VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVÁNÍ BLAZOR KOMPONENT Z C# TŘÍD
GENERATING BLAZOR COMPONENTS FROM C# CLASSES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DÁVID ŠPAVOR
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN PLUSKAL
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Špavor Dávid**
Program: Informační technologie
Název: **Generování Blazor komponent z C# tříd**
Generating Blazor Components from C# Classes
Kategorie: Informační systémy

Zadání:

1. Nastudujte technologie .NET, Blazor a jiné webové technologie umožňující automatizované generování UI z DTO (Data Transfer Object).
2. Analyzujte open source informační systémy, dostupnou literaturu a vyberte běžně používané typy UI komponent, které se v těchto systémech často opakují. Z těchto komponent vytvořte sady pohledů, které reprezentují scénáře z reálného použití.
3. Navrhněte framework, který umožní automatizovaně generovat za běhu pohledy pro Blazor v aplikacích na základě DTO. Uvažujte testovací sadu pohledů z bodu 2.
4. Navržený framework z bodu 3 implementujte a testujte pomocí jednotkových testů.
5. Vytvořte demonstrační aplikaci prezentující testovací scénáře z bodu 2. Porovnejte způsob použití vašeho frameworku s ostatními z bodu 1.

Literatura:

- Litvinavicius, T., 2019. Blazor hosted. In *Exploring Blazor* (pp. 67-111). Apress, Berkeley, CA.
- Litvinavicius, T., 2019. Blazor server-side. In *Exploring Blazor* (pp. 19-42). Apress, Berkeley, CA.
- Aponte, M., 2020. Blazor Server vs. Blazor WebAssembly. In *Building Single Page Applications in .NET Core 3* (pp. 19-31). Apress, Berkeley, CA.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 26. října 2020

Abstrakt

Táto práca sa zameriava na možnosť automatického generovania pohľadov v dynamických webových aplikáciach vytvorených pomocou technológie Blazor. Úlohou je analyzovať časti užívateľského rozhrania webových aplikácií, technológiu Blazor a vybrať také pohľady, ktoré by bolo možné automaticky vygenerovať a tým zjednodušiť a zefektívniť vývoj. Pre túto úlohu som navrhol a implementoval framework, ktorý obsahuje komponenty podporujúce generovanie pohľadov na základe kontextu v podobe modelu dát obsiahnutých v C# triede. Tieto komponenty podporujú parametre, pomocou ktorých je možné prispôsobovať pohľady priamo vrámci kódu. Výsledkom práce sú implementované dve rozsiahle C# knižnice, ktoré sú dostupné online ako open-source projekt a aj ako balíčky pre možnosť integrácie do existujúcich Blazor aplikácií.

Abstract

This thesis focuses on the possibility of automatic generation of views in dynamic web applications using Blazor technology. The task is to analyze parts of the user interface of web applications, Blazor technology and select such views that could be automatically generated, thus simplifying and streamlining development. For this task, I designed and implemented a framework that contains components that support the generation of views based on context in the form of a model of data contained in the C# class. These components support parameters that can be used to customize views directly within the code. The result of the work are implemented two extensive C# libraries, which are available online as an open-source project and also as packages for the possibility of integrating into existing Blazor applications.

Kľúčové slová

Blazor, generovať, .NET, užívateľské rozhranie, Razor, trieda, C#

Keywords

Blazor, genarate, .NET, user interface, Razor, class, C#

Citácia

ŠPAVOR, Dávid. *Generování Blazor komponent z C# tříd.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Generování Blazor komponent z C# tříd

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jana Pluskala. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

. . . .
Dávid Špavor
6. mája 2021

Poďakovanie

Rád by som sa poďakoval pánovi Ing. Janovi Pluskalovi za včasné a vecné odpovede na moje otázky a za jeho rady, vďaka ktorým ma nasmeroval správnym smerom.

Obsah

1 Úvod **2**

2 Webové technológie **3**
- 2.1 Platforma .NET a webový vývoj . . . 3
- 2.2 Technológia Blazor . . . 4
- 2.3 Data Transfer Object . . . 8
- 2.4 Iné nástroje umožňujúce automatizované generovanie UI . . . 10

3 Analýza užívateľského rozhrania **13**
- 3.1 Bežne používané UI komponenty . . . 13
- 3.2 Užitočné pravidlá pri návrhu UI . . . 15
- 3.3 Analýza UI open-source aplikácie . . . 17

4 Návrh frameworku **21**
- 4.1 Princíp generovania komponentov . . . 21
- 4.2 Možnosti prispôsobenia . . . 23
- 4.3 Návrh pohľadov . . . 26

5 Implementované knižnice **31**
- 5.1 Knižnica reflexie . . . 31
- 5.2 Knižnica komponentov . . . 34

6 Testovanie **37**
- 6.1 Jednotkové testy . . . 37
- 6.2 Integrácia do Blazor aplikácie . . . 39
- 6.3 Použitie komponentov generovania . . . 40
- 6.4 Porovnanie s existujúcimi aplikáciami . . . 43
- 6.5 Spätná väzba komunity a možné rozšírenia . . . 44

7 Záver **45**

Literatúra **46**

A Obsah priloženého dátového média **48**

Kapitola 1

Úvod

Vývoj webových aplikácií je v dnešnej dobe veľmi populárny. Existuje množstvo nástrojov a technológií, vďaka ktorým vývojári vytvárajú či už jednoduché statické stránky alebo zložitejšie dynamické stránky. Tieto webové aplikácie sa skladajú z rôznych pohľadov, pomocou ktorých prezentujú užívateľovi svoj zámer a aj pomocou ktorých je možné so stránkou interagovať. Niektoré tieto pohľady majú rovnakú štruktúru a líšia sa len v obsiahnutých dátach. Môžu to byť napríklad rôzne zoznamy, tabuľky či formuláre. Pri zložitejších aplikáciach, dáta, ktoré sú zobrazované v pohľadoch, sú získavané pomocou komunikácie medzi klientskou stránkou a obslužujúcim serverom. Z týchto dát môžu byť vytvorené objekty, ktoré predstavujú model dát pre zobrazované pohľady. Jedna z hlavných myšlienok tejto práce je zamyslieť sa nad možnosťou použiť tento model dát na automatické vygenerovanie takýchto pohľadov a tým zefektívniť a zjednodušiť vývoj.

V roku 2019 firma Microsoft predstavila novú technológiu na vývoj webových aplikácií. Jej názov je Blazor. Vývojári pomocou technológie Blazor dokážu vytvoriť kvalitnú webovú aplikáciu, kde využívajú výhody programovacieho prostredia od firmy Microsoft. Hlavné výhody sú napríklad dostupné populárne vývojové nástroje, podpora rozširujúcich balíčkov, komplexný programovací jazyk C# a veľká komunita vývojárov. Práve preto som sa rozhodol zamerať práve na technológiu Blazor, pretože v nej vidím veľký potenciál do budúcnosti.

Cieľom bakalárskej práce je vytvoriť framework, ktorý umožní automaticky generovať pohľady na základe modelu dát obsiahnutých v C# triede. Tento framework musí obsahovať sadu často používaných komponentov a pohľadov z webových aplikácií, ktoré následne vytvoria výsledné vygenerované užívateľské rozhranie. Cieľom je aj poskytnúť možnosti prispôsobenia štruktúry a vzhľadu vygenerovaného pohľadu. Výsledné knižnice budú online dostupné ako balíčky, pomocou ktorých je možné rozšíriť existujúcu Blazor aplikáciu o možnosť automatického generovania pohľadov.

V kapitole 2 sa venujem problematike webového vývoja na Microsoft platforme. Popisuje sa tu detailne technológia Blazor a použitie data transfer objektov pri vývoji webových aplikácií. Tiež sa zameriavam na existujúce technológie, ktoré umožňujú automaticky generovať pohľady. V ďalšej kapitole 3 analyzujem zloženie a dizajn užívateľských rozhraní aplikácií. Na základe všetkých týchto informácií som navrhol pohľady, ktoré framework bude generovať. Dizajn týchto návrhov sa nachádza v kapitole 4, spolu s logikou generovania a možnosťou prispôsobovania pohľadov. Tento návrh je neskôr transformovaný do dvoch implementovaných knižníc, ktoré popisuje kapitola 5. Následne v kapitole 6 sa popisuje testovanie, použitie a aj analýza výsledného riešenia.

Kapitola 2

Webové technológie

Kapitola sa zaoberá webovými technológiami od spoločnosti Microsoft, predstaví úvod do platformy .NET a jej rozšírenia ASP.NET a tiež priblíži bližšie fungovanie webovej technológie Blazor. V kapitole je tiež rozoberaný pojem data transfer object a jeho použitie pri vývoji webových aplikácií. Následne sa tu nachádza aj analýza existujúcich riešení pre automatické generovanie UI.

2.1 Platforma .NET a webový vývoj

Platforma .NET (dotnet) je vývojová platforma vytvorená z vývojových nástrojov, programovacích jazykov a knižníc pre vytváranie rôznych typov aplikácií [9]. Vďaka .NET platforme sme schopní vytvárať webové, mobilné, desktopové aplikácie, rôzne služby fungujúce v kontajneroch či 2D alebo 3D hry.

Obsah .NET platformy môžme zhrnúť do nasledujúcich bodov:

- programovacie jazyky C#, F# a Visual Basic,
- základne knižnice pre prácu so stringami, dátumami, súbormi/IO a, mnohé ďalšie,
- textové editory a integrované vývojové prostredia pre Windows, Linux, macOS a Docker.

Platformu môžeme momentálne rozdeliť do troch hlavných verzií:

- .NET Framework,
- .NET Core,
- .NET 5.

ASP.NET

ASP.NET (Active Server Pages) je open-source webový framework vytvorený firmou Microsoft pre vývoj moderných webových aplikácií a služieb pomocou .NET platformy 2.1. Rozširuje platformu .NET o nástroje a knižnice pre vytváranie webových stránok — statických či dynamických. Rozšírenia, ktoré prináša ASP.NET sú napríklad:

- framework pre spracovávanie webových požiadavok,

- Razor stránky (2.2) na vytvárania dynamických webových aplikácií ,
- knižnice pre známe webové návrhové vzory (MVC),
- autentikačný systém,
- špecifické rozšírenia pre textové editory na webový vývoj. [9]

V roku 2002 vyšla prvá verzia ASP.NET 1.0 [1] fungujúca nad vtedy novým prostredím s názvom .NET Framework. V priebehu rokov vyšlo mnoho nových verzií, ktoré priniesli množstvo vylepšení. Vývoj bol avšak obmedzený iba na systémy Windows, preto Microsoft začal vyvíjať novú platformu ASP.NET Core. Tá priniesla výkonnostné vylepšenia ale hlavne umožňovala cross-platform vývoj. Prvá verzia ASP.NET Core vyšla v roku 2016 a postupne nahrádzala ASP.NET fungujúci nad .NET Frameworkom. Platforma .NET Core sa postupne vyvíjala a nakoniec z nej v roku 2020 vznikol .NET 5, ktorý spojil z oboch platforiem to najlepšie. Momentálne je už vo vývoji aj .NET 6, ktorý ďalej rozširuje .NET 5.

2.2 Technológia Blazor

Blazor je webový framework založený na prostredí .NET umožňujúci vývoj frontendu webových aplikácií. Dovoľuje vyvíjať frontend logiku inak, ako je zaužívané a to používaním programovacieho jazyka C#, čo môže priniesť určité výhody [8, Ch. 1, p. 1]. Aké sú to výhody, si môžme uviesť na príklade štandardné webového projektu z praxe.

Na vývoj štandardného webového projektu sú potrební:

- aspoň jedna osoba pre JavaScript frontend,
- minimálne ďalšia osoba pre backend,
- dizajnér, ktorý pracuje s HTML elementami, s kaskádovými štýlmi či s inými úlohami v oblasti dizajnu.

Technológia Blazor určite neodstráni závislosť na dizajnéra ale určite dokáže odstrániť závislosť na JavaScript frontend. To znamená, že je možné vyvíjať webové aplikácie bez znalostí JavaScriptu. Blazor používa Razor syntax [2.2] (C# pomiešaný s HTML), kde HTML časť sa vyrenderuje iba raz, zatiaľ čo kód v C# sa volá či už pri načítaní samotnej stránky alebo vyvolaní nejakých udalostí (stlačenie tlačidla, zmena dát...) [8, Ch. 1, p. 1-2].

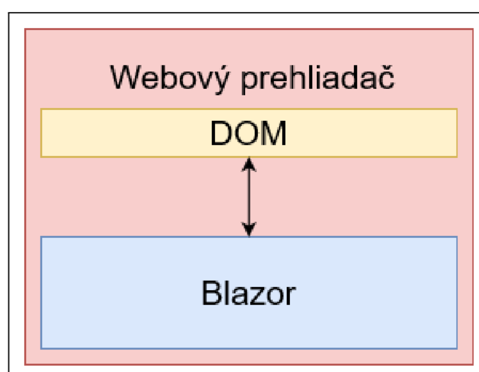
WebAssembly

Blazor umožňuje vytvárať aplikácie, ktoré sú schopné fungovať na technológii WebAssembly. WebAssembly (WASM) je binárny formát inštrukcií pre virtuálne stroje, ktoré sú zásobnikovo založené. WASM je navrhnutý byť prenosným cieľom pre kompilačné kódy rôznych programovacích jazykov, ktorý umožňuje nasadenie či už klientských alebo serverových webových aplikácií [18].

WASM podporujú všetky populárne webové prehliadače (Chrome, Mozilla, Safari, Edge) v svojich najnovších verziách. Vďaka WebAssembly je možné vytvoriť zložité, dynamické webové aplikácie, ktoré fungujú priamo vo webovom prehliadači. Práve Blazor poskytuje možnosť vytvárať aplikácie postavené aj na WebAssembly.

Blazor Client WebAssembly

Blazor Client WebAssembly je klientská aplikácia, ktorá priamo beží v prehliadači na WASM kóde, do ktorého sa skompiloval z .NET runtimu. Celá Blazor aplikácia s všetkými závislosťami je nahraná do podporovaného webového prehliadača. Po načítaní aplikácie sa .NET runtime spustí a odkáže sa na kód aplikácie po zostavení (assembly code). Spustí sa začiatočná logika aplikácie a vyrenderuje sa koreňový komponent s všetkými potomkami. Každé aktualizovanie užívateľského rozhrania vypočíta Blazor na základe vyrenderovaných komponentov vrámci webového prehliadača. Následne sa aktualizuje DOM [1] a prejavia sa zmeny na stránke [13, Ch. 3, p. 8]. Vizualizácia štruktúry Blazor Client WebAssembly a jej komunikácia s DOM je zobrazená na obrázku 2.1.



Obr. 2.1: Vizualizácia Blazor WebAssembly aplikácie, ktorá komunikuje s DOM vrámci webového prehliadača.

Výhody klientského prístupu:

- Aplikácia je veľmi rýchla s takmer rýchlosťou natívnych aplikácií.
- Aplikácia funguje aj v offline móde, čo znamená, že nepotrebuje internetové pripojenie pre správne fungovanie.
- Nie je potrebný server, keďže všetky súbory sa nahrajú do prehliadača, čo prináša jednouché nasadenie aplikácie [15].

Nevýhody klientského prístupu:

- Aplikácia je obmedzená možnosťami webového prehliadača.
- Dlhšie načítavanie stránok kvôli veľkému množstvu súborov.
- Keďže je potrebný WASM, aplikácia nebude fungovať na starších webových prehliadačoch [15].

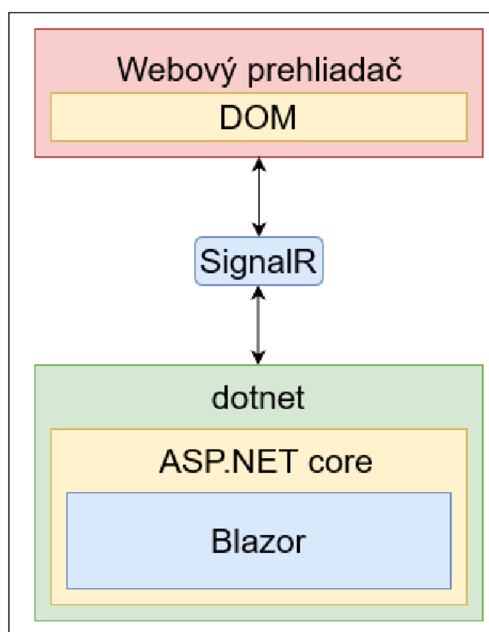
Blazor Server

Blazor Server je typ aplikácie fungujúci na webovom servery vrámci prostredia ASP.NET. Narozdiel od klientskeho prístupu, kde komunikácia medzi aplikáciou a DOM-om prebieha priamo vrámci prehliadača, DOM sa aktualizuje zo servera cez knižnicu SignalR.

[1] DOM: Document Object Model

SignalR je knižnica, ktorá posktytuje real-time funkcionalitu pre webové aplikácie. To znamená, že server je schopný poslať dáta do pripojeného klienta v reálnom čase, ktoré sú schopné sa aj hneď zobraziť. Vďaka tomu technológia SignalR umožňuje vytvárať webové aplikácie, ktoré vyžadujú veľké množstvo aktualizácií zo servera. Príklad použitia môže byť hranie hier v reálnom čase vo webových aplikáciach [10]. Vizualizácia štruktúry Blazor Server aplikácie a jej komunikácia s DOM cez SignalR je zobrazená na obrázku 2.2.

Narozdiel od Blazor WebAssembly aplikácie, komponenty sú vyrenderované do abstrakcie s názvom RenderTree priamo na serveri. Blazor porovnáva aktuálny RenderTree s predchádzajúcim a ich rozdiel posiela klientovi, kde sa s potrebnými zmenami aktualizuje DOM [13, Ch. 3, p. 9].



Obr. 2.2: Vizualizácia Blazor Server aplikácie, ktorá komunikuje s DOM cez technológiu SignalR.

Výhody prístupu zo servera:

- Počet stiahnutých súborov do prehliadača je výrazne menší, vďaka čomu je načítanie aplikácie rýchlejšie.
- Možnosť využívať technológii servera a všetkých jeho zdrojov.
- Možnosť debuggovať aplikáciu.
- Podpora starších webových prehliadačov [15].

Nevýhody prístupu zo servera:

- Fungovanie aplikácie offline nie je možné.
- Keďže každá interakcia s aplikáciou sa posiela na server, pri väčšom vyťažení môžu vznikať pri odpovediach servera oneskorenia.
- Nasadenie aplikácie do produkcie je zložitejšie a drahšie kvôli hosťovaniu servera a jeho zdrojov [15].

Komponenty a Razor

Aplikácie Blazor podobne ako u frameworkoch React či Angular sú založené na komponentoch. Komponent v Blazor aplikácií si môžeme predstaviť ako UI element, ktorý predstavuje napríklad stránku, dialóg, či formulár. Komponenty v Blazor sú C# triedy, ktoré:

- definujú logiku UI renderovania,
- spracuvávajú udalosti,
- môžu byť vnorené a znovupoužitelné,
- môžu sa distribuovať ako balíček [2].

Komponenty sú písané pomocou syntaxu Razor, preto môže byť na ne odkazované aj ako Razor komponenty. Razor je značkovací jazyk, ktorý sa skladá z C# a HTML kódu. V Blazor, súbory obsahujúce Razor syntax majú príponu *.razor* [12]. Tento syntax bol pôvodne vytvorený pre webový framework od Microsoftu nazývaný Razor Pages, vďaka ktorému bolo možné vytvárať dynamické webové aplikácie. Razor Pages priniesli koncept navigovania medzi stránkami špecifikovaný v zdrojových súborov. Súbor môže obsahovať direktívu *@page*, pomocou ktorej môžeme špecifikovať, na ktorej adrese sa daná stránka bude nachádzať. Tento prístup sa využíva aj v Blazor komponentoch. Kód 2.3 predstavuje ilustračnú štruktúru Blazor komponentu a tiež ukazuje použitie direktívy *@page*, ktorá je nastavená na koreňovú adresu stránky.

Blazor komponenty využívajú Data Binding (väzba dát). Data Binding je vo všeobecnosti technika, pomocou ktorej sme schopný prepojiť spolu dve dátové/informačné zdroje a udržovať ich dáta v synchronizácii [16]. Vo webových aplikáciach často musíme zachytávať zmeny, ktoré užívateľ vykoná a ukladať modifikované dáta. Jedna možnosť na získanie modifikovaných dát je kontrolovať, či sa dáta zmenili a ak áno, následne ich načítať na ďalšie spracovanie. Druhá možnosť je použiť techniku Data Binding, vďaka ktorej sa zmeny automaticky zachytia do zosynchronizovanej dátovej premennej. Ako príklad si môžeme uviesť zápis textu užívateľom do textového poľa. Textové pole je pomocou Data Bindingu synchronizované s dátovou premennou v zdrojovom kóde aplikácie. Keď sa zmení text v textovom poli, automatický sa zmení text aj v prepojenej dátovej premennej. Vďaka tomu môžeme jednoduchšie spracovávať dáta a reagovať na zmeny vytvorené užívateľom. Blazor používa dva hlavné typy Data Bindingu:

- One-Way Data Binding (jednosmerný),
- Two-Way Data Binding (obojsmerný).

One-way Data Binding synchronizuje dáta medzi zdrojmi v jednom smere. Keď sa zmení prvý zdroj, zmeny sa prejavia aj v druhom zdroji, ale keď sa zmení druhý zdroj, prvý zdroj zmeny nezachytí.

Two-Way Data Binding synchronizuje dáta v oboch smeroch. Keď sa zmení prvý zdroj, zmení sa aj druhý a opačne. V kóde 2.3 môže vidieť príklad Two-Way Data Bindingu pomocou direktívy *@bind* medzi elementom *input* na riadku 4 a dátovou premennou *DataVariable* na riadku 8. Po stlačení tlačidla *Change* na riadku 5 sa zmení text v premennej *DataVariable* a pomocou Data Bindingu aj v zobrazovaných UI elementoch.

```
@page "/"

<p>@DataVariable</p>
<p><input @bind="@DataVariable"></p>
<button @onclick="@(() => ChangeText())">Change</button>

@code {
    string DataVariable = "Hello World!";
    void ChangeText()
    {
        DataVariable = "Hello Programmer!";
    }
}
```

Obr. 2.3: Príklad štruktúry Razor súboru, ktorý ukazuje techniku Data Binding.

2.3 Data Transfer Object

Data Transfer Object (DTO) je zvyčajne inštancia POCO [2] triedy, ktorý sa používa ako kontajner na enkapsuláciu dát na prenášanie z jednej vrstvy aplikácie do druhej. Typicky sa DTO objekty používajú na získanie dát z databázovej vrstvy do prezentačnej vrstvy, kde sú v určitej forme prezentované užívateľovi [6]. Objekty neobsahujú žiadne metódy, výhradne slúžia len na prenos dát. Príklad štruktúry objektu reprezentuje obrázok 2.5.

Použitie objektov

Keď vývojár vytvára aplikácie s databázovou a prezentačnou vrstvou, musí byť schopný pristúpiť k dátam v databáze a správne ich zobraziť pre užívateľa. Jedná z možností je definovať si model dát, izomorfný s dátami v databáze a prenášať si ho medzi jednotlivými vrstvami. Toto je síce funkčné riešenie, avšak považuje sa to za dosť významnú návrhovú chybu. Aplikácia odhaľuje internú reprezentáciu dát v prezentačnej vrstve, čo môže viesť k potencionálnym bezpečnostným hrozbám. Preto je lepšie si navrhnúť modely dát (DTO objekty), pre jednotlivé použitia vrámci vyvíjanej aplikácie, kde budú premapované iba špecifické dáta z databázy.

Abstrakcia

DTO objekty môžu byť použité na abstrakciu objektov databázy od prezentačnej vrstvy. Následok je, že prezentačná vrstva je oddelená od databázovej vrstvy. Keď sa zmenia požiadavky na zobrazované dáta, zmení sa len štruktúra prezentačnej vrstvy zatiaľ čo aplikácia bude fungovať s existujúcou databázovou vrstvou [6]. Vznikne medzivrstva, ktorá slúži na obsluhu či už databázovej alebo prezentačnej vrstvy. Vďaka tomuto sú jednotlivé logiky vrstiev od seba rozdelené a sú schopné fungovať bez závislostí na seba.

Schovanie dát

DTO objekty sa taktiež môžu použiť na schovanie dát. To znamená, že pomocou DTO vrátime prezentačnej vrstve iba také dáta, ktoré potrebuje a požaduje [6].

[2] Plain Old CLR Objects sú triedy, ktoré nie sú závislé na bázovej infraštruktúre (ako napríklad EntityFramework)[7].

Uveďme si príklad. Vytvárame informačný systém pre rezerváciu lístkov na festivaly. V databáze máme tabuľku, ktorá obsahuje dáta o jednotlivých festivaloch. Obsiahnuté dáta o festivaloch reprezentuje entita FestivalEntity na obrázku 2.4. Za úlohu máme vytvoriť užívateľské rozhranie, kde budú zobrazené všetky festivaly v liste. Predpokladáme, že máme k dispozícii metódu GetAllFestivals(), ktorá nám z databázy vráti list všetkých festivalov vo forme FestivalEntity.

```
1  public class FestivalEntity {
2      public Guid Id { get; set; }
3      public string Name { get; set; }
4      public MusicGenre Genre { get; set; }
5      public string Country { get; set; }
6      public string LogoUri { get; set; }
7      public string City { get; set; }
8      public string Street { get; set; }
9      public string Description { get; set; }
10     public DateTime StartTime { get; set; }
11     public DateTime EndTime { get; set; }
12     public decimal Price { get; set; }
13     public int Capacity { get; set; }
14 }
```

Obr. 2.4: Príklad entity FestivalEntity, ktorá obsahuje rovnakú štruktúru dát ako je v databáze.

Avšak keď sa zamyslíme, aké dáta chceme zobrazovať v našom liste, zistíme, že mnoho dát z FestivalEntity je pre nás nadbytočných. Preto si vytvoríme DTO objekt FestivalListDto s pre nás relevantnou štruktúrou dát. Štruktúra DTO objektu FestivalListDto je špecifikovaná na obrázku 2.5.

```
1  public class FestivalListDto {
2      public Guid Id { get; set; }
3      public string Name { get; set; }
4      public MusicGenre Genre { get; set; }
5      public string Country { get; set; }
6      public string LogoUri { get; set; }
7      public string City { get; set; }
8  }
```

Obr. 2.5: Príklad štruktúry DTO objektu FestivalListDto, ktorého štruktúra vychádza z entity FestivalEntity.

Následne špecifikujeme mapovaciu logiku [3] objektov medzi sebou. Ako výsledok získame list festivalov, kde jednotlivý element listu obsahuje dáta relevantné pre naše užívateľské rozhranie. Následne sme schopní náš nový list použiť a relevantné dáta jednoducho zobraziť.

[3] Môžeme použiť nástroj Automapper, ktorý je k dispozícii z https://automapper.org.

2.4 Iné nástroje umožňujúce automatizované generovanie UI

Blazor je relatívne nová technológia. Prvá stabilná verzia vyšla v roku 2018. Toto môže byť tiež jeden z dôvodov, prečo som v mojom výskume nenašiel žiadnu existujúcu open-source knižnicu, ktorá by pridávala možnosť generovania pohľadov v Blazor. Na druhej strane som našiel množstvo knižníc, ktoré obsahujú sadu komponentov za účelom zjednodušiť vytváranie takýchto aplikácií. Príkladom jednej z takýchto knižníc je knižnica Telerik UI [4], ktorá ponúka UI komponenty stavané pre prostredie ASP.NET.

Jeden z hlavných dôvodov, prečo som si vybral práve framework Blazor je, že je postavený na .NET platforme. Vďaka tomu som schopný využívať všetky výhody jazyka C#. Jedna dôležitá výhoda je podpora reflexie. Vďaka reflexii sme schopní pristúpiť k inštanciám objektov v dobe behu programu a následne získať potrebnú štruktúru a dáta inštanciovaných objektov. Tento prístup by sa dal teoreticky využiť aj v JavaScriptových frameworkoch ako je napríklad React a Vue, ktoré sú založené na komponentoch. Vďaka vstavanému JavaScript objektu Reflect [5] sme tiež schopní pristúpiť v dobe behu aplikácie k dátam v DTO objektoch. Avšak pri mojom výskume existujúcich riešení som nenašiel knižnice v Blazor, ktoré by pristupovali ku generovaniu komponentov podobným spôsobom. V Reacte existuje rozširujúca knižnica plop.js [6], vďaka ktorej na základe špecifikovaní štruktúry React komponentu sme schopní vygenerovať React komponentu cez CLI (Command Line Interface) rozhranie. Vo frameworku Vue tiež existuje CLI nástroj vue-generate-component [7], ktorý obsahuje šablóny predprogramovaných UI komponentov a je schopný ich vygenerovať. Avšak toto je vskutku iný prístup v použití automatického generovania. V tejto sekcii nasledovne predstavím dva nástroje, pomocou ktorých je možné generovať užívateľské rozhranie.

Inxton

Inxton je sada nástrojov a knižníc, ktoré umožňujú programovať PLC [8] zariadenia v prostredí .NET. Jadrom frameworku je transpilátor, ktorý vytvára dvojitú reprezentáciu programu PLC v prostredí .NET [5]. Vzniknú .NET objekty, ktoré sú priamo napojené na PLC zariadenie. Tieto objekty sa následne môžu použiť na vytvorenie HMI [9] rozhrania. Na to sa používa technológia WPF.

Inxton obsahuje knižnicu s názvom Inxton.Vortex.Presentation.Wpf [10]. V tejto knižnici sa nachádza tzv. Renderer Engine, ktorý slúži na automatické generovanie UI do WPF aplikácie. Ako vstup generátora je predaný .NET objekt vytvorený transpilátorom. Následne generátor tento objekt prejde, nájde korešpondujúce pohľady pre rôzne typy elementov obsiahnuté v objekte a vygeneruje rozhranie. Tieto pohľady je možné prispôsobovať pomocou atribútov v PLC kóde, ktoré sú pomocou transpilátora prenesené aj do .NET prostredia.

[4] Telerik UI: https://www.telerik.com/blazor-ui

[5] Reflect: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect

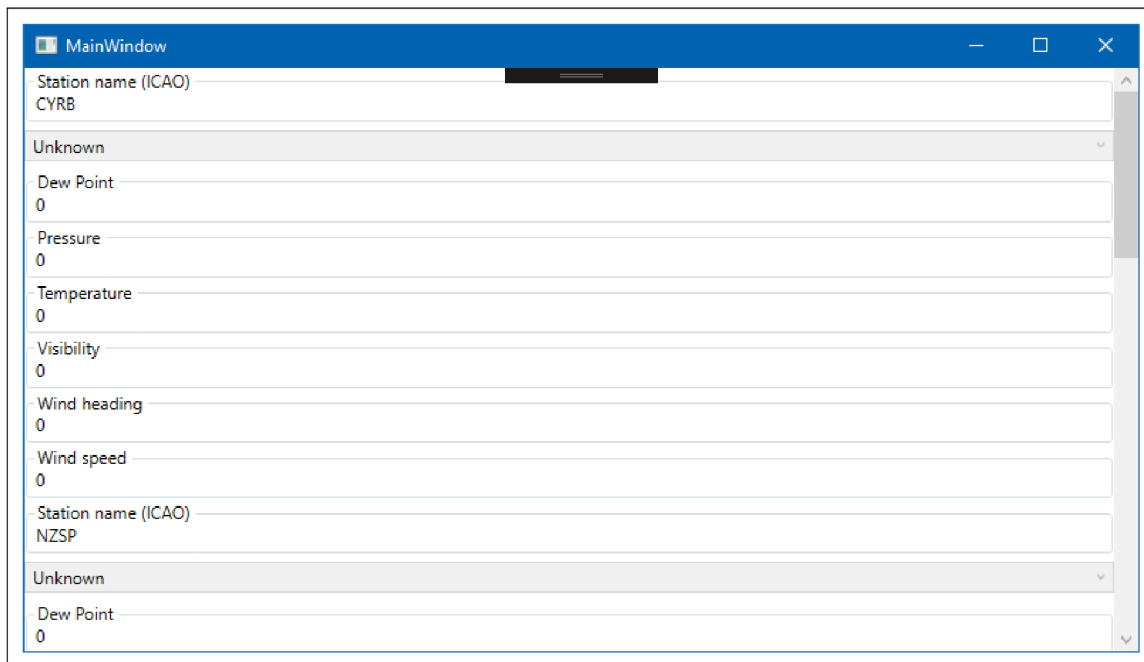
[6] plop.js: https://plopjs.com/

[7] vue-generate-component: https://www.npmjs.com/package/vue-generate-component

[8] Programmable Logic Controller

[9] Human Machine Interface

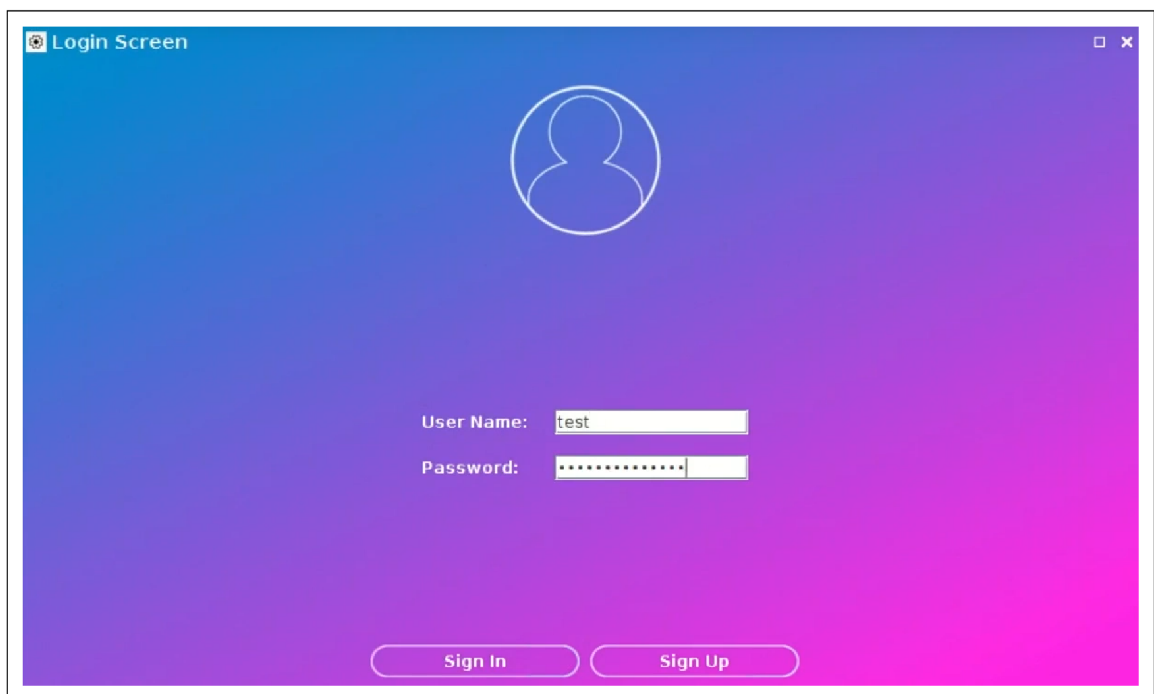
[10] https://docs.inxton.com/docu/articles/units/Inxton.Vortex.Presentation.Wpf/README.html



Obr. 2.6: Príklad užívateľského rozhrania vo WPF, ktoré vygeneroval Inxton framework.

ReflectionUI

ReflectionUI je open source knižnica na generovanie užívateľského rozhrania v jazyku Java. Okrem samotného objektu, generovanie grafického užívateľského rozhrania nevyžaduje žiadne ďalšie informácie. Jeho štruktúra je objavená za behu programu pomocou Java reflexie. Vygenerované grafické užívateľské rozhranie je možné prispôsobiť prepísaním niektorých kľúčových metód objektov generátora [11]. ReflectionUI tiež poskytuje WYSIWYG prispôsobenie. To znamená, užívateľ si v aplikácií vykliká požadované užívateľské rozhranie, ktoré sa potom vygeneruje do kódu. Príklad vytvoreného UI ukazuje obrázok 2.7.



Obr. 2.7: Príklad užívateľského rozhrania, ktoré je vytvorené pomocou Java nástroju ReflectionUI.

Kapitola 3

Analýza užívateľského rozhrania

Kapitola sa zaoberá dizajnom webových aplikácií. Sú tu predstavené často používané komponenty obsiahnuté vrámci webových aplikácií. Tiež obsahuje analýzu UI existujúcej webovej aplikácie a sú predstavené užitočné typy, ktoré by sa mali dodržiavať pri návrhu.

Dizajn webových aplikácii nie je jednoduchý proces, i keď sa na prvý pohľad nemusí zdať. Webstránka totiž musí plniť niekoľko funkcií:

- byť informatívna,
- byť praktická na používanie,
- byť estetická, príjemná na pohľad,
- efektívne komunikovať svoj zámer [19].

Nie vždy je jednoduché naplniť vyššie spomenuté funkcie. Ak sa ale budeme držať určitými dizajnovými pravidlami pri návrhu dizajnu, môže nám to značne uľahčiť robotu a zvýšiť kvalitu našej aplikácie.

3.1 Bežne používané UI komponenty

V sekcii je predstavený stručný výber bežne používaných UI komponentov vo webových aplikáciach. Webová aplikácia obsahuje rôzne pohľady, vďaka ktorým dovoľuje užívateľovi či už prezentovať alebo vkladať dáta. Pohľady sa skladajú z určitých UI komponentov, ktoré zaisťujú danú funkcionalitu. Existuje veľké množstvo rôznych UI komponentov a ich modifikácii. Preto si predstavíme iba základné a bežne používané komponenty v praxi.

Informačné komponenty

Informačné komponenty sú základne prvky každej webovej stránky. Slúžia výhradne na predávanie informácií, ktoré dokážu prezentovať s rôznou dôležitosťou. Niektoré bežne používané informačné komponenty sú:

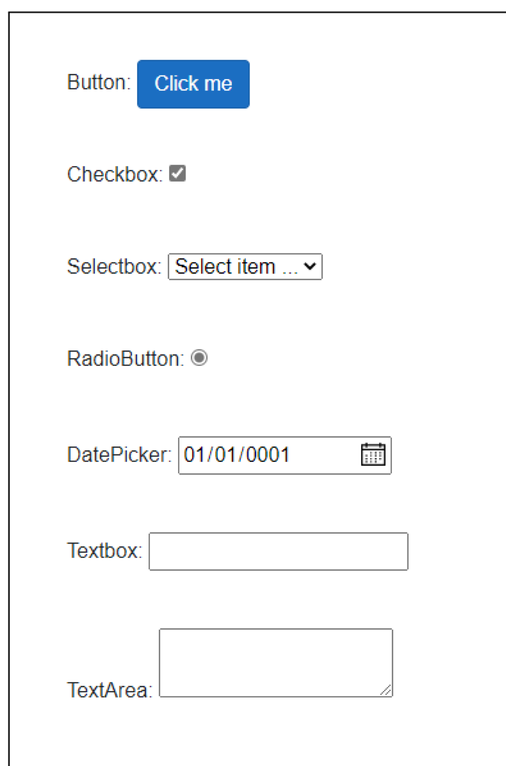
- **Nadpis** — komponenta, vďaka ktorej môžeme vytvoriť hierarchiu nadpisov (H1 — H6), čím stránku môžeme štrukturovať.
- **Paragraf** — komponenta, na zobrazovanie paragrafov textu.
- **Označenie** — komponenta, slúžiaca na vytvorenie textu, ktorý sa použije na označenie inej komponenty.

- **Obrázok** — komponenta na zobrazenie obrázku.
- **List** — komponenta na zobrazenie viacerých elementov naraz.

Komponenty s možnosťou interakcie

Komponenty s možnosťou interakcie slúžia užívateľovi na komunikovanie s rozhraním webovej aplikácie. Vďaka týmto komponentom je užívateľ schopný vytvárať rôzne akcie (napríklad uloženie, vymazanie a vkladanie dát). Príklad niektorých takýchto komponentov predstavuje obrázok 3.1. Často používané komponenty s možnosťou interakcie sú:

- **Button** — komponenta vďaka ktorej pomocou stlačenia môžeme vyvolať akciu.
- **Textbox** — komponenta, do ktorej sme schopní zadať jednoriadkový text. Pri formulároch zvykne obsahovať validáciu dát.
- **Textarea** — komponenta, do ktorej sme schopní zadať multi-riadkový text.
- **Combobox** — komponenta, kde sme si môžeme vybrať hodnotu z preddefinovaného listu možností.
- **Checkbox** — komponenta, ktorá reprezentuje bool hodnoty.
- **Radio buttons** — komponenta, slúžiaca na výber možnosti z viacerých možností.
- **Date/Time picker** — komponenta na zvolenie dátumu/času v správnom formáte.



Obr. 3.1: Ukážka často používaných komponentov, ktoré sa zvyčajne používajú v editačných formulároch.

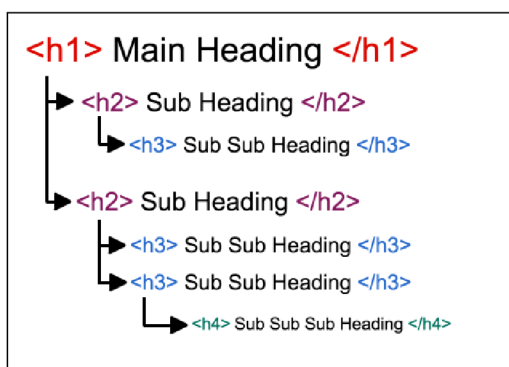
3.2 Užitočné pravidlá pri návrhu UI

Pri navrhovaní dizajnu UI je vhodné sa držať určitými pravidlami. Vďaka týmto pravidlám zvýšime značne kvalitu výsledného užívateľského rozhrania, či už z estetickej alebo funkčnej stránky.

Hierarchia nadpisov, textových štýlov a fontov

Vytvorenie hierarchie štýlov textu a výber typov písma pred začatím vývoja užívateľského rozhrania produktu je mimoriadne dôležité pre konzistentný dizajn v rámci celého projektu. Aby sme to dosiahli, musíme navrhnúť systém, ktorý bude zobrazovať hierarchiu štýlov s textovými štýlmi, počnúc od najväčšieho po najmenší a udržiavajúc ho minimálne na dvoch typoch písma [3].

Nadpisy sa používajú na zabezpečenie hierarchie a prehľadnosti webovej stránky. To pomáha návštevníkom rýchlo skenovať stránku a pomáha webovým vyhľadávačom pochopiť jej štruktúru a tému. Na obrázku 3.2 je možné vidieť ukážku vhodnej štruktúry nadpisov.



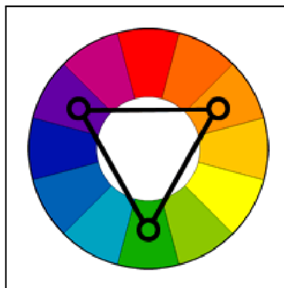
Obr. 3.2: Príklad štruktúry nadpisov, ktoré vytvárajú hierarchiu stránky.[1]

Použitie farieb

Farba je jedným zo základných a najviac užitočných nástrojov pre každého webdizajnéra. Farba môže upútať pozornosť návštevníka web stránky, ovplyvniť jeho emócie, vnímanie a v neposlednom rade aj jeho správanie na webe.

Je doporučené použiť maximálne 5 farieb, keďže zakomponovať viacej typov farieb efektívne môže byť problémové. Pre najlepšie efektívne farebné prezentovanie webovej aplikácie je vhodné použiť kombináciu troch farieb — **triadickú farebnú schému** [14], ktorú reprezentuje obrázok 3.3.

[1]Prevzaté z: https://www.nomensa.com/blog/2017/how-structure-headings-web-accessibility



Obr. 3.3: Ukážka výberu farieb z triádickej farebnéj schémy.[2]

Triádická farebná schéma používa farby, ktoré sú rovnomerne rozmiestnené okolo farebného kolesa. Použitím farieb z triádickej farebnej schémy vzniknú farebné harmónie, ktoré sú celkom živé, aj keď sa vyberú bledé alebo nenasýtené verzie odtieňov. Farby by mali byť starostlivo vyvážené — jedna farba dominuje ďalšie dve sa použijú pre zvýraznenie [17].

Použitie ikon

Primárnym cieľom používania ikon by malo byť vždy rýchle sprostredkovanie autorovej myšlienky. Dizajn ikony by mal mať zrozumiteľný symbol, čitateľný a mal by predstavovať funkcionalitu, ktorú nemožno najlepšie opísať slovami [3].

Zobrazovanie chybových hlášok

Keď užívateľ bude zadávať dáta napríklad do editačných formulárov, je veľmi pravdepodobné, že sa dopustí chýb. Je veľmi dôležité užívateľa informovať akú chybu spravil a ako čo najjednoduchšie ju vyrieši.

Štyl komponentov, kde nastala chyba musí byť dostatočne rozpoznateľný od ostatných komponentov. Chybová hláška by mala byť stručná a s informatívnym charakterom, ktorý poskytne užívateľovi nápovedu ako vyriešiť vyskytnutú chybu [3].

Použitie UI návrhových vzorov

UI návrhové vzory sú komponenty, ktoré používajú dizajnéri na vyriešenie častých problémov pri návrhu užívateľského rozhrania. Dizajnéri môžu komponenty použiť na širokú škálu prípadov avšak je potrebné ich prispôsobiť konkrétnemu kontextu [4]. Existuje veľké množstvo rôznych UI návrhových vzorov. Z dostupných zdrojov som vybral tie, ktoré si myslím, že sú dostatočne relevantné pre moju prácu.

- **Breadcrump** — umožňuje užívateľovi zobraziť jeho aktuálnu lokáciu na stránke súčasne s hierarchiou stránky. Vďaka tomu je užívateľ schopný sa jednoducho odnavigovať naspäť zo zanorenej lokácie vrámci hierarchie.
- **Forgiving Format** — umožňuje užívateľovi zadať dáta v rôznych formátoch, napríklad pri zadávaní dátumu, poštového smerovacieho čísla, adresy... Návrhový vzor je vhodné napríklad použiť pri vyhľadávaní dát naprieč stránkou, kde užívateľ nemusí zadať presný text, ale stačí iba jeho určitá časť.

[2]Prevzaté z: https://www.tigercolor.com/color-lab/color-theory/color-harmonies.htm

- **Clear Primary Actions** — slúži na informovanie užívateľa aby mu bolo jasné, čo má na stránke robiť, kam ma kliknúť a aká bude výsledná akcia. Napríklad je vhodné rozdeliť tlačidlá farbami, aby za zvýraznila priorita jednotlivých tlačidiel alebo umiestniť tlačidlo s jasným popisom na miesto, kde užívateľovi bude jasné, aká bude výsledná akcia po kliknutí.
- **Password strength meter** — komponenta, vďaka ktorej sme schopní informovať užívateľa aké silné heslo vytvoril. Týmto ho môžeme prinútiť vytvárať silné heslá a tak zvýšiť bezpečnosť.
- **Dashboard** — je vhodné použiť, keď chceme prezentovať užívateľovi dáta z viacerých zdrojov tak, aby bol schopný jedným pohľadom dáta čo najjednoduchšie spracovať.
- **Slideshow** — používame keď užívateľovi chceme prezentovať list statických obrázkov. Môže slúžiť na zachytenie pozornosti užívateľa a zároveň minimalizuje množstvo priestoru, potrebného pre zobrazenie všetkých obrázkov.

3.3 Analýza UI open-source aplikácie

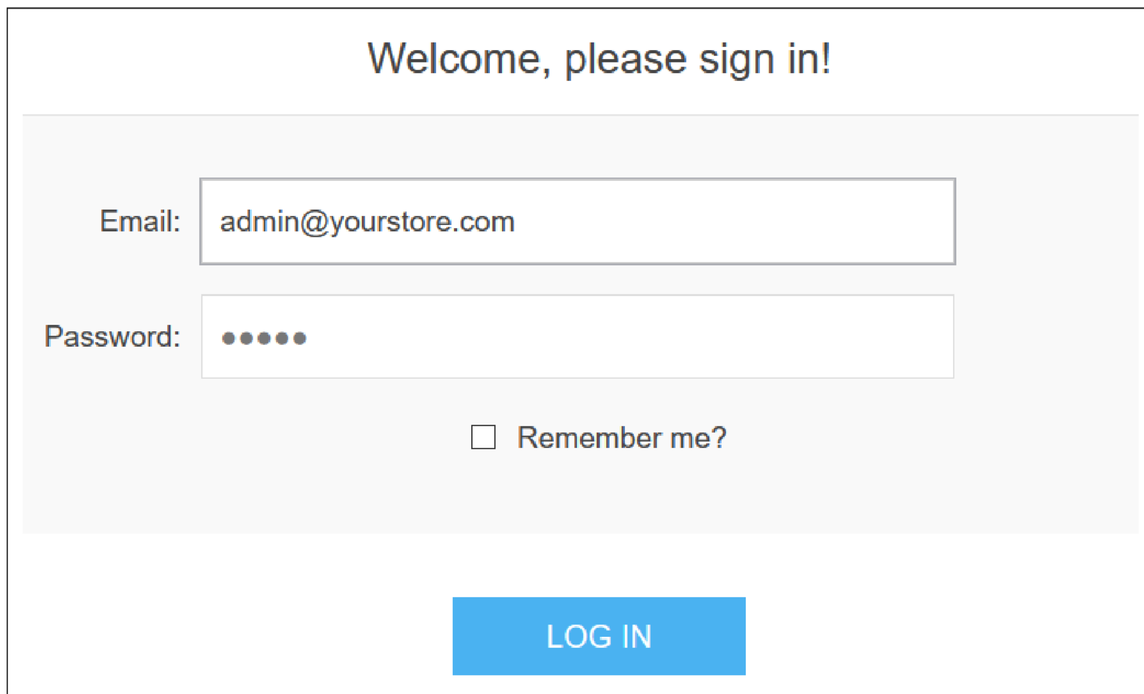
Sekcia obsahuje analýzu užívateľského rozhrania webovej aplikácie. Ako demonštračnú aplikáciu som vybral open-source projekt nopCommerce[3]. NopCommerce je najobľúbenejšie open-source riešenie aplikácie typu eCommerce založené na ASP.NET Core s vyše 6000 hviezdičkami na GitHube a veľkou komunitou prispievateľov. Táto aplikácia obsahuje pohľady, ktoré prezentujú použitie často používaných UI komponentov v praxi. Toto sú hlavné dôvody, prečo som vybral práve takúto aplikáciu. Zameral som sa hlavne na pohľady, ktoré by bolo teoreticky možné automaticky generovať. Analýzou by som chcel poukázať na zloženie takejto aplikácie, jej dizajn a možnosť generovať niektoré existujúce pohľady automaticky. Analýza bola vykonaná na dostupnom online deme[4] admin sekcie aplikácie.

Prihlasovanie a úvodná stránka

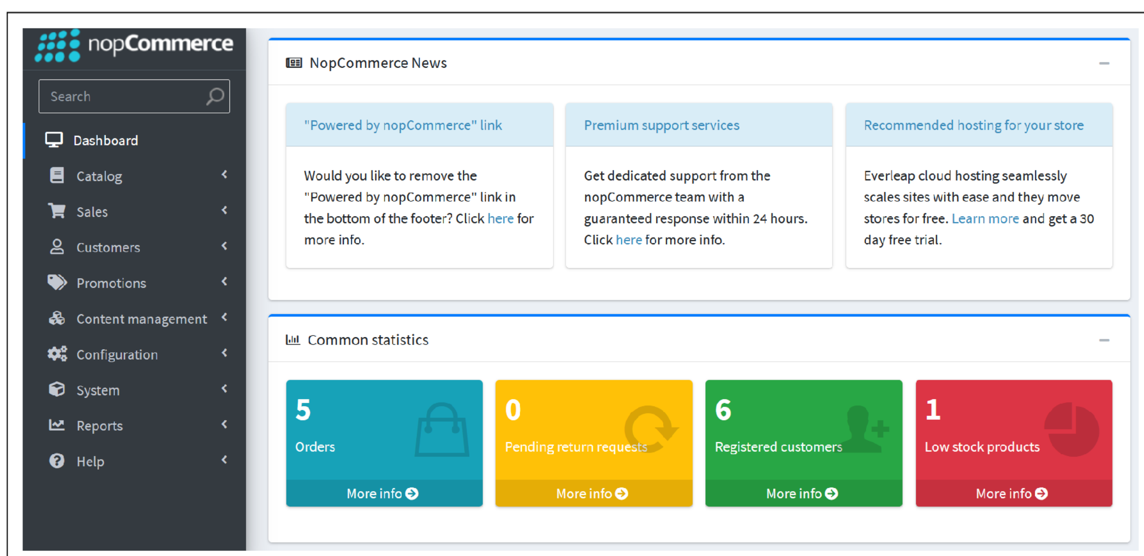
Demo aplikácia nás víta prihlasovacím pohľadom. Tento prihlasovací pohľad je zobrazený na obrázku 3.4. Prihlasovací pohľad obsahuje 2 komponenty typu textbox. Jednu pre zadanie emailu a druhú pre heslo. Obe textbox komponenty obsahujú validáciu dát — pre prihlásenie je povinné a potrebné zadať email v správnom formáte a heslo. Obe políčka nemôžu byť prázdne. Ak nie je splnená jedna z podmienok validácie dát, aplikácia poskytne chybovú hlášku so stručným textom daného problému (3.2). Prihlasovací pohľad tiež obsahuje komponentu checkbox na zapamätanie prihlásenia a tlačidlo na akciu samotného prihlásenia.

Po prihlásení sa zobrazí úvodná stránka a hlavné rozloženie aplikácie. Naľavo je zobrazené menu s vyhľadávaním. Úvodná stránka obsahuje hlavné informácie o samotnom informačnom systéme a štatistiky. Štatistiky sú zobrazené v podobe grafov a tabuliek. Všetky tieto elementy sú uložené v komponente Dashboard (3.2), pomocou ktorej sú všetky informácie podané prehľadne a v modernom dizajne. Obrázok 3.5 predstavuje úvodný pohľad aplikácie.

[3]NopCommerce: https://github.com/nopSolutions/nopCommerce
[4]Demo: https://admin-demo.nopcommerce.com/Admin



Obr. 3.4: Prihlasovací pohľad zobrazujúci základne textbox komponenty.



Obr. 3.5: Úvodný pohľad zobrazujúci menu a Dashboard komponentu.

Listové pohľady

Z ľavého menu je možné sa prekliknúť na listové pohľady — tzv. katalógy. Z dôvodu veľkého počtu rôznych typov katalógov som sa rozhodol analyzovať katalóg zákazníkov, ktorého pohľad je zobrazený na obrázku 3.6. Tento pohľad obsahuje tabuľku s informáciami o zákazníkoch s možnosťou prekliku na detail jednotlivého zákazníka. V tabuľke sú informácie o emaile, mene a spoločnosti zákazníka, jeho role vrámci systému a statusu. Keďže štruk-

túra každej položky v katalógu je rovnaká a líši sa len hodnotami, je vhodné zamyslieť sa nad použitím automatického generovania UI položiek.

☐	Email	Name	Customer roles	Company name	Active	Edit
☐	victoria_victoria@nopCommerce.com	Victoria Terces	Registered		✔	Edit
☐	brenda_lindgren@nopCommerce.com	Brenda Lindgren	Registered		✔	Edit
☐	james_pan@nopCommerce.com	James Pan	Registered		✔	Edit
☐	arthur_holmes@nopCommerce.com	Arthur Holmes	Registered		✔	Edit
☐	steve_gates@nopCommerce.com	Steve Gates	Registered		✔	Edit
☐	admin@yourStore.com	John Smith	Administrators, Forum Moderators, Registered		✔	Edit

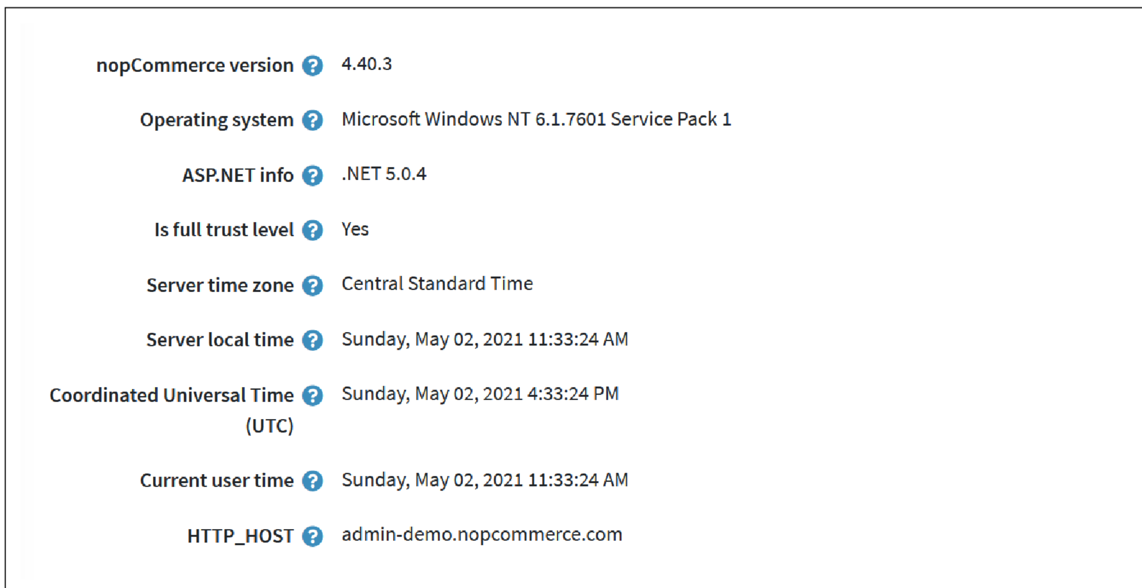
1 Show 15 items 1-6 of 6 items

Obr. 3.6: Listový pohľad zobrazujúci katalóg zákazníkov používajúcich systém.

Detailné pohľady

Aplikácia zobrazuje viacero typov detailných pohľadov, či už sú to informačné pohľady alebo pohľady v podobe formulárov. Ako príklad pre informačný pohľad som vybral pohľad zobrazujúci informácie o systéme. Tento pohľad prezentuje obrázok 3.7, kde sú zobrazené informácie o systéme a serveri, na ktorom beží. Sú tu informácie o verzií aplikácie, operačnom systéme, čase... Informácie mohli byť získané v podobe objektu priamo zo servera. Takýto objekt by sa teoreticky mohol použiť na automatické vygenerovanie UI daných informácií.

Príklad pre editačný pohľad v podobe formulára je zobrazený na obrázku 3.8. Podobne ako pri listovom pohľade som vybral pohľad zameraný pre zákazníkov. Z dôvodu veľkého rozsahu obsiahnutých polí je zobrazená len časť formulára. Tento formulár obsahuje základné informácie o zákazníkovi, ktorý používa daný systém. Formulár obsahuje rôzne typy elementov. Obsahuje textboxy pre základné údaje a názvy, radio buttons pre výber pohlavia, komponentu date picker pre výber dátumu narodenia a mnohé ďalšie. Tieto elementy sú zobrazené v jednoduchom UI kontajneri pod sebou. Takéto jednoduché UI by bolo možné automaticky vygenerovať a aj prispôsobiť.



Obr. 3.7: Detail informačného pohľadu, ktorý zobrazuje základné údaje o systéme pomocou jednoduchého UI.

Email victoria_victoria@nopCommerce.com

Password Change password

First name Victoria

Last name Terces

Gender Male

Female

Date of birth

Company name

Is tax exempt

Newsletter

Obr. 3.8: Detail editačného pohľadu v podobe formulára, ktorý zobrazuje základné informácie o zákazníkovi s možnosťou úprav.

Kapitola 4

Návrh frameworku

V tejto kapitole je obsiahnutá časť o návrhu výsledného frameworku. Je tu popísaný základný princíp generovania elementov vstupného objektu ako Blazor komponenty, ktorý sa opiera o štruktúru a typy obsiahnutých elementov. Ďalej sa rieši problematika prispôsobenia výsledných pohľadov, či už po vizuálnej stránke (rozloženie elementov, štýly) alebo z pohľadu logiky generovania (zmena poradia, ignorovanie elementov).

4.1 Princíp generovania komponentov

Hlavná myšlienka projektu je umožniť generovať užívateľské rozhranie z akéhokoľvek objektu. Avšak tento objekt môže obsahovať ďalšie rôzne elementy, pri ktorých treba zvážiť akým spôsobom budú generované. Pochopenie typov týchto elementov je dôležité pre pochopenie celkovej logiky generovania. Práve tieto typy elementov určujú chovanie frameworku a následne vloženie elementov do správne nájdených Blazor komponentov, ktoré dokopy tvoria výsledný vygenerovaný pohľad.

Typy elementov

Vstupný objekt môže obsahovať 3 typy elementov:

- primitívny element,
- komplexný element,
- pole.

Primitívny element môžeme chápať ako jednoduchý dátový typ (string, integer, bool...) nesúci hodnotu, ktorú zobrazujeme. Keďže takýchto jednoduchých dátovych typov je viacej, je vhodné použiť generické komponenty na zobrazovanie hodnôt.

Komplexný element je štrukturovaný dátový typ, ktorý môže obsahovať všetky typy elementov — primitívne, komplexné a aj pole. Komplexný element predstavuje aj vstupný objekt, z ktorého výsledný framework vygeneruje užívateľské prostredie. Logika renderovania elementov je práve prispôsobená štruktúre komplexného elementu.

Pole je dátový typ obsahujúci komplexné elementy. Idea je rovnaká ako pri samotnom komplexnom elemente, narozdiel od toho, že toto pole musíme lineárne prejsť a každý element zvlášť vyrenderovať.

Obrázok 4.1 predstavuje konkrétny príklad objektu, ktorý obsahuje všetky možné typy elementov.

```
1 // complex element
2 public class StageDetailDto {
3     public string Name { get; set; } // primitive element
4     public FestivalListDto Festival { get; set; } // complex element
5     public IList<InterpretListDto> Interprets { get; set; } // array
6 }
```

Obr. 4.1: Konkrétny príklad komplexného objektu, ktorý obsahuje ďalšie rôzne typy elementov.

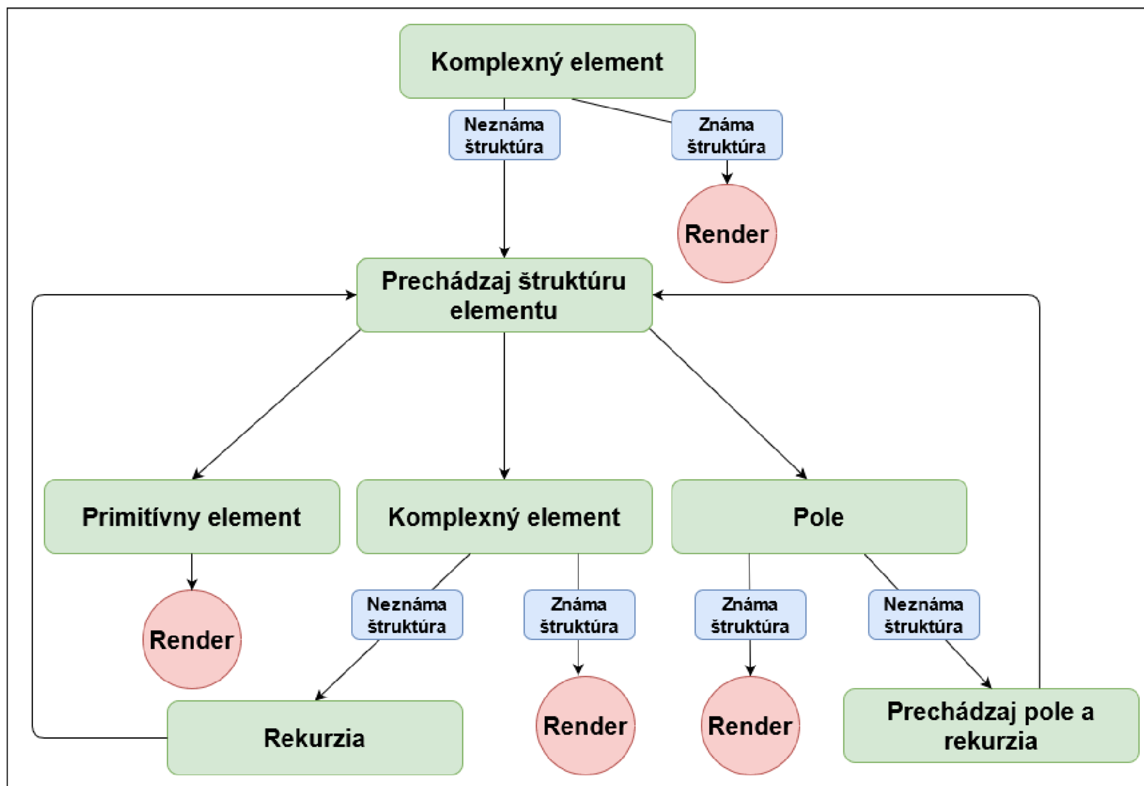
Logika generovania elementov

Logika generovania elementov je závislá na typoch elementov, ktoré vstupný objekt obsahuje. Diagram na obrázku 4.2 predstavuje, ako pri prechádzaní komplexného elementu sa vyrenderujú rôzne typy elementov na základe známej alebo neznámej štruktúry. Tieto elementy a ich nesúce hodnoty sa následne vložia do korešpondujúcich Blazor komponentov, ktoré sú dané typom hodnoty. Tieto komponenty spolu vytvoria výsledný vygenerovaný pohľad.

Na začiatku máme vstupný bod v podobe komplexného elementu. Ak obsahuje element známu štruktúru, to znamená elementy, ktoré už poznáme a sú definované v šablóne vrámci frameworku, šablónu vyrenderujeme s poskytnutými hodnotami. V prípade neznámej štruktúry elementu (štruktúra nie je obsiahnutá vrámci frameworku) musíme vstupný element prejsť a jednotlivé položky na základe ich typu vyrenderovať.

Ak sa jedná o primitívny element, zistíme jeho dátový typ a poskytneme mu príslušnú UI komponentu. Ak sa jedná znova o komplexný element, pýtame sa či poznáme jeho štruktúru. Ak poznáme, komponent vyrenderujeme do príslušnej šablóny alebo prechádzame objekt rekurzívne až kým nenarazíme na primitívne dátové typy.

V prípade, že sa jedná o pole, zistíme, či je špecifikovaná šablóna, ktorú následne generujeme, alebo iterujeme cez prvky pola a znova rekurzívne prechádzame objekty. Tento prípad platí, aj ak je pole komplexných elementov vstupný bod generovania.



Obr. 4.2: Diagram, ktorý popisuje princíp generovania užívateľského rozhrania komplexného elementu a jeho štruktúry.

4.2 Možnosti prispôsobenia

Pri používaní frameworku by mal byť užívateľ schopný si prispôsobiť vygenerované pohľady svojím potrebám. Práve preto som navrhol systém, ako špecifikovať vlastnosti generovania a s akými podporovanými možnosťami prispôsobenia.

Špecifikácia vlastností generovania

Máme dve možnosti ako špecifikovať vlastnosti, ktoré určia ako bude výsledný vygenerovaný pohľad vyzerať:

- špecifikovanie vlastností pomocou dátových anotácií vrámci objektu,
- špecifikovanie vlastností vrámci vstupného komponentu pomocou parametrov.

Oba tieto typy majú svoje výhody a nevýhody. Výhoda dátových anotácií je taká, že poskytujú komplexnejšie možnosti prispôsobenia pohľadov relatívne prehľadným a jednoduchým spôsobom. Na druhej strane ale nemôžeme ich vždy použiť, lebo môže sa stať, že naše DTO objekty budú automatický generované a enkapsulované vrámci iných objektov, vďaka čomu použitie anotácií bude značne znemožnené.

V prípade špecifikovania vlastností pomocou parametrov vstupného komponentu nie sme závislý na pôvode a type vstupného objektu, avšak špecifikovanie parametrov môže byť viacej neprehľadné a nie až také komplexné ako pri anotáciách.

Po zvážení týchto argumentov som sa rozhodol navrhnúť framework, ktorý bude podporovať oba prístupy, keďže chcem, aby užívateľ bol schopný generovať pohľady z rôznych typov objektov – či to je DTO objekt alebo statický objekt. V prípade použitia oboch typov špecifikovania vlastností naraz, bude framework preferovať možnosti špecifikované pomocou parametrov v komponente. Je to hlavne kvôli jednotnému prístupu ku generovaniu objektov vrámci použitia frameworku. Príklady použitia vlastností sú v sekcii 6.3.

Možnosti prispôsobenia pohľadov

Pre užívateľa som navrhol minimálne tieto možnosti, ako si môže vygenerovaný pohľad prispôsobiť svojím potrebám.

- Prezentačný typ (Presentation Type)
- Rozloženie elementov (Layout)
- Poradie elementov (Order)
- Ignorovanie elementov (Ignore)
- Vlastné názvy elementov (AttributeName)
- Špecifikovanie obrázkov (IsPicture)
- Štýlovanie (Styling)

Prezentačný typ

Prezentačný typ určuje typ zobrazenia elementov. Navrhol som dva prezentačné typy:

- Display — prezentačný typ, ktorý zobrazuje dáta elementov bez možnosti úprav vrámci webovej aplikácie.
- Control — prezentačný typ, ktorý umožňuje zobrazovať a aj upravovať dáta elementov.

Prezentačný typ Display sa môže použiť pri vytváraní informačných pohľadov v aplikácií. Zatiaľ čo Control sa môže použiť na vytváranie pohľadov s možnosťou interakcie, ako sú rôzne formuláre a podobne, kde užívateľom zadané dáta sú vrámci aplikácie ďalej spracuvávané. Prezentačný typ bude možné špecifikovať vrámci vstupného komponentu pomocou parametra a enum hodnoty, ktorá bude obsiahnutá vrámci frameworku.

Rozloženie elementov

Rozloženie elementov určuje vzhľad výsledného pohľadu. Užívateľ si bude môcť vybrať, či svoj objekt vyrenderuje do preddefinovanej šablóny (avšak musí zaistiť správne premapovanie DTO objektu do objektu šablóny) alebo použije niektorý z podporovaných rozložení vrámci generovania. Framework bude minimálne podporovať tieto rozloženia:

- **Stack** – základné rozloženie elementov pod seba.
- **Wrap** – rozloženie elementov vedľa seba. V prípade zmeny veľkosti okna elementy sa búdu premiestňovať pod seba.

- **Grid** – možnosť rozloženia elementov do riadkov a stĺpcov s podporou responzívneho dizajnu. Podpora špecifikovania počtu sĺpcov v riadku.
- **Tabs** – uloženie jednotlivých elementov do záložiek s možnosťou prepínanie sa medzi nimi.

Možnosť špecifikovať rozloženie bude možné aj vrámci parametrov a aj pomocou dátových anotácií. Avšak treba poznamenať, že pri špecifikovaní pomocou parametrov užívateľ nie je schopný nadefinovať rozloženie pre vnorené elementy. Pri komplexnejších rozloženiach a ich kombinácií je práve preto vhodnejšie použiť možnosť špecifikovania pomocou dátových anotácii vrámci objektu.

Poradie elementov

Pri generovaní objektu sa jeho elementy prechádzajú postupne tak, ako sú definované. Užívateľ môže poradie zmeniť tak, že jednotlivé elementy manuálne v objekte poprehadzuje. Avšak toto nie je vždy reálna možnosť. Preto som navrhol podporu pre parameter Order.

V prípade použitia Order ako parametra do vstupného komponentu generovania, je potrebné predať inštanciu dátového typu Slovník (Dictionary), kde je priradené poradie pre jednotlivé elementy objektu na základe ich mena. Príklad inštancie slovníka pre poradie predstavuje obrázok 4.3. Pridanie možnosti špecifikovania poradia elementov pomocou atribútov podľa mňa nie je potrebné. Ak má užívateľ prístup k objektu, je schopný si elementy manuálne poprehadzovať do požadovaného poradia.

```
1 var order = new Dictionary<string, int>
2 {
3     {"Image", 0},
4     { "Name", 1},
5     { "Surname", 2}
6 };
```

Obr. 4.3: Príklad vytvorenia slovníka, ktorý sa použije ako parameter na určenie vlastného poradia vygenerovaných elementov. Položka slovníka sa skladá z mena elementu a čísla, ktoré predstavuje jeho poradie.

Ignorovanie elementov

Ignorovanie elementov je vhodné použiť, keď nechceme zobrazovať všetky údaje obsiahnuté v objekte — napríklad keď objekt obsahuje nejaké ID na jeho identifikáciu. Vstupnému komponentu môžeme predať parameter v podobe dátového typu string, ktorý obsahuje názvy elementov rozdelené čiarkou, ktoré sa majú ignorovať. Druhá možnosť je pridať anotáciu RenderIgnore.

Vlastné názvy elementov

Element má vrámci objektu svoj názov. Tento názov bude označovať aj samotný UI element. Avšak nie vždy chceme používať predvolený názov. Niekedy môže byť aj viac-slovné pomenovanie elementu. Preto som navrhol možnosť špecifikovania vlastných názvov elementov. Užívateľ bude schopný nastaviť vlastné značenie elementu pomocou vstupného parametra

Labels, kde predá inštanciu typu slovník alebo pomocou dátovej anotácie. Príklad vytvorenia inštancie slovníka pre vlastné názvy je ukázaný na obrázku 4.4.

```
1 var labels = new Dictionary<string, string>
2 {
3     {"Name", "Person name:"},
4     {"Surname", "Person surname:"},
5     {"City", "Town:"},
6     {"Born", "Date Of Birth:"}
7 };
```

Obr. 4.4: Príklad vytvorenia slovníka, ktorý sa použije ako parameter na určenie vlastných názvov vygenerovaných elementov. Položka slovníka sa skladá z pôvodného mena elementu a z upraveného mena elementu.

Špecifikovanie obrázkov

DTO objekty môžu v sebe obsahovať URI obrázkov ako dátový typ string. Hlavne v prípade prezentačného typu Display nechceme zobrazovať obrázok ako URI. Práve preto som navrhol spôsob určenia elementov, ktoré sa majú vygenerovať ako obrázky. Buď sa použije anotácia IsPicture alebo v prípade špecifikovania parametrom sa predá vstupnému komponentu dátový typ string s menami elementov oddelených čiarkami, ktoré predstavujú obrázky.

Štýlovanie

Štýlovanie elementov vo frameworku bude postavené na knižnici Bootstrap. Je to hlavne kvôli tomu, že Bootstrap je jedna z najpopulárnejších knižníc pre štýlovanie. Bootstrap bude obsiahnutý priamo vo frameworku a užívateľ bude mať možnosť výberu, ktorý Bootstrap štýl bude chcieť použiť. Vďaka technológii SCSS je možné prepísať pôvodné farebné hodnoty Bootstrap tried. Následne SCSS súbory sa preložia pomocou prekladača s názvom WebCompiler [1], ktorý je dostupný ako rozšírenie Visual Studia. Výsledok prekladu sú CSS súbory, ktoré je následne možné referencovať v Blazor aplikácii. Príklad referencie na Bootstrap CSS je ukázaný na obrázku 6.2. Tým sa vytvorí sada preddefinovaných štýlov, ktoré je užívateľ schopný použiť. Je vhodné zahrnúť aj základný generický Bootstrap štýl.

4.3 Návrh pohľadov

Na základe analýzy open-source aplikácie nopCommerce, literatúry a dostupných informácií som navrhol pohľady, ktoré môj framework bude schopný vygenerovať na základe modelov dát. Nasledujúce ukážky budú obsahovať model dát (DTO objekt) a samotné UI, ktoré mu môže zodpovedať. Návrhy pohľadov sú ilustračné, pretože modelové triedy môžu obsahovať rôzne dáta, ktoré určujú finálnu podobu vygenerovaného UI. Návrhy boli vytvorené pomocou vstavaných komponentov v Blazor, ktoré sú štýlované pomocou knižnice Bootstrap.

[1] WebCompiler: https://github.com/madskristensen/WebCompiler

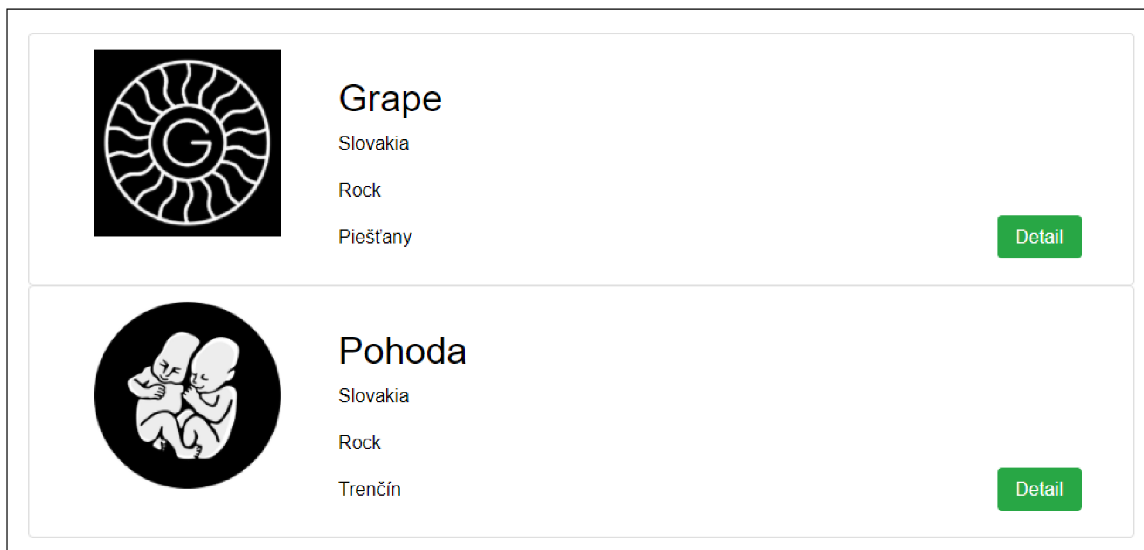
Listový pohľad

Listový pohľad má mať informatívny charakter o elementoch, ktoré zobrazuje. Z listového pohľadu by malo byť možné sa prepnúť na detailný pohľad elementu. Na to môžeme použiť UI návrhový vzor Clear Primary Actions 3.2.

Listový pohľad môžeme zobrazovať s obrázkami alebo bez obrázkov. V prípade, že zobrazujeme elementy s obrázkami, je vhodné mať šírku a výšku elementu väčšiu. Návrh pohľadu, ktorý korešponduje s DTO objektom 4.5 a zobrazuje situáciu obrázka v listovom elemente prezentuje obrázok 4.6. V druhom príklade môžeme zobrazovať iba listový element s základným popisom a s možnosťou sa presunúť na detailný pohľad. Túto situáciu vystihuje návrh 4.8, ktorému korešponduje DTO objekt 4.7.

```
1  public class FestivalListDto {
2      public string LogoUri { get; set; }
3      public string Name { get; set; }
4      public string Country { get; set; }
5      public MusicGenre Genre { get; set; }
6      public string City { get; set; }
7  }
```

Obr. 4.5: Štruktúra DTO objektu FestivalListDto, ktorý predstavuje modelovú triedu pre listový pohľad s obrázkom.



Obr. 4.6: Návrh listového pohľadu, ktorý obsahuje text a obrázky.

```
1  public class FestivalListBaseDto {
2      public string Name { get; set; }
3  }
```

Obr. 4.7: Štruktúra DTO objektu FestivalListBaseDto, ktorý predstavuje modelovú triedu pre základný listový pohľad.

Grape Detail

Pohoda Detail

Oktoberfest Detail

Tommorowland Detail

Obr. 4.8: Návrh listového pohľadu, ktorý neobsahuje obrázky.

Detailný a editačný pohľad

Detailný pohľad môže obsahovať väčšie množstvo dát, preto ich musí užívateľovi prezentovať tak, aby im čo najjednoduchšie porozumel. Z modelu detailného pohľadu môže priamo vychádzať aj editačný pohľad. To je prípad, keď napríklad admin informačného systému bude chcieť zmeniť obsah detailného pohľadu. Editačný pohľad si môžeme predstaviť v podobe formulára.

Editačný pohľad obsahuje rôzne komponenty s možnosťou interakcie (3.1), ktoré korešpondujú dátovým položkám z DTO modelu FestivalDetailDto 4.9. Obrázok 4.10 prezentuje návrh jednoduchého detailného pohľadu (naľavo) a návrh editačného pohľadu (napravo) vo forme formulára s tlačidlom uloženia zmien.

```
1  public class FestivalDetailDto {
2      public string Name { get; set; }
3      public string LogoUri { get; set; }
4      public MusicGenre Genre { get; set; }
5      public string Country { get; set; }
6      public string City { get; set; }
7      public string Description { get; set; }
8      public DateTime StartTime { get; set; }
9      public int Capacity { get; set; }
10     public decimal Price { get; set; }
11 }
```

Obr. 4.9: Štruktúra DTO objektu FestivalDetailDto, ktorý predstavuje modelovú triedu pre detailný a editačný pohľad.

Obr. 4.10: Návrh detailného pohľadu, ktorý obsahuje informačné komponenty je zobrazený vľavo. Návrh editačného pohľadu, ktorý obsahuje komponenty na interakciu je zobrazený vpravo.

Prihlasovací a registračný pohľad

Prihlasovací pohľad obsahuje zvyčajne dve textové polia, kde užívateľ zadá svoje prihlasovacie údaje. Textové pole pre heslo je chránené pred zobrazením holého textu. Pohľad tiež obsahuje tlačidlo, ktoré vykoná akciu prihlásenia. Je vhodné zahrnúť odkaz na registračný pohľad v prípade neregistrovaného užívateľa. Kód 4.11 predstavuje DTO objekt pre prihlasovací pohľad, ktorý je navrhnutý na obrázku 4.13 naľavo.

Registračný pohľad je špecifikovaný DTO objektom 4.12. Podobne ako pri prihlasovacom pohľade obsahuje textové polia pre prihlasovacie meno a heslo. Pri vytváraní hesla sa môže použiť UI návrhový vzor Password strength meter (3.2), vďaka ktorému prinútime užívateľa vytvoriť silné heslo. Registračný pohľad popisuje DTO objekt UserRegisterDto 4.12. UserRegisterDto objekt definuje ďalšie polia, ktoré sú požadované pre úspešnú registráciu. Preto je potrebné aby vložené dáta boli validované a v prípade nevalidného vstupu bol užívateľ o chybe stručne informovaný. Návrh registračné pohľadu je na obrázku 4.13 v pravo.

```
1  public class UserLoginDto {
2      public string Username { get; set; }
3      public string Password { get; set; }
4  }
```

Obr. 4.11: Štruktúra DTO objektu UserLoginDto, ktorý predstavuje modelovú triedu pre prihlasovací pohľad.

```
1  public class UserRegisterDto {
2      public string Name { get; set; }
3      public string Surname { get; set; }
4      public string Country { get; set; }
5      public string City { get; set; }
6      public string Email { get; set; }
7      public string Username { get; set; }
8      public string Password { get; set; }
9  }
```

Obr. 4.12: Štruktúra DTO objektu UserRegisterDto, ktorý predstavuje modelovú triedu pre registračný pohľad.

Obr. 4.13: Základný návrh prihlasovacieho (vľavo) a registračného pohľadu (vpravo), ktoré sa skladajú s komponentov interakcie.

Kapitola 5

Implementované knižnice

Táto kapitola popisuje dve implementované knižnice, ktoré spolu tvoria API pre používanie frameworku. Prvá sekcia popisuje knižnicu Blazor.GenUI.Reflection, ktorá predstavuje základ pre funkcionalitu frameworku s využívaním reflexie. Druhá sekcia popisuje knižnicu Blazor.GenUI.Components, ktorá poskytuje samotné Blazor komponenty pre generovanie užívateľského rozhrania.

5.1 Knižnica reflexie

Knižnica Blazor.GenUI.Reflection implementuje základne dátové typy pre možnosť prispôsobenia UI. Obsahuje implementáciu rozhraní či už pomocných alebo pre typy elementov spomenutých pri návrhu frameworku. Tiež implementuje dynamické vyhľadávanie typov pohľadov a používa reflexiu na zaistenie základne funkcionality frameworku.

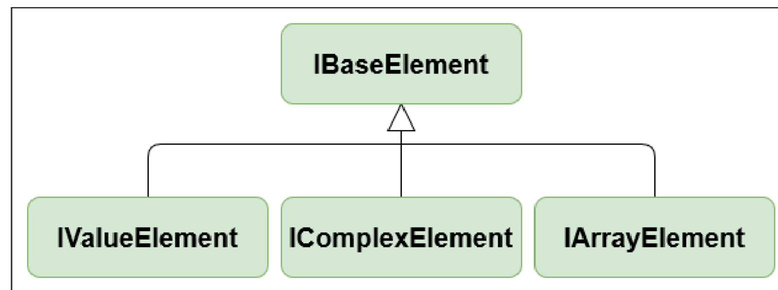
Rozhrania typov elementov

Vrámci implementácie, typy elementov sú identifikované pomocou rozhraní. Rozhrania reprezentujú jednotlivé typy (4.1) v podobe abstraktnej štruktúry. Štruktúra popisuje, aké dáta, metadáta o objekte a metódy jednotlivé inštancie implementujúce tieto rozhrania obsahujú. Súvis medzi jednotlivými rozhraniami je ukázaný na obrázku 5.1.

Implementovaný framework obsahuje štyri rozhrania typov elementov:

- **IBaseElement** — reprezentuje bázový typ pre ostatné rozhrania. Obsahuje základne informácie o elemente, ako sú napríklad jeho meno a typ. Vďaka typu IBaseElement sme schopný vytvoriť list štruktúry vstupného objektu — tzv. deti objektu, ktorý následne generátor UI prechádza a vkladá jednotlivé typy do korešpondujúcich pohľadov.
- **IValueElement** — reprezentuje primitívny element objektu. Obsahuje konkrétnu hodnotu elementu a udalosť na zachytenie v prípade zmeny hodnoty, ktorú užívateľ môže vykonať v UI.
- **IComplexElement** — reprezentuje komplexný element objektu, ktorý v sebe enkapsuluje v podobe dátové typu Object. Poskytuje metódu na získanie štruktúry enkapsulovaného objektu. Návratová hodnota metódy je pole, ktoré obsahuje položky typu IBaseElement.

- **IArrayElement** — reprezentuje pole. Obsahuje metódu na získanie položiek v poli, kde typ položky je IBaseElement.



Obr. 5.1: Diagram dedičnosti, ktorý ukazuje hierarchiu implementovaných rozhraní typov elementov.

Dynamické vyhľadávanie a načítavanie pohľadov

Pre poskytnutie príslušného UI pre typ elementu slúžia ako šablóny implementované Blazor komponenty (5.2). Keďže UI je generované za behu programu, framework musí byť schopný nájsť dynamicky príslušnú UI komponentu vrámci assembly, inicializovať a načítať ju s príslušnými hodnotami. Tento systém je implementovaný v triede `ComponentService` a na identifikáciu komponentov slúži rozhranie `IRenderableComponent`.

Trieda `ComponentService` obsahuje sadu metód pre zaistenie požadovanej funkcionality vyhľadávania komponentov. Obsahuje metódy na načítanie programovej assembly, vyhľadanie komponentov na základe implementovaného rozhrania a získania inštancie komponentov na základe mena komponentu. Tiež obsahuje pole s typmi načítaných komponentov z assembly. Požadovaný typ komponentu sa na základe jeho mena nájde v načítanom poli typov a vytvorí sa inštancia daného komponentu. Práve do tejto inštancie komponentu je neskôr predaná hodnota (ak je to primitívny typ), ktorá sa následne vygeneruje.

Dynamické vyhľadávanie a načítavanie pohľadov a následné ich vygenerovanie zjednodušene funguje minimálne v týchto bodoch:

- implementovanie rozhrania `IRenderableComponent` požadujúcim pohľadom (Blazor komponenty),
- na základe rozhrania vyhľadanie pohľadov vrámci assembly,
- uloženie typov pohľadov do poľa,
- získanie typu pohľadu na základe jeho mena,
- vytvorenie inštancie pohľadu,
- predanie hodnoty do inštancie,
- vygenerovanie pohľadu s aktuálnou hodnotou.

Pre poskytnutie layoutov a šablón pre jednotlivé inštancie pohľadov slúžia tzv. providery. Providery sú triedy, ktoré v sebe obsahujú dátový typ Slovník s informáciami o type layoutov/šablón a ich umiestnení vrámci programovej assembly. Funkciu provideru môžme zhrnúť v nasledujúcom v príklade:

Keď sa užívateľ rozhodne použiť existujúcu šablónu pre jeho vstupný objekt, ako parameter špecifikuje typ šablóny. Následne vrámci frameworku sa tento typ predá provideru, ktorý poskytne informácie o type korešpondujúceho pohľadu a jeho assembly. Tieto informácie sa použijú pre vytvorenie inštancie pohľadu a následne vygenerovanie UI s predanými dátami.

Použitie reflexie

Reflexia prestavuje významnú rolu pri celkovej funkčnosti frameworku. Reflexia je vlastnosť, pomocou ktorej sme schopní pristúpiť k hodnotám a typom elementov za behu programu. Vďaka používaniu reflexie framework zaisťuje:

- získanie štruktúry vstupného objektu a hodnôt jeho elementov,
- aktualizovanie pôvodných hodnôt vo vstupnom objekte.

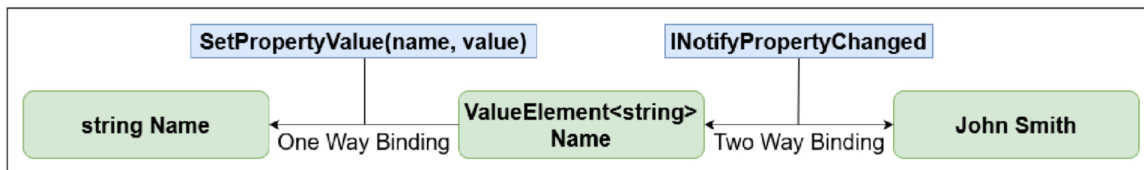
Získanie štruktúry objektu

Na začiatku po predaní vstupného objektu Blazor komponente na generovanie sa vytvorí inštancia typu `ComplexElement`, ktorý obsahuje enkapsulovaný vstupný objekt. Typ `ComplexElement` v sebe obsahuje metódu `GetChildren()`, ktorá z enkapsulovaného objektu pomocou reflexie získa názvy, typy a hodnoty jeho elementov. Na základe typu elementov sú vytvorené zodpovedajúce inštancie typu `ValueElement`, ktorým sú predané práve hodnoty získané pomocou reflexie. Tieto inštancie sa ukladajú do listu v podobe bázového rozhrania `IBaseElement`. Následne vstupná Blazor komponenta iteruje cez vrátený list, identifikuje jednotlivé typy a vytvára zodpovedajúce pohľady.

Aktualizovanie pôvodných hodnôt

Jedna z fundamentálnych vlastností frameworku je zaistenie správneho prepojenia pôvodných hodnôt s vygenerovanými UI hodnotami. Dôležité je, že ak sa zmení hodnota v UI elemente, musí sa zmeniť hodnota aj v inštancii ValueElement a aj v zdroji tejto hodnoty, čiže v pôvodnom objekte. Obrázok 5.2 predstavuje ako sú hodnoty spolu prepojené a ako prebieha aktualizovanie hodnôt. Vygenerovaná UI hodnota je previazaná s inštanciou typu ValueElement pomocou two way bindingu. To znamená, ak sa zmení hodnota v UI, automaticky sa zmení hodnota aj v inštancii a aj opačne. Keď nastane táto zmena, pomocou implementovaného rozhrania `INotifyPropertyChanged` sa vyvolá udalosť, ktorá oznámi, že nastala zmena hodnoty a je potrebné aktualizovať pôvodnú hodnotu vo vstupnom objekte. Nad vstupným objektom sa použije reflexia, kde sa pomocou názvu vyhľadá príslušný element nesúci pôvodnú hodnotu, ktorá sa prepíše novou hodnotou zadanou v UI. Na to práve slúží metóda s názvom `SetPropertyValue`, ktorá pochádza z externej knižnice FasterFlect[1]. Knižnica FasterFlect poskytuje rozširujúce metódy, umožňujúce používať reflexiu nad C# objektami jednoduchšie, intuitívnejšie a hlavne s lepším výkonom.

[1] FasterFlect: https://github.com/buunguyen/fasterflect



Obr. 5.2: Ukážka použitia reflexie pri aktualizácií pôvodnej hodnoty na hodnotu zadanej v UI.

5.2 Knižnica komponentov

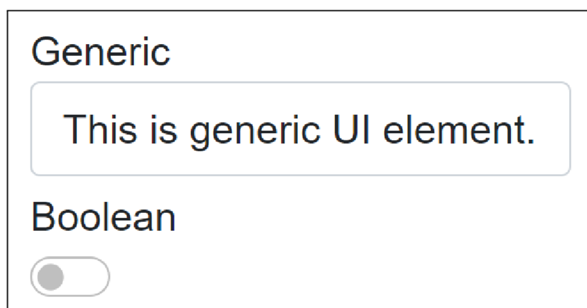
Knižnica Blazor.GenUI.Components je Razor knižnica, ktorá implementuje Blazor komponenty korešpondujúce pre jednotlivé typy elementov zo vstupného objektu. Poskytuje pohľady pre primitívne a komplexné typy. Tiež poskytuje pohľady pre podporované layouty a obsahuje v sebe podporované Bootstrap štýly v podobe CSS súborov. Táto knižnica v sebe implementuje aj dve Blazor komponenty generovania, ktoré slúžia ako základné komponenty pre dynamické generovanie pohľadov.

Komponenty primitívnych typov

Blazor komponenty primitívnych typov predstavujú konkrétne implementované pohľady pre primitívne typy. Tieto komponenty implementujú rozhranie `IRenderableComponent`, vďaka ktorému je ich framework schopný vyhľadať. Sú rozdelené podľa prezentačného typu. Keďže primitívnych typov je v jazyku C# viacero, nie je vhodné pre každý tento typ implementovať jeden pohľad. Tým by sa znížila efektivita frameworku a celková prehľadnosť kódu. Platí to hlavne pre typ string a pre numerické typy, ktorých je väčší počet (int, uint, double, byte...). Preto je vhodnejšie použiť možnosť generických Blazor komponentov.

Pri inštanciovaní primitívnej generickej Blazor komponenty je jej predaný parameter v podobe typu hodnoty, ktorú bude niesť. Typ je získaný pomocou reflexie zo vstupného objektu. Vďaka tomuto môže slúžiť jedna generická Blazor komponenta ako jeden pohľad pre rôzne typy.

Avšak nie pre všetky typy je vhodné použiť generickú komponentu. Príkladom je typ Boolean, pre ktorý je vhodnejšie vytvoriť vlastnú komponentu z dôvodu zobrazenia hodnoty do elementu checkbox. Rozdiel medzi týmito komponentami ukazuje obrázok 5.3.



Obr. 5.3: Príklad vygenerovaných elementov, ktorý ukazuje rozdiel medzi generickou UI komponentou a komponentou typu Boolean.

Ďalším príkladom je primitívna komponenta pre časové hodnoty v podobe typu DateTime. Typ DateTime môže v sebe niesť viacero hodnôt, pre ktoré je vhodné poskytnúť vlastný pohľad. Môžu to byť hodnoty v podobe:

- Dátum (Date),
- Čas (TimeOfDay),
- Dátum a Čas (DateTime).

Na základe typu nesúcej hodnoty výsledný UI element poskytuje vyberače časových hodnôt (date picker, time picker, datetime picker). Namiesto ručného zadávania časových hodnôt si užívateľ pomocou vyberačov vykliká požadovanú hodnotu v správnom formáte. Na špecifikovanie formátu časovej hodnoty je možné použiť dátovú anotáciu Date s typom nesúcej hodnoty.

Pre hodnoty typu Enum je tiež vytvorený vlastný pohľad. Na zobrazovanie hodnôt tohoto typu sa používa buď UI elementa selectbox alebo radio buttons.

Komponenty komplexných typov

Komponenty komplexných typov predstavujú UI šablóny implementované vrámci frameworku ako Blazor komponenty. Každej tejto šablóne korešponduje jej model dát, ktoré zobrazuje. Ako kontext týmto komponentom sa predá jej korešpondujúci objekt, ktorý je k dispozícii vrámci frameworku. Užívateľ musí zaistiť vytvorenie inštancie modelovej triedy a jej naplnenie dátami. Následne framework nebude prechádzať štruktúru rekurzívne, ale hodnoty sa priamo vygenerujú do špecifikovanej šablóny. Framework obsahuje zatiaľ tieto implementované komponenty v podobe pohľad:

- Detailný pohľad
- Prihlasovací pohľad

Komponenty generovania

Komponenty generovania sú verejne dostupné Blazor komponenty, ktoré slúžia práve pre generovanie pohľadov. Užívateľ im predá kontext (vstupný objekt alebo pole), prípadne parametre a na základe toho sa vygeneruje užívateľské rozhranie. Framework implementuje presne dve takéto komponenty:

- komponenta RenderableContentControl,
- komponenta RenderableContentListControl.

RenderableContentControl

RenderableContentControl predstavuje hlavnú komponentu kde je implementovaná logika generovania (4.1). Skladá sa z dvoch súborov, kde jeden súbor slúži ako code behind[2], kde sú špecifikované parametre komponentu (možnosti prispôsobenia), pomocné funkcie a inicializačné udalosti. Druhý súbor predstavuje pohľadovú časť, kde sú implementované funkcie, ktoré sa postupne navzájom volajú (aj rekurzívne) a spoločne vytvárajú navrhnutú logiku generovania.

[2]Code behind: technika, kde kód pre UI a kód pre logiku sú uložené v samostatných súboroch.

Tieto funkcie z pohľadovej časti vracajú delegát typu `RenderFragment`. Zjednodušene, pri invokácii delegátu sa vytvorí HTML kód ktorý určuje implementácia danej funkcie. Vďaka tomu môžeme vrámci takýchto funkcií používať mix C# kódu, HTML kódu a aj iných Blazor komponentov.

Jedna z hlavných takýchto funkcií je funkcia `RenderChildren`, v ktorej sa prechádza pole získaných elementov vstupného komplexného objektu. Následne na základe typu elementu sa volajú tzv. generátory, ktoré pomocou `RenderTreeBuilder` vytvoria konkrétne implementované pohľady, či sú to už primitívne Blazor komponenty alebo komplexné šablóny.

Inštancie primitívnych Blazor komponentov sa získavajú pomocou tzv. lokátorov. Lokátory (obrázok 5.4) sú pomocné funkcie, ktorým sa ako parameter predá typ primitívneho elementu a prezentačný typ. Na základe parametrov sa vytvorí názov primitívnej komponenty, ktorá sa má použiť. Pomocou dynamického vyhľadávania komponentov sa táto komponenta získa a vytvorí sa. Získavanie a vytvorenie inštancií komplexných typov funguje prakticky rovnako, až na ten rozdiel, že namiesto dynamického vyhľadávania komponentov sú použité providery (5.1).

```
1 public IRenderableComponent ViewBaseLocatorBuilder(string primitiveTypeName,
2 PresentationType presentationType)
3 {
4     var buildedComponentName =
5                     $"Component{primitiveTypeName}{presentationType}View";
6     return ComponentService.GetComponent(buildedComponentName);
7
8 }
```

Obr. 5.4: Príklad funkcie lokátor, ktorá na základe názvu typu elementu a prezentačného typu vyhľadá a vráti inštanciu korešpondujúceho pohľadu.

RenderableContentListControl

RenderableContentListControl je Blazor komponenta určená na generovanie pohľadov z poľa. Kontext komponenty predstavuje pole, ktoré obsahuje komplexné typy. Táto komponenta nepodporuje generovanie vnorených komponentov, pretože pre účel tejto komponenty by to nedávalo zmysel. Momentálne táto komponenta poskytuje 3 pohľady, do ktorých je možné generovať pole:

- **Listový pohľad**(ListGroupLayout) — umožňuje generovať list položiek s menom s možnosťou prekliku na cestu, ktorá je určená zadanými parametrami.
- **Tabuľkový pohľad** (TableLayout) — umožňuje generovať obsah komplexných elementov do tabuľky s možnosťou špecifikácie jej štruktúry.
- **Záložkový pohľad** (TabLayout) – umožňuje generovať obsah komplexných elementov do záložiek. Názov záložky môže byť špecifikovaný parametrom.

Kapitola 6

Testovanie

Kapitola sa zaoberá overovaním funkčnosti frameworku, jeho integráciou do existujúcej Blazor aplikácie a jeho používaním. Tiež sa tu výsledný framework porovnáva s existujúcimi aplikáciami zo sekcie 2.4 a analyzuje sa spätná väzba komunity s možnými rozšíreniami.

Framework sa vyvíjal cez nástroj Visual Studio 2019 v prostredí .NET 5. Popri vývoji sa jednotlivé časti implementácie overovali jednotkovými testami. Na testovanie pomocou jednotkových testov slúži nástroj XUnit [1] vo verzii 2.4.1.

6.1 Jednotkové testy

Pri vývoji bol framework testovaný pomocou jednotkových testov obsiahnutých v knižnici `BlazorGenUI.Tests`. Niektoré z týchto testov používajú komplexné objekty, vytvorené špeciálne pre potreby testovania. Jednotkové testy sú rozdelené do kategórií podľa zamerania testovania na:

- základné testy,
- testy reflexie,
- testy vlastností,
- testy generovania.

Metodika testov

Testy sú písane pomocou metodiky **Arrange-Act-Assert**[2]. Každý jednotkový test je rozdelený do troch častí:

- **Arrange** — Splnenie potrebných podmienok pred testovaním, pripravenie inštancií testovania a vstupov.
- **Act** — Akt testovania nad objektom alebo testovanou metódou.
- **Assert** — Vyhodnotenie a porovnanie s očakávaným výsledkom.

Vďaka použitiu spomenutej metodiky sú jednotkové testy lepšie štrukturované a jednoduchšie na pochopenie. Príklad použitia testovacej metodiky je popísany na obrázku 6.1.

[1] XUnit: https://xunit.net/
[2] Arrange-Act-Assert: http://wiki.c2.com/?ArrangeActAssert

Testovacie objekty

Pre potreby testovania som vytvoril sadu testovacích objektov, obsahujúcu:

- objekt primitívnych typov,
- objekt komplexných typov,
- objekt polí,
- objekt mixu typov,
- objekt elementov s dátovými anotáciami.

V triede `BlazorGenUITestsFixture` sú vytvorené inštancie týchto objektov. Následne je táto trieda použitá ako kontext pre jednotlivé testovacie prípady. Vďaka tomu majú každé testy aj prístup ku inštanciám objektov testovaného frameworku a ich metódam.

Základne testy

Základne testy obsahuje trieda `FundmentalTests`. Obsahuje sadu testov pre overenie funkcionality vyhľadávania typov komponentov vrámci assembly. Testuje metódy obsiahnuté v triede `ComponentService` pre získavanie komponentov z assembly na základe mena. Tiež testuje funkcionalitu lokátorov komponentov, ktoré na základe prezentačného typu a typu hodnoty elementu vrátia inštanciu korešpondujúceho pohľadu. Príklad jednotkového testu lokátora popisuje obrázok 6.1.

```
1  [Fact]
2  public void Locator_ViewBaseLocatorControl_IsNotNull()
3  {
4      //Arrange
5      var typeName = "Boolean";
6      PresentationType presentation = PresentationType.Control;
7      //Act
8      var component =
9          _fixture.RenderableContent.ViewBaseLocatorBuilder(typeName,
10             presentation);
11     //Assert
12     Assert.NotNull(component);
13 }
```

Obr. 6.1: Kód ukazujúci použitie testovacej metodiky Arrange-Act-Assert pri testovaní lokátora komponentov.

Testy reflexie

Testy reflexie obsahuje trieda `ReflectionTests`. V tejto triede sa overuje funkcionalita pri použití reflexie. Overuje sa implementovaná logika získavania elementov z komplexného objektu dynamicky za behu programu. Testy používajú testovacie objekty obsahujúce rôzne typy elementov. Úspešnosť testu sa vyhodnocuje na základe porovnávania počtov vrátených elementov typu `IBaseElement` s pôvodnými elementami v zdrojovom objekte. Tiež sa testuje aktualizovanie pôvodných hodnôt implementované pomocou reflexie. Očakáva sa, že keď sa zmení hodnota v type `IValueElement` tak sa zmení aj pôvodná hodnota.

Testy vlastností

Testy vlastností obsahuje trieda `FeatureTests`. Testy overujú funkčnosť možností prispôsobenia. Testuje sa aplikovanie layoutov, poradia, ignorovania, typov dátumov, nastavenia vlastných názvov a špecifikovanie obrázkov. Tieto možnosti sú buď zadané ako parametere alebo v podobe dátových anotácií.

Testy generovania

Testy generovania obsahuje trieda `RenderableContentTests`. Testovacie objekty sa predajú renderovacej metóde `RenderComponent`, ktorá vracia delegát typu `RenderFragment`. Po invokácií delegáta prebieha inšpekcia rámcov buildera UI `RenderTreeBuilder`. Rámce predstavujú jednotlivé fragmenty, z ktorých Blazor framework vytvorí UI. Testy v týchto fragmentoch hľadajú vygenerované elementy, ktorých počet potom porovnávajú s počtom elementov v testovacom objekte. Ak sa počty zhodujú, test je úspešný a z elementov sa úspešne vytvorili rámce, z ktorých vznikne výsledné UI.

6.2 Integrácia do Blazor aplikácie

Najjednoduchší spôsob ako integrovať implementované knižnice do existujúcej Blazor aplikácie je pomocou NuGet balíčka BlazorGenUI [3]. Tento NuGet balíček v sebe obsahuje implementovanú knižnicu BlazorGenUI.Components, s referenciou na knižnicu reflexie a ostatné použité knižnice. Tiež v sebe obsahuje podporované Bootstrap štýly. Po nainštalovaní balíčka je potrebné urobiť ďalšie kroky, ktoré zaistia správnu funkcionalitu. Musí sa pridať referencia na Bootstrap knižnicu, zaregistrovať implementované služby vrámci DI kontajnera a pridať názvy menných priestorov.

Pridanie Bootstrap knižnice

Pre podporu štýlovania je potrebné pridať referenciu na Bootstrap CSS súbory obsiahnuté vo frameworku. Pri aplikácií Blazor WebAssembly sa pridá referencia do súboru `index.html`. Pri aplikácií Blazor Server sa referencia pridá do súboru `_Host.cshmtl`. Užívateľ sa môže rozhodnúť, či použije niektoré z preddefinovaných Bootstrap štýlov obsiahnutých vrámci frameworku alebo si pridá vlastnú referenciu. S dôvodu kompatibility odporúčam používať preddefinované štýly Bootstrapu. Príklad, ako takýto preddefinovaný štýl referencovať je zobrazený na obrázku 6.2.

```
1  <link href="_content/BlazorGenUI/bootstrap-primary.min.css" rel="stylesheet"/>
```

Obr. 6.2: Ukážka kódu, pomocou ktorého sa môže referencovať primárny Bootstrap štýl obsiahnutý vrámci frameworku.

Pridanie menných priestorov a registrácia služieb

Po pridaní NuGet balíčka musíme zabezpečiť vrámci Blazor aplikácie prístup k jednotlivým komponentom, dátovým typom a modelom frameworku. To zabezpečíme pridaním menných priestorov pomocou príkazu `@using` do súboru `_Imports.razor`. Vďaka tomuto bude možné

[3] Knižnica BlazorGenUI je verejne dostupná na adrese: https://www.nuget.org/packages/BlazorGenUI/

pristupovať k frameworku naprieč všetkými Blazor komponentami v aplikácií. Obrázok 6.3 ukazuje pridanie všetkých potrebných menných priestorov pre používanie frameworku.

```
1  @using BlazorGenUI.Components.Renderable
2  @using BlazorGenUI.Reflection.Enums
3  @using BlazorGenUI.Reflection.Models
```

Obr. 6.3: Potrebné menné priestory, ktoré sa musia pridať do súboru `_Imports.razor` pre prístup ku komponentom a ostatným potrebným súborom.

Ďalej je potrebné zabezpečiť registráciu služieb používaných vo frameworku do DI [4] kontajnera. Na to framework obsahuje rozširujúcu metódu `AddBlazorGenUIServices()`. Túto metódu treba zaregistrovať pri štarte aplikácie. Avšak pre jednotlivé typy Blazor aplikácií sa táto registrácia líši. Pre Blazor Server je potrebné zaregistrovať služby v metóde `ConfigureServices` v súbore `Startup.cs`. Registráciu pre Blazor server demonštruje obrázok 6.4. Naopak pre Blazor WebAssembly je potrebné zaregistrovať služby v metóde `Main` v súbore `Program.cs`. Túto skutočnosť demonštruje obrázok 6.5.

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3      .
4      services.AddBlazorGenUIServices();
5      .
6  }
```

Obr. 6.4: Registrácia služieb pre aplikáciu typu Blazor Server v súbore `Startup.cs`.

```
1  public static async Task Main(string[] args)
2  {
3      .
4      builder.Services.AddBlazorGenUIServices();
5      .
6  }
```

Obr. 6.5: Registrácia služieb pre aplikáciu typu Blazor WebAssembly v súbore `Program.cs`.

6.3 Použitie komponentov generovania

Po úspešnej integrácií frameworku do Blazor aplikácie užívateľ získa prístup ku generovacím komponentom (5.2). Ako bolo spomenuté tieto komponenty obsahujú parametre, ktoré slúžia pre prispôsobenie vygenerovaných pohľadov. Táto sekcia obsahuje príklady použitia komponentov generovania a ich parametrov. Avšak nepokryjem všetky možnosti, pretože by to bolo moc rozsiahle. Pre viacej príkladov použitia frameworku slúži demonštračná aplikácia. Pre zjednodušenie, každý komponent bude mať kontext v podobe vstupného objektu Person, ktorého štruktúra je ukázaná na obrázku 6.6. Príklady výsledného vygenerovaného UI sú dostupné v publikovanej demonštračnej aplikácii [5].

[4] DI = Dependency Injection
[5] BlazorGenUI demo: https://specter-13.github.io/

```
1 public class Person
2 {
3     public string Name { get; set;}
4     public string Surname { get; set;}
5     public int Age { get; set;}
6     public string City { get; set;}
7     public string AvatarUri { get; set;}
8 }
```

Obr. 6.6: Štruktúra demonštračného objektu, ktorý sa používa ako kontext v príkladoch použitia komponentov generovania.

Základné použitie

Pre základné vygenerovanie UI stačí predať komponentu kontext v podobe vstupného objektu, prípadne špecifikovať prezentačný typ. Na obrázku 6.7 je zobrazené, ako sa parametru Context predá inštancia objektu a ako sa pomocou typu PresentationType vyberie požadovaný prezentačný typ.

```
1 <RenderableContentControl Context="@Person"
2                           Presentation="PresentationType.Control"/>
```

Obr. 6.7: Príklad základného použitia komponentu generovania, ktorý vygeneruje elementy pod seba s možnosťou modifikovania hodnôt.

Použitie možností prispôsobenia

Komponent generovania obsahuje parametre na prispôsobenie výsledného pohľadu. Možnosti ako je možné pohľad prispôsobiť sú popísané v sekcii 4.2. Na obrázku 6.8 je ukázané, akým spôsobom sa komponentu predávajú parametre prispôsobenia. Pre určenie rozloženia elementov slúži parameter Layout, ktorému je predaná konkrétna hodnota z typu LayouTypes. Parameter PictureFields slúži na určenie elementov, ktoré sa majú vygenerovať ako obrázky. Parametru je predaný typ string, kde sú názvy elementov, ktoré predstavujú obrázky, rozdelené čiarkami. Podobne je implementovaný parameter IgnoredFields, ktorý špecifikuje elementy, ktoré sa budú ignorovať.

Parameter Order slúži na zadanie poradia elementov. Požaduje inštanciu typu slovník. Príklad vytvorenia inštancie slovníka je ukázaný na obrázku 4.3. Podobne sa používa parameter Labels, ktorý slúži na nastavenie vlastných názvov pre vygenerované elementy. Tiež požaduje inštanciu typu slovník, ktorého podoba je ukázaná na príklade 4.4.

```
1 <RenderableContentControl Context="@Person"
2                           Layout="LayoutTypes.Grid2"
3                           PictureFields="AvatarUri"
4                           IgnoredFields="City, Age"
5                           Order="@order"
6                           Labels="@labels"/>
```

Obr. 6.8: Príklad použitia komponentu generovania s parametrami, ktoré slúžia na prispôsobenie výsledného pohľadu.

Použitie šablón

Na špecifikovanie šablón slúži parameter Template. Požaduje hodnotu typu Template, ktorá určuje použitú šablónu. Avšak ako kontext sa už nepredáva vstupný objekt, ale modelový objekt korešpondujúci každej šablóne. Preto užívateľ musí zaistiť správne premapovanie dát medzi zdrojovým objektom a objektom danej šablóny. Na obrázku 6.9 je ukázané použitie komponentu na generovania dát do šablóny DetailView. Ako kontext je predaná inštancia objektu typu DetailTemplateModel (model obsiahnutý vo frameworku), v ktorom sú predané dáta zo zdrojového objektu (v tomto príklade to môže byť objekt Person).

```
1  <RenderableContentControl Context="@Detail"
2                            Template="Template.DetailView"/>
```

Obr. 6.9: Príklad použitia komponentu generovania, kde je použitá šablóna detailného pohľadu a kontext predstavuje modelový objekt šablóny.

Použitie atribútov vrámci objektu

Ak užívateľ má prístup ku štruktúre objektu môže použiť atribúty na prispôsobenie elementov. Na obrázku 6.10 je ukázaný príklad použitia atribútov. Idea prispôsobenia je rovnaká ako pri použití parametrov. Napríklad na špecifikovanie layoutu sa používa rovnaký typ LayoutTypes. Pomocou atribútu RadioButtonsEnum je možné typ Enum vygenerovať ako súbor radio tlačidiel. Tiež je možné prispôsobiť typ zobrazovaného dátumu pomocou atribútu Date a použiť atribút Ignore na ignorovanie elementov.

```
1  [Layout(LayoutTypes.Wrap)]
2  public class PersonAttributes
3  {
4      public string Name { get; set;}
5      [RadioButtonsEnum]
6      public Gender Gender {get; set;}
7      public int Age { get; set;}
8      [LabelAttribute("Home city")]
9      public string City { get; set;}
10     [Date(DateTypes.Date)]
11     public DateTime Born { get; set;}
12     [RenderIgnore]
13     public string AvatarUri { get; set;}
14 }
```

Obr. 6.10: Príklad objektu, v ktorom sú použité atribúty na prispôsobenie vygenerovaného UI.

Použitie komponentu pre generovanie polí

Komponenta RenderableContentListControl slúži na generovanie polí. Kontext v tomto prípade predstavuje pole, ktoré obsahuje zložené objekty. Momentálne framework podporuje 3 typy pohľadov polí, ktoré je možné vygenerovať. Tieto pohľady sa špecifikujú pomocou parametra Layout a typu ArrayLayout. Obrázok 6.11 predstavuje použitie komponentu pre generovanie polí, so špecifikovaním tabuľkového pohľadu.

```
1  <RenderableContentListControl ContextList="forecasts"
2                               Layout="ArrayLayout.Table"/>
```

Obr. 6.11: Príklad použitia komponentu generovania polí, kde je špecifikovaný tabuľkový pohľad.

6.4 Porovnanie s existujúcimi aplikáciami

Z môjho výskumu existujúcich riešení vyplýva, že pre Blazor dedikovaný open-source framework na generovanie UI momentálne neexistuje. Hlavná výhoda oproti existujúcim riešeniam spomenutým v sekcii 2.4 je taká, že môj framework je schopný generovať UI do webovej aplikácie. Inxton sa zameriava na generovanie UI do WPF a používa ako vstup špecifické objekty viazané na samotný framework. Oproti tomu moja implementácia sa zameriava na DTO objekty používané pri prenose dát medzi serverom a klientom. Môj framework poskytuje aj možnosť prispôsobenia pohľadov pomocou parametrov. Inxton výhradne používa na prispôsobenie atribúty.

ReflectionUI používa na prispôsobenie UI iný prístup. Používa WYSIWIG editor nad auto-generovaným UI zo vstupného objektu. Tento prístup môže mať výhodu, že poskytuje prepracovanejšie prispôsobenie UI. Avšak týmto sa zvyšuje zložitosť pri používaní frameworku.

Ďalší podstatný rozdiel medzi mojím frameworkom a ostatnými je rozdielna väzba s UI. Inxton a ReflectionUI podporujú obojsmernú väzbu medzi zdrojovým objektom a vygenerovanými UI elementami. Vďaka tomu zmena dát v zdrojovom objekte sa okamžite prejaví vo výslednej aplikácii. Toto môže byť užitočné pri vývoji real-time aplikácií. Na druhej strane sú tu otázky ohľadom výkonu aplikácie, keďže veľké množstvo aktualizácii UI môže degradovať responzivitu aplikácie. Môj framework aktuálne podporuje jednosmernú väzbu dát. To znamená, dáta v zdrojovom objekte sa zmenia iba keď sú zmenené vo vygenerovaných UI elementoch. Tento prístup pre DTO objekty je postačujúci, avšak pri ďalšom možnom rozšírení frameworku je vhodné zvážiť a implementovať aj obojsmernú väzbu dát. Podstatné rozdiely medzi jednotlivými frameworkami sú zhrnuté v tabuľke 6.1.

	BlazorGenUI	**Inxton**	**ReflectionUI**
použitie	web C# aplikácia	desktop WPF aplikácia	desktop Java aplikácia
prispôsobenie UI	parametre + atribúty	atribúty	WYSIWYG editor
technológie	C# + HTML + Javascript	C# + XAML	Java
väzba s UI	jednosmerná	obojsmerná	obojsmerná

Tabuľka 6.1: Porovnanie základných vlastností implementovaného frameworku BlazorGenUI s existujúcimi riešeniami.

6.5 Spätná väzba komunity a možné rozšírenia

Výsledný framework je verejne dostupný ako open-source v GitHub repozitári[6]. Po zverejnení kódu som vytvoril posty na sociálnej sieti Reddit do skupín s názvom r/Blazor a r/dotnet. Predstavil som komunite môj nápad a ideu za týmto frameworkom a zaujímal som sa o spätnú väzbu, či takýto framework má zmysel. Bol som prekvapený počtom reakcií, čo sa následne prenieslo aj na GitHub repozitár, ktorý momentálne má 40 hviezdičiek a vyše 1500 pozretí. Je to hlavne asi preto, že takýto open-source framework je prvý svojho druhu pre Blazor. Z reakcií vzišlo, že framework má potenciál hlavne pri formulároch. Formuláre sú často používané naprieč webovými aplikáciami a možnosť ich automaticky generovať z DTO objektu by mohlo značne byť pohodlnejšie a rýchlejšie. Preto jedno z možných rozšírení je implementovať validáciu dát. Jeden z ďalších nápadov bol rozšíriť framework o generovanie UI z tried v backende. To znamená podpora pre obsiahlejšie zložitejšie triedy. Tam by bolo dôležité sa zamyslieť nad tým, ako špecifikovať položky, ktoré sú obsiahnuté vrámci takéjto zložitejšej triedy. Avšak padli aj výhrady, či je prispôsobovanie UI pomocou parametrov a atribútov praktické. Je pravda, že pri požadovanej zložitejšej štruktúre UI môže tento systém prispôsobovania byť neprehľadný a ťažší na používanie. Avšak hlavný nápad bol generovať jednoduché a praktické UI s možnosťou prispôsobenia aj z enkapsulovaných DTO objektov, ku ktorým užívateľ nemá priamy prístup. Zložitejšiu štruktúru UI by bolo pravdepodobne vhodnejšie implementovať manuálne.

Nuget balíček BlazorGenUI vo verzii 1.0.0 má momentálne už vyše 60 stiahnutí, čo indikuje, že o výsledný framework je záujem a používa sa. BlazorGenUI sa objavil aj vo týždennom blogu BestOfBlazor[7] ako nový zaujímavý open-source projekt, či v článku v portáli InfoQ[8], ktorý zhromažďuje novinky a zaujímavosti o .NET platforme. Jedna z reakcií na Reddite bola, že užívateľ reálne použil implementovaný framework vo svojej aplikácií pre generovanie základných formulárov. Na základe tejto spätnej väzby je vhodné rozširovať a udržiavať implementovaný framework ďalej do budúcnosti.

[6]GitHub repozitár: https://github.com/Specter-13/blazorgenui
[7]BestOfBlazor blog: https://bestofblazor.blog/blazor-weekly-newsletter-8
[8]InfoQ článok: https://www.infoq.com/news/2021/05/dotnet-news-roundup-apr26-2021/

Kapitola 7

Záver

Cieľom tejto práce bolo vytvoriť framework, ktorý rozšíri existujúce webové aplikácie založené na technológií Blazor o možnosť automatického generovania určitých webových pohľadov. Výsledkom tejto práce je rozsiahly open-source projekt s názvom BlazorGenUI[1], ktorého balíčky sú online dostupné na stiahnutie.

V prvej časti mojej práce som sa zaoberal hlavne webovými technológiami od firmy Microsoft a vlastnosťami užívateľských rozhraní. Bližšie som špecifikoval technológiu Blazor a použitie DTO objektov pri vývoji webových aplikácií. Predstavil som tiež vybrané existujúce nástroje na generovanie užívateľského rozhrania. Následne prebehla analýza štruktúry a dizajnu existujúcej open-source aplikácie, kde som identifikoval jej zloženie a pohľady, ktoré by teoreticky mohli byť automaticky generované. Druhá časť práce sa venovala návrhu frameworku, následnej implementácií a testovaniu. Navrhol som princíp automatického generovania elementov z C# triedy do webových pohľadov a možnosti ich prispôsobenia. Tento návrh sa pretransformoval do dvoch implementovaných knižníc. Popri vývoji som overoval funkcionalitu frameworku pomocou jednotkových testov zameraných na špecifické časti. Nakoniec som predstavil postup integrácie frameworku do Blazor aplikácie, jeho použitie a porovnanie s existujúcimi riešeniami a analyzoval spätnú väzbu komunity a možné rozšírenia.

Implementovaný framework získal rozsiahly ohlas od komunity. Komunita ocenila ideu za týmto projektom, hlavne za účelom vytvárania jednoduchých, škálovateľných užívateľských rozhraní, jednoducho, efektívne a priamo z modelovej triedy. Zo spätnej väzby bolo zjavné, že takýto framework má hlavne potenciál pri vytváraní formulárov, kde v prípade pridania alebo odobrania elementu v modelovej triede sa užívateľské rozhranie automaticky aktualizuje bez ďalších potrebných zásahov. Tento framework bol aj zverejnený v článkoch zameraných na novinky ohľadom .NET prostredia, ako nový, zaujímavý open-source projekt.

Musím skonštatovať, že táto práca mi dala množstvo nových vedomostí a skúsenosti. Detailne som pochopil princíp fungovania technológie Blazor a získal skúsenosť ohľadom open-source projektov. Bol som veľmi príjemne prekvapený z reakcií komunity, ktorú som vôbec nečakal. Tieto všetky faktory ma len utvrdili, že má zmysel ďalej pokračovať vo vývoji projektu BlazorGenUI. Do budúcna sa framework môže rozšíriť o ďalšie webové pohľady. Môže sa pridať podpora pre validáciu dát či zložitejšie C# objekty, čo by následne ešte viacej rozšírilo použite frameworku.

[1] GitHub repozitár: https://github.com/Specter-13/blazorgenui

Literatúra

[1] ByteScout. *ASP.NET: History, Purpose, Versions* [online]. 2020 [cit. 2021-01-08]. Dostupné z: https://bytescout.com/blog/2014/01/aspnet-history-purpose-versions.html.

[2] Daniel Roth, L. L. *Introduction to ASP.NET Core Blazor* [online]. 2020 [cit. 2021-01-08]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0.

[3] Gollota, R. *10 key UX & UI Design elements that will provide the best positive user experience* [online]. 2020 [cit. 2020-12-22]. Dostupné z: https://uxdesign.cc/10-key-ux-ui-design-elements-that-will-provide-the-best-positive-user-experience-f94c6323046e.

[4] Interaction Design Foundation. *User Interface (UI) Design Patterns* [online]. [cit. 2020-12-22]. Dostupné z: https://www.interaction-design.org/literature/topics/UI-design-patterns.

[5] Inxton. *PLC's gateway to the PC world* [online]. 2021 [cit. 2021-4-26]. Dostupné z: https://inxton.com/.

[6] Kanjilal, J. *How to use Data Transfer Objects in ASP.NET Core 3.1* [online]. Infoworld Media Group, jún 2020 [cit. 2019-12-04]. Dostupné z: https://www.proquest.com/trade-journals/how-use-data-transfer-objects-asp-net-core-3-1/docview/2413232158/se-2?accountid=17115.

[7] Lerman, J. *Programming Entity Framework*. O'Reilly Media, Inc., 2010. ISBN 978-0-596-80726-9.

[8] Litvinavicius, T. *Exploring Blazor*. Apress, 2019. ISBN 978-1-4842-5446-2.

[9] Microsoft. *What is ASP.NET?* [online]. [cit. 2020-12-26]. Dostupné z: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet.

[10] Microsoft. *Introduction to SignalR* [online]. 2014 [cit. 2020-4-12]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr.

[11] ReflectionUI. *Java reflection-based GUI generator library* [online]. 2021 [cit. 2021-4-26]. Dostupné z: http://javacollection.net/reflectionui/.

[12] Rick Anderson, D. V. *Razor syntax reference for ASP.NET Corer* [online]. 2020 [cit. 2021-01-08]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-5.0.

[13] Roth, D., Fritz, J. a Southwick, T. *Blazor for ASP.NET Web Forms Developers* [online]. Redmond, Washington 98052-6399: Microsoft Developer Division, 2020 [cit. 2020-11-27]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/blazor-for-web-forms-developers/.

[14] Schäferhoff, N. *How to Choose Website Colour Schemes* [online]. 2019 [cit. 2020-12-26]. Dostupné z: https://websitesetup.org/website-color-schemes/.

[15] Spasojevic, M. *Blazor Introduction – Differences Between Blazor Server and Blazor WebAssembly, Pros, and Cons* [online]. 2020 [cit. 2021-01-14]. Dostupné z: https://code-maze.com/blazor-webassembly-introduction/.

[16] STANDS4 LLC.. *What does data binding mean?* [online]. 2021 [cit. 2021-01-08]. Dostupné z: https://www.definitions.net/definition/data+binding.

[17] Tiger Color. *Color Harmonies* [online]. 2015 [cit. 2020-12-26]. Dostupné z: https://www.tigercolor.com/color-lab/color-theory/color-harmonies.htm.

[18] WebAssembly. *WebAssembly* [online]. [cit. 2020-2-12]. Dostupné z: https://webassembly.org/.

[19] Web&go design s.r.o.. *Webdizajn: 9 zákonov pre tvorbu dokonalého webu* [online]. 2016 [cit. 2020-12-21]. Dostupné z: https://webandgo.sk/9-zakonov-tvorbu-dokonaleho-webu.

Príloha A

Obsah priloženého dátového média

- **blazorgenui/**
  - **src/** — zdrojové súbory implementovaných knižníc
  - **demo/** — zdrojové súbory demonštračnej aplikácie
  - **tests/** — zdrojové súbory testovacej knižnice
  - **assets/** — obrázky použité v README.md
  - **BlazorGenUI.sln** — súbor Visual Studio Solution
  - **README.md** — základné informácie o projekte v anglickom jazyku
- **latex/**
  - zdrojové súbory latex šablóny
- **text/**
  - text práce vo formáte pdf
- **README.txt**
  - základné informácie a pokyny ku spusteniu