroup);
  }
}
```



Obr. 4.1 Reprezentácia výstupu zhôd z metódy Match

4.1.2 Vykreslovanie

Primárny dôvod prečo sme potrebovali stromovú štruktúru zo zhôd je ich vykreslenie. Našim cieľom je zobraziť text s farebným vyznačením jednotlivých elementov aby si užívateľ vedel lepšie predstaviť ako sa jeho regulárny výraz aplikuje na text a to vo forme zvýraznenej tabuľky a stromu.

Stromová štruktúra prináša komplikáciu, keď ju chceme využiť priamo v dokumente. To, že strom musíme prechádzať rekurzívne nám v podstate znemožňuje iterovať po jednotlivých prvkoch a teda nevieme výpis štruktúry priamo zapuzdriť v HTML tagoch. Takéto vypisovanie dát do HTML Blazor realizuje najčasnejšie pomocou iterácií a každý prvok osobitne vypíše do dokumentu.

Aby sme obišli tento problém potrebujeme dosiahnuť podobný formát aký sme pôvodne dostali na vstupe pri vytváraní stromu, avšak v tomto prípade už realizovaný našimi objektami. Nad koreňom stromu zavoláme rekurzívne funkciu `ToList`, ktorá si posiela referenciu na list inštancií ako parameter. Tento list sa inicializuje pri koreni, teda zvyšné inštancie doň len pridajú samých seba a rekurzívne zavolajú túto funkciu nad potomkom.

Keď už máme k dispozící list vieme ho ľahko preiterovať a tak naplniť tabuľku s jeho hodnotami. Ostáva vyriešiť jednotlivé farby, ktoré však v rámci iterácie môžeme aplikovať len na celý prvok nie na potomkov ukrytých vo vnútri prvku. V prípade, že by sme sa snažili prvok postupne napĺňať po jednotlivých zhodách a tie farebne identifikovať zabráni nám v tom kompilácia HTML, ktorá nie je schopná v rámci iterácie začať s otvoreným tagom, kým naplníme vnútro a potom ho eventuálne ukončiť.

Jediný spôsob ako môžeme prvky vypísať so špecifickým HTML kódom, ktorý by ich obalil je nechať každý prvok vypísať samého seba v rekurzií. Začneme tým, že sa rekurzívne dostaneme ku listom stromu a po ceste späť ich jednotlivo vypíšeme pomocou funkcie `ToHtml` viz kód 4.2. Takto dostávame súvislý reťazec obsahujúci už zaobalené prvky stromu v HTML kóde, ktorý im pridáva farbu. Jediné čo nám ostáva je tento reťazec vpísať do dokumentu a aby sme zabezpečili, že sa nám nezobrazí len ako text, ale stane sa súčasťou dokumentu musíme tento reťazec pretransformovať do dátoveho typu s názvom MarkupString.

Vykreslenie stromovej štruktúre ja ďaleko viac náročné. Prvotne nevieme povedať koľko jednotlivých prvkov bude mať každá jedna úroveň stromu čo komplikuje horizontálne centrovanie prvkov o úroveň vyššie. Zároveň tým, že dokument je štrukturovaný do jednotlivých blokov nie je za bežných okolností možné kresliť čiary, ktoré by jednotlivé prvky stromu spájali a dávali mu štruktúru.

Strom teda vyzobrazíme formou pyramídy aby sme ukázali koľko a akých prvkov sa nachádza na jednotlivých vrstvách. Paradoxne je pyramída obrátená, lebo jednotlivé zhody vždy obsahujú menej textu ako ich rodičia. Aby sme vedeli vypísať každú úroveň jednotlivo, pridáme do nášho objektu, ktorý reprezentuje zhodu, hĺbku v ktorej sa nachádza. To vieme zabezpečiť už počas inicialneho vytvárania stromovej štruktúry tým, že si pri každom zanorení rekurzie posu-

nieme ako parameter aj predošlú hĺbku, ktorú následne inkrementujeme. Na konci pridáme metódu `ToListByDepth`, ktorá s využitím už existujúcej metódy `ToList` preiteruje zoznam našich prvkov a vypíše tie, ktoré sa nachádzajú v danej hĺbke.

Poslednou výzvou je zabezpečiť farby pre naše zobrazenie. Vyberanie úplne náhodných farieb by mohlo spôsobiť neprehľadnosť a zároveň chceme zaručiť aby sa farby opakovali v rozumnom cykle a dali sa bez veľkej námahy vymieňať. Preto implementujeme statickú triedu `ColorPalette`, ktorou jedinou úlohou je udržiavať zoznam farieb a funkcia, ktorá tie farby postupne dáva k dispozícií. Cyklovanie farby zabezpečíme jednoduchým indexom, ktorý sa pri každom zavolaní statickej metódy `GetColor` inkrementuje a vďaka modulu ho vieme obmedziť na hodnoty v rámci rozsahu nášho zoznamu farieb. Táto trieda musí byť implementovaná ako singleton, keďže ju chceme zavolať na rôznych nezávislých miestach v kóde počas obaľovania prvkov do HTML tagov pre pridanie farby.

Výpis kódu 4.2 Funkcia ToHtml

```
public string ToHtml()
{
  string html = this.Value;
  int offset = 0;

  foreach(ExpressionGroup group in ExpressionGroups)
  {
    string snippet = group.ToHtml();

    html = html.Remove(group.Index + offset, group.Value.Length);
    html = html.Insert(group.Index + offset, snippet);

    offset += snippet.Length - group.Value.Length;
  }

  return @$"
        <div class='node' style='background-color: {color}'>
          {html}
        </div>
      ";
}
```

4.1.3 Notifikácie

Dôležitou súčasťou našej aplikácie je odovzdať užívateľovi informáciou o výsledku vykonanej operácie. Takáto funkcionalita prichádza s niekoľkými výzvami. Potrebujeme zabezpečiť aby sa zobrazovala pri každej stránke bez ohľadu na to, ktorú časť aplikácie užívateľ používa. Zároveň je potrebné aby aj ľubovolná časť implementácie mala možnosť oboznámiť užívateľa s dosiahnutým výsledkom. Napokon

chceme zabezpečiť modularitu pre jednoduché pridávanie a prípadne úpravy jednotlivých typov notifikácií. Naše notifikácie sa teda budú skladať z troch častí. Blazor komponent zodpovedný za zobrazovanie notifikácie v dokumente, servisnej vrstvy, ktorá bude komunikovať medzi časťami aplikácie a komponentom a objekt, ktorý notifikáciu bude reprezentovať.

Blazor výsledný DOM skladá často z mnohých vrstiev. Každá osobitná strana aplikácie sa vždy vyobrazí do triedy MainLayout. Táto trieda obsahuje hlavne väčšinou statické nemenné prvky ako napríklad menu, header, alebo footer. Tu umiestnime náš `NotificationComponent` aby bez ohľadu na to v ktorej časti sa nachádzame bol dostupný a viditeľný.

`NotificationComponent` potom ľahko pomocou HTML tagov a CSS vizuálne zobrazíme. Aby sme umožnili variabilitu v notifikáciach využijeme C# markup, ktorý vie priamo vpisovať C# kód do dokumentu. Komponetu tak môžeme poskytnúť vytvorenú inštanciu notifikácia nastavenú presne podľa potreby. Prvá komplikácia nastáva, keď chceme aby sa zobrazená notifikácia rozložila a urobila priestor pre ďalšiu. Vizuálnu časť vieme vyriešiť pomocou prechodov v CSS, kde vieme nastaviť aby zobrazený komponent napodobnil animáciu pri ktorej postupne prejde z priehľadného na nepriehľadný a naspäť. Následkom je skrytie objektu nie fyzické odstránenie z dokumentu. Aby sme objekt vedeli automaticky odstrániť potrebujeme asynchrónnu metódu, ktorá vytvorí nový proces čakajúci po dobu, kedy animácia prebieha. Po jej skončení inštanciu notifikácie natrvalo odstráni priamo z dokumentu. Za predpoklad, že užívateľovi zobrazíme len jednu notifikáciu v danej chvíli tento prístup funguje perfektne. Problém nastáva, keď užívateľ vykoná viacero operácií v krátkom čase a potrebujeme zobraziť ďalšiu notifikáciu v dobe, keď sa predošlá ešte neskončila. Aby sme hĺbku tohto problému lepšie pochopili, predstavme si špecifický prípad, kedy máme notifikáciu už zobrazenú na obrazovke a prichádza ďalšia. Ako sme si už vyššie vysvetlili Blazor pri každej zmene nájde rozdiely medzi DOM a tie potom naspäť spracované posiela do aplikácie. Ak je notifikácia už zobrazená, rozdiel, ktorý sa aplikuje bude iba v texte, prípadne farbe. To spôsobí, že prvá notifikácia začne animáciu, v tom sa text zmení na druhú notifikácie a animácia skončí tak ako by skončila pôvodná. Za istých okolností nebolo ani možné prečítať akú notifikáciu sme dostali. Tento problém vyriešime tak, že do hlavného `NotificationComponen` pridáme miesto jednej notifikácie zoznam notifikácií, do ktorého ich postupne podľa potreby pridávame a automaticky ich asynchrónna metóda odoberá po vypršaní ich platnosti. Ostáva nám len pripomenúť, že treba Blazor oboznámiť o zmene v komponente a to zavolaním metódy `StateHasChanged` viz kód 4.3, ktorá naštartuje životný cyklus komponentu.

Keďže sme `NotificationComponent` vložili do koreňa celého dokumentu potrebujeme spôsob ako k nemu umožniť prístup. Preto si vytvoríme servisnú vrstvu s názvom NotificationServise. Bude plniť jednoduchý účel a tým je obsahovať asynchronnú úlohu, ktorú dokáže jej inštancia zavolať. Pri prvotnom inicializovaní NotificationComponentu využijeme jednu z metód životného cyklu komponentov a tou je OnInitialized, kde nastavíme servise ako delegáta metódu komponentu,

ktorá zabezpečuje pridávanie notifikácií do zoznamu. Posledným nevyhnutným krokom je zabezpečiť aby celá aplikácia používala práve jednu inštanciu servisu, ktorá musí byť dostupná v rámci celého projektu. Keďže finálny dokument zobrazený v prehliadači je v podstate kaskáda rôznych Blazor komponentov vytvorenie takejto inštancie a následne dodanie do každého komponentu by výrazne komplikovalo celý vývoj. Preto Blazor poskytuje vlastné riešenie. V súbore `Program.cs` sa nastavujú rôzne servisy, kde máme k dispozícií hneď niekoľko verzií singleton inštancií, ktoré budú dostupné v našich komponentoch. My využijeme pre tento prípad funkciu `AddSingleton`, ktorá poskytuje singleton inštanciu v rámci celého života aplikácie.

Výpis kódu 4.3 Implementácia zobrazovania a následného mazania notifikácií

```
public void AddNotification(Notification notification)
{
  lock (notificationLock)
  {
    this.notifications.Add(notification);

    this.StateHasChanged();
  }

  Task.Run(async () =>
  {
    await Task.Delay(1900);
    RemoveNotification(notification);
  });
}

private void RemoveNotification(Notification notification)
{
  lock (notificationLock)
  {
    this.notifications.Remove(notification);
    this.StateHasChanged();
  }
}
```

4.1.4 Servisná vrstva

Pre prístup do databázy používame ASP.NET Core na ktoré sa chceme pripojiť cez HTTP požiadavky. Podobne ako s `NotificationServis` potrebujeme servisnú vrstvu dostupnú v celej aplikácií. Najskôr vytvoríme rozhranie, ktoré usmerní každú triedu, pristupujúcu k databáze, implementovať všetky základné CRUD operácie. Tu narážame na polsedný výrazný problém v implementácií klienta.

Každá trieda potrebuje na svoj chod inštanciu `HttpClient`, ktorá umožní komunikáciu so serverom. Inštancia je od založenia projektu dostupná pre každý komponent vďaka funkcií `AddHttpClient` volaná v Program.cs. Táto funkcia však nevracia inštanciu, ale priamo ju pridáva medzi servisy aplikácie takže nie sme schopný ju využiť v našich osobitných súboroch tvoriacich servisnú vrstvu. Tento problém musíme obísť vytvorením vlastnej inštancie, ktorú následne spravíme dostupnou pre triedy servisnej vrstvy formou parametru v konštruktore. Následne môžeme tieto inštancie pridať aj medzi servisnú vrstvu Blazoru a tým ich sprístupníme naprieč celou aplikáciou.

4.2 Server

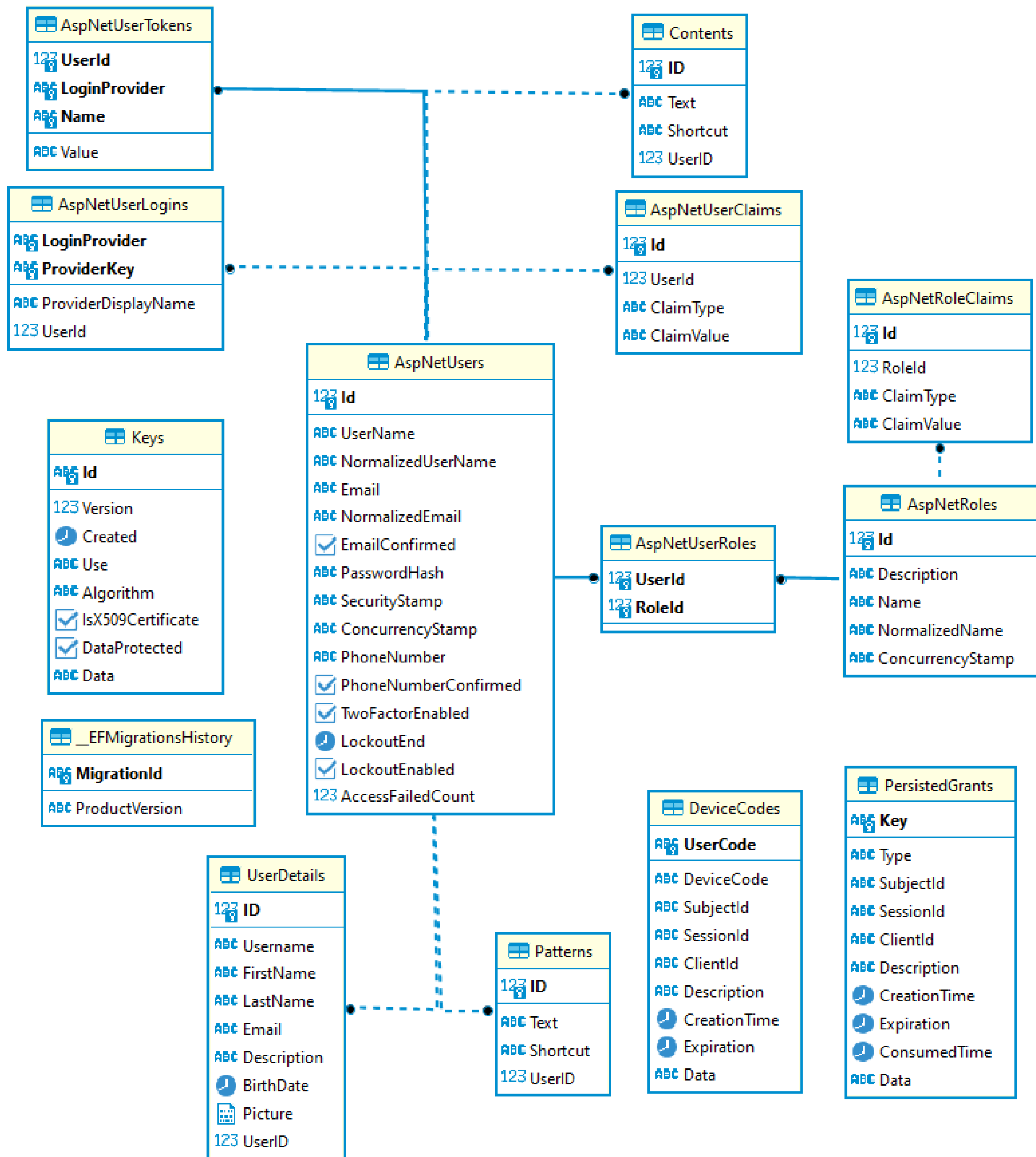
4.2.1 Identity Server

Pri vytváraní projektu máme možnosť zvoliť zabudovanú autorizáciu. Táto autorizácia je realizovana formou vstavaných knižníc s názvom Identity Server. Taktiež nám poskytuje klasickú autorizáciu užívateľa s registráciou čo nám ušetrí veľa času pri implementácií. Táto implementácia je však po vizuálnej stránke veľmi minimalistická takže je potrebné ju prepísať. Visual Studio nám svojou funkciu nazývanou scaffolding sprístupní kópie týchto súborov, ktoré si programátor môže následne podľa svojej predstavy upraviť.

Tu narážame na doteraz pre mňa záhadný problém pri ktorom na konci procesu scaffolding vyhlási, že nie je možné stiahnuť potrebné závislosti pre dokončenie operácie. V našej implementácií bolo potrebné vyčistiť NuGet cache a zavolať tú istú funkciu cez Pcakage Manager Console. Po niekoľkých neúspešných pokusoch sa nám podarilo získať kópie súborov, ktoré reprezentujú prihlasovacie a registračné okno.

Identity Server má rovnako predvytvorené triedy, ktoré reprezentujú v databáze užívateľa, role a mnoho iných4.2. Vďaka Entity Frameworku sú tieto triedy odzrkadlené v databáze a umožňujú perzistentné manipulovanie s dátami. Triedy si však v tomto prípade vieme ľahko podediť a implementovať zmeny pre naše potreby. Problém nastáva v predvolenej implementácií, kde obe tieto entity využívajú ako svoje identifikátory reťazce. Ak chceme tieto reťazce zameniť za čísla, ako je bežným zvykom čaká nás niekoľko nevyhnutných úprav. Prvotne si musíme uvedomiť, že ApplicationDbContext, ktorý si drží informácie za chodu aplikácie predtým ako sú perzistentne uložené v databáze, je implementovaný len ako demonštrácia a musíme si vytvoriť vlastný. Vytvoríme si abstraktnú triedu DbContext, ktorá bude dediť z IdentityDbContext a IPersistedGrantDbContext. Pre zmenu implementácie úmyselne nepoužijeme priamo ApplicationDbContext, kvôli komplikáciam pri migrácií, kde je v takom prípade nevyhnutné pripisovať generické parametre. Následné dedenie DbContextu nám umožní špecifikovať triedy reprezentujúce užívateľa, úlohu a dátový typ pre ich identifikátory. Ostáva nám

vytvoriť novú migráciu a máme užívateľa s číselným identifikátorom.



Obr. 4.2 UML diagram databázy

4.2.2 Repository-Service Pattern

Na prijímanie požiadaviek a mapovanie ich na jednotlivé časti implementácie ASP.NET Core používa takzvané kontrolery. Tieto kontrolery reagujú na špecifický formát URL a podľa typu požiadavku prípadne parametra spustia požadovanú časť implementácie. Pre vytvorenie kontroleru môžeme bez problémov využiť vyššie

spomínaný scaffolding, tento krát bez problémov. Scaffolding nám dá na výber pre akú triedu chceme kontroler vytvoriť a ktorý databázový kontext má byť použitý na komunikáciu s databázou. Takto vytvorený kontroler implementuje radu základných CRUD metód, ktoré priamo spracujú a komunikujú s databázou. Takýto prístup však výrazne komplikuje hneď niekoľko vecí. Musíme si implementovať každý kontroler osobitne, nemáme servisnú vrstvu, ktorá by mohla vykonávať biznis logiku a všetky metódy sú priamo napojené na prácu s databázou. Výrazne tým prichádzame o modularitu a komplexnejšie spracovanie požiadaviek by úplne zahltilo kontroler a komplikovalo tak vývoj.

My preto využijeme populárny Repository-Service pattern. Vytvorenie servisnej vrstvy nám umožní pre každú triedu reprezentujúcu entitu v databáze implementovať dodatočné spracovanie dát. Na druhej strane repository implementuje základné CRUD operácie priamo nad databázou čím zabránime zbytočnému opakovaniu kódu v servisách a odseparujeme tak biznis logiku od databázy ako takej. Takáto rozdelená implementácia do troch vrstiev je veľmi flexibilná voči zmene databázy, alebo rôznym rozšíreniam do budúcnosti.

4.2.3 Testovanie

Testovanie je dôležitou súčasťou vývoja softvéru. Bežne sa aplikácie preverujú pomocou Unit testingu, ale vzhľadom k tomu, že je naša implementácia veľmi jednoduchá a ťažko by sa rozdeľovala na jednotlivé testovateľné časti, využili sme systémové integračné testovanie.

Vytvorili sme niekoľko rôznych prípadov použitia nášho softvéru, ako napríklad vytvorenie užívateľského účtu, alebo zmenu dát užívateľa. Takéto testy vyskúšajú funkcionalitu na celej škále jednotlivých modulov od prístupu do databázy až po správne zobrazenie užívateľského rozhrania.

Ak by sme však softvér ďalej rozširovali je určite nevyhnutné pridať Unit testy. Ich úlohou je testovať jednotlivé moduly prípadne algoritmy. Často sa zapínajú priamo pri budovaní projektu a sú nám dostupné už priamo v .NET.

4.2.4 Nasadenie

Softvér máme pripravený a rozdelený tak, aby mohol fungovať v osobitných kontajneroch už počas vývoja. Visual Studio poskytuje funkcionalitu pre nasadenie ako docker-image, avšak nevytvára priamo kontajner do našej lokálnej inštancie Dockera, ale snaží sa vytvorený docker-image nahrať do nášho súkromného Docker účtu. Za bežných okolností by toto správanie bolo vyhovujúce, keď by sme softvér chceli nasadiť u klienta na ich stroji. My však potrebujeme nájsť spôsob akým dokážeme skompilovať náš projekt do docker-image a lokálne ho už spustiť vo finálnej verzií zarovno s databázou.

Využijeme Visual Studio na vytvorenie takzvaného Dockerfilu, ktorý bude slúžiť

ako návod s inštrukciami vysvetlujúcimi ako sa má kompilovať celý náš projekt do obrazu. Aby sme teda mohli vytvoriť obraz vytvoríme si spustiteľný súbor, do ktorého zadáme potrebné konzolové príkazy pre Docker. Je vhodné v takomto súbore zahrnúť vymazanie predošlého kontajneru a vytvorenie novej inštancie čím dosiahneme ľahké vylepšenie našeho projektu, keď vytvoríme novú verziu.

4.2.5 Dokumentácia a inštalácia

Dokumentácia bola vytvorená pomocou programu Doxygen[25]. Program je dostupný zadarmo a za predpokladu, že počas vývoja budeme dbať na správne komentovanie jednotlivých časti kódu, vytvára veľmi prehladnú a systematickú dokumentáciu4.3.

RegularExpressionValidator.Server.Services.IService< TEntity > Interface Template Reference

Interface constraining basic CRUD operation used by services More...

Public Member Functions

Task< TEntity >	**Create** (TEntity entity) Creates the specified entity. More...
Task< TEntity >	**Read** (int id) Reads the specified identifier. More...
Task< bool >	**Update** (TEntity entity) Updates the specified entity. More...
Task< bool >	**Delete** (int id) Deletes the specified identifier. More...
Task< List< TEntity > >	**ListAll** () Lists all entities. More...

Obr. 4.3 Náhlad dokumentácie vytvorenej programom Doxygen

Pre inštaláciu je potrebné nasledovať kroky v priloženom súbore s názvom `README`4.4. Pre správny chod programu potrebujeme zabezpečiť funkčnú inštanciu aplikácie Docker, vytvorenie kontajneru s databázou, aplikovanie migrácie nad databázou a napokon vytvorenie kontajneru s aplikáciou.

README.md

Setup:

- Install VisualStudio 2022 Download Link
- Install WSL2 Guide Link
    - Open Command Prompt and type "wsl --install"
    - Search for "Turn Windows features on or off"
        - Select Windows Subsystem for Linux
- Install Docker for WSL Download Link
- Run file "docker-database" from within the project folder
- Open VisualStudio
    - Open Packege Manager Console Guide Link
        - Type "Update-Database"
- Run file "docker-project"
- Turn on both containers

Obr. 4.4 Náhlad súboru s inštrukciami pre inštaláciu

Kapitola 5
Zhodnotenie

5.1 Miera naplnenia našich cieľov

Ciele, ktoré sme si stanovili pri zadávaní práce sa nám podarilo splniť.

Implementácia sa dá veľmi ľahko nasadiť na prakticky ľubovoľnom systéme vďaka tomu, že ju realizujeme formou kontajnerov. Užívateľ si môže pri otvorení aplikácie v prehliadači zvoliť možnosť inštalovať ako progresívnu aplikáciu. Takto inštalovaná aplikácia podporuje prácu v offline režime pre vyhodnotenie regulárnych výrazov a zároveň ponúka užívateľské rozhranie, ktoré je nepriamo prepojené so serverom, kde môžeme manipulovať s dátami. Klient realizovaný pomocou jazyku C# a priamo využíva .NET knižnice, čo nám výrazne rozširuje obzory kam sa dá až zájsť pri vývoji takéhoto softvéru.

Vďaka rozdeleniu softvéru na jednotlivé moduly vieme ukážkový program za krátky čas nahradiť reálnou implementáciou podľa potreby.

5.2 Vývoj v Angular frameworku

Osobne mám dlhodobú skúsenosť s vývojom pomocou spomínaného frameworku Angular. V práci sme vyvíjali užívateľské rozhranie, ktoré zo serveru napísaného v Jave získavalo informácie z databázy a umožňovalo užívateľom ich spracovávať pomocou webového rozhrania.

Na vývoj som využíval Visual Studio Code. Oproti klasickému Visual Studiu ladenie bolo podstatne zložitejšie. Angular aplikácia vypisuje chyby do konzoly v prehliadači, ktoré je výrazne náročnejšie opravovať narozdiel od Visual Studio, ktoré nám nie len ukáže chybu priamo na mieste, kde sa stala, ale môžeme odkrokovať jednotlivé sekcie kódu. Angular projekt nám kompiluje a spúšťa Angular klient. Klient zaznamenával každú zmenu v systéme súborov projektu a automaticky aplikoval každú zmenu. To bolo ďaleko pohodlnejšie ako komplikovaný ladiaci mód Visual Studia, ktorý často zmeny nezaznamenal a musel som projekt nanovo

kompilovať a spúšťať.

Ukážková implementácia pri vytvorení Angular projektu uľahčí prvotné pochopenie tejto technológie. Podobne ako v našej Blazor aplikácií, Angular rozdeľuje implementáciu na jednotlivé komponenty, ktoré sa za pomoci JavaScriptu v reálnom čase renderujú v prehliadači. Spracovanie dát mala na starosti servisná vrstva, ktorá pomocou HTTP protokolu kontaktovala server a pomocou formátu JSON prijímala a odosielala informácie. JSON sa následne serializoval na objekty, ktoré vieme vytvoriť vďaka TypeScriptu a regulárne používať podľa potreby. Mať rozličné objekty na oboch koncoch spojenia prináša mnohé komplikácie. Možnosť mať vytvorenú konkrétnu triedu, ktorá sa zdieľa medzi Blazor WebAssembly implementáciou a ASP.NET Core serverom je ďaleko príjemnejšie a výrazne urýchľuje vývoj, keďže nie len vieme ľahko vytvárať nové triedy, ale máme istotu, že dáta sú konzistentné od databázy až po užívateľské rozhranie.

Nevyhnutnosť mať osobitný server prináša mnohé komplikácie. Obe časti implementácie potrebujú vlastný server, ktorý ich udržuje v behu. Napriek podobnosti v názvoch Java a JavaScript sú to výrazne odlišné jazyky s vlastnými pravidlami a celou škálou princípov, ktoré sa programátor potrebuje naučiť aby mohol takúto aplikáciu vyvíjať. Napriek všetkému skúsený vývojár by nemal mať problém vytvárať pomocou týchto technológií progresívne webové aplikácie podobné našej a považujem Angular ako platnú alternatívu k Blazor frameworku.

5.3 Vízia do budúcnosti

Implementáciu sme sa snažili zanechať čo najviac generickú. Ak by sme chceli pokračovať a implementovať tento krát už reálny informačný systém je ľahké vytvárať nové entity a k nim adekvátne komponenty v užívateľskom rozhraní určené na zobrazenie a editovanie.

V prípade, že potrebujeme realizovať komplikované algoritmy prípadne použiť .NET knižnice môžeme sa rozhodnúť či chceme prerozdeliť záťaž na klienta, alebo ich nechať vykonávať serverom. Keďže nasadenie je formou kontajnerov môžeme zvážiť využitie softvéru ako Kubernetes aby sme mohli pripraviť server aj na veľkú záťaž.

Kapitola 6
Záver

Primárnym cieľom bolo implementovať základ pre informačný systém od multiplatformového užívateľského rozhrania až po perzistenčnú vrstvu serveru. Chcel som zistiť aké výzvy ma budú čakať počas vývoja a na aké limity v rámci takéhoto riešenia narazím.

Aplikáciu sa mi podarilo dokončiť plnohodnotne a to za pomerne krátky čas oproti mojim predošlým skúsenostiam s podobným vývojom. WebAssembly v podaní Blazoru fungovalo perfektne od prvého spustenia. Ak som s niečim zápasil bol to skôr framework ako taký, no na žiadnu konkrétnu limitáciu som nenarazil. Kombinovaný s ASP.NET Core je Blazor schopný vytvárať kompletné aplikácie bez potreby ďalších špecifických technológií, ktoré nie sú priamo súčasťou Blazoru.

Odpoveď Microsoftu na WebAssembly vo forme Blazor Frameworku je skutočne šikovné riešenie a už dnes pripravené na reálnu komerčnú produkciu. Narazil som na mnohé drobné problémy, ktoré však korenili v nedostatku informácií v dôsledku zmätku, ktorý často prevláda v dokumentácií. Po pochopení základných princípov si myslím, že skúsený programátor dokáže veľmi rýchlo a efektívne produkovať naozaj kvalitný standalone softvér.

Na prvý pohľad WebAssembly budí dojem ako medzikrok od komplexného jazyka ako C# priamo do funkcionality v browseri no v tomto prípade poskytuje len .NET runtime, ktorý nám umožňuje vykonávať C# kód vo formáte .NET assemblies priamo u klienta. Takéto riešenie nie je tak efektívne ako priamo vykonávať kód preložený do WebAssembly, ale umožňuje plnohodnotnú funkcionalitu a silu väčšiny .NET knižníc. Treba spomenúť, že nový .NET 6 framework už vie poskytnúť sľúbenú ahead of time kompiláciu, ktorá produkuje priamo WebAssembly čím výrazne zvýši efektivitu za cenu dlhej kompilácie. Zatiaľ mi to však nepríde vhodné pre bežný vývoj, keďže rozdiel nie je veľmi výrazný a zreteľne komplikuje celý vývoj čo je v kontraste s primárnym dôvodom prečo som sa rozhodol používať práve Blazor.

Nahradí teda WebAssembly JavaScript? Ako ste sa mohli dočítať v mojej práci WebAssembly je na JavaScripte priamo závislé a potrebuje ho k svojmu fungova-

niu. To však nemení nič na veci, že Blazor mi, po mojich skúsenostiach s konkurenčným Angularom, príde ďaleko šikovnejší a praktickejší. Primárny cieľ používať jeden jazyk pre celý vývoj je tu perfektne vyšperkovaný a doposiaľ som nezistil limity, ktoré by Blazor nezvládol v porovnaní s konkurenciou. Po dokončení práce som toho názoru, že sa svet webových aplikácií navždy zmení, nastáva doba WebAssembly.

Bibliografia

1. HIEL, Amy van der. *World Wide Web Consortium (W3C) brings a new language to the web as WebAssembly becomes a W3C recommendation*. [B.r.]. Dostupné tiež z: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en#:~:text=https%3A%2F%2Fwww.w3.org,for%20efficient%20execution%20and%20compact.
2. ÖDEMIS, Osman Recai. *Don't use GWT or google web toolkit ?* Blog about digital transformation, 2021. Dostupné tiež z: https://blog.oedemis.io/dont-use-gwt-or-google-web-toolkit.
3. [B.r.]. Dostupné tiež z: https://www.w3.org/.
4. DEGROAT, T.J. *The history of JavaScript: Everything you need to know*. 2021. Dostupné tiež z: https://www.springboard.com/blog/data-science/history-of-javascript/#:~:text=JavaScript%20Origins,-(Source.)&text=In%20September%201995%2C%20a%20Netscape,LiveScript%20and%2C%20later%2C%20JavaScript..
5. *Webassembly*. [B.r.]. Dostupné tiež z: https://developer.mozilla.org/en-US/docs/WebAssembly.
6. MEHMOOD, Ehtesham. *What is common language runtime (CLR)*. [B.r.]. Dostupné tiež z: https://www.c-sharpcorner.com/UploadFile/9582c9/what-is-common-language-runtime-in-C-Sharp/.
7. HEDDINGS, Anthony. *What is Microsoft's blazor web framework, and should you use it?* CloudSavvy IT, 2021. Dostupné tiež z: https://www.cloudsavvyit.com/12493/what-is-microsofts-blazor-web-framework-and-should-you-use-it/.
8. TWITTERGITHUBGLITCHHOMEPAGE, Sam Richard; RICHARD, Sam; TWITTERGITHUBGLITCHHOMEPAGE; TWITTERGITHUBGLITCHHOMEPAGE, Pete LePage; LEPAGE, Pete. *What are progressive web apps?* [B.r.]. Dostupné tiež z: https://web.dev/what-are-pwas/.

9. CHEUNG, Kevin. *Moving a 30 Year Code Base to the Web*. 2020. Dostupné tiež z: https://qconnewyork.com/ny2018/presentation/autocad-webassembly-moving-30-year-code-base-web.
10. SCOPEL, Fabio; LATHAM, Luke; ANDERSON, Rick. *ASP.NET Core Blazor*. [B.r.]. Dostupné tiež z: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0.
11. MORRIS, Peter. *Component lifecycles*. [B.r.]. Dostupné tiež z: https://blazor-university.com/components/component-lifecycles/.
12. FLETCHER, Patrick. *Introduction to SignalR*. [B.r.]. Dostupné tiež z: https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr.
13. THORNTON, Jacob; OTTO, Mark. *Grid system*. [B.r.]. Dostupné tiež z: https://getbootstrap.com/docs/4.0/layout/grid/.
14. STROPEK, Rainer; ROUSOS, Mike. *ASP.NET core Blazor Dependency injection*. [B.r.]. Dostupné tiež z: https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/dependency-injection.
15. *Dependency injection*. Wikimedia Foundation, 2022. Dostupné tiež z: https://en.wikipedia.org/wiki/Dependency_injection.
16. RUBENS, Paul. *What are containers and why do you need them?* 2017. Dostupné tiež z: https://www.cio.com/article/247005/what-are-containers-and-why-do-you-need-them.html.
17. SHIRODKAR, Akita. *What is kubernetes?* 2022. Dostupné tiež z: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.
18. SCHAEFER, Lauren. *What is nosql? NoSQL databases explained*. [B.r.]. Dostupné tiež z: https://www.mongodb.com/nosql-explained.
19. *What is entity framework? (Tutorial)*. [B.r.]. Dostupné tiež z: https://www.entityframeworktutorial.net/what-is-entityframework.aspx.
20. *MySQL*. Wikimedia Foundation, 2022. Dostupné tiež z: https://en.wikipedia.org/wiki/MySQL.
21. *License*. The PostgreSQL Global Development Group, [b.r.]. Dostupné tiež z: https://www.postgresql.org/about/licence/.
22. [B.r.]. Dostupné tiež z: https://www.sqlite.org/howitworks.html.
23. *What is Angular?* [B.r.]. Dostupné tiež z: https://angular.io/guide/what-is-angular.
24. VAILSHERY, Lionel Sujay. *Most used web frameworks among developers 2021*. 2022. Dostupné tiež z: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/.
25. [B.r.]. Dostupné tiež z: https://www.doxygen.nl/.

Obsah priloženého média

- `readme.txt` ... stručný popis obsahu média
- `src`
  - `Client` ... zdrojové kódy implementácie klietnskej aplikácie
  - `Documentation` ... dokumentácia
  - `Server` ... zdrojové kódy implementácie serveru
  - `Shared` ... zdrojové kódy implementácie zdieľaného projektu
- `thesis` ... zdrojová forma práce ve formátu LaTeX
- `text` ... text práce
  - `thesis.pdf` ... text práce vo formáte PDF