Vývoj CMS systému ve frameworku Blazor

Jeroným Holer

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jeroným Holer**
Osobní číslo:	**A21002**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Vývoj CMS systému ve frameworku Blazor**
Téma práce anglicky:	**Development of a CMS System in the Blazor Framework**

Zásady pro vypracování

1. Popište současný stav problematiky webových frameworků.
2. Zaměřte se na framework Blazor a použití Server Side Renderingu.
3. Navrhněte řešení pro Content Management System s využitím frameworku Blazor.
4. Realizujte vývoj navrženého systému a popište jeho klíčové části.
5. Zhodnoťte dosažené výsledky a možnosti dalšího rozvoje aplikace.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. TAURIUS, Litvinavicius. *Exploring Blazor: Creating Hosted, Server-Side, and Client-Side Applications with C#*. Apress, 2019. ISBN 9781484254455.
2. HIMSCHOOT, Peter. *Microsoft Blazor: Building Web Applications in .NET 6 and Beyond*. 3rd ed. Apress, 2021. ISBN 9781484278444.
3. *ASP.NET Core Blazor | Microsoft Learn* [online]. c2023 [cit. 2023-11-05]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/
4. *Blazor | Build client web apps with C# | .NET* [online]. c2023 [cit. 2023-11-05]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor
5. *Blazor School* [online]. c2021–2023 [cit. 2023-11-05]. Dostupné z: https://blazorschool.com/
6. *Introduction to Razor Pages in ASP.NET Core | Microsoft Learn* [online]. c2023 [cit. 2023-11-05]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/

Vedoucí bakalářské práce: **doc. Ing. Radek Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

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

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne 3. 5. 2024 Jeroným Holer, v. r.

ABSTRAKT

Bakalářská práce se zabývá návrhem a vývojem systému pro server-side rendering webových stránek naplněných daty pomocí CMS systému webManager. Cílem práce je vytvoření systému a ukázková implementace webu na tento systém. Teoretická část se zabývá přehledem CMS systémů, platformou .NET CORE a frameworkem Blazor.
V praktické části je provedena analýza požadavků na vytvářený systém, popis klíčových části systému a ukázka implementace webové stránky na tento systém.

Klíčová slova: Blazor, .NET, Server-Side Rendering, CMS systém, C#

ABSTRACT

The bachelor's thesis deals with the design and development of a system for server-side rendering of websites filled with data using the webManager CMS system. The aim of the work is the creation of a system and a sample implementation of the website on this system. The theoretical part deals with an overview of CMS systems, the .NET CORE platform and the Blazor framework. In the practical part, an analysis of the requirements for the system being created, a description of the key parts of the system and a demonstration of the implementation of the website on this system is carried out.

Keywords: Blazor, .NET, Server-Side Rendering, CMS system, C#

Tímto bych rád poděkoval vedoucímu práce doc. Ing. Radkovi Šilhavému, Ph.D. a doc. Ing. Petrovi Šilhavému, Ph.D. za jejich odborné vedení a ochotu věnovat mi svůj čas.

Dále bych chtěl poděkovat mé skvělé ženě za nekonečnou podporu, která mi pomohla tuto práci dokončit.

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

OBSAH

ÚVOD

V práci se zabýváme tvorbou webů a jejich implementaci na vlastní CMS systém. Tento systém se jmenuje webManager. CMS systém webManager se skládá ze tří částí – API a klient pro admin rozhraní a systém pro server-side rendering webů naplněných pomocí dat zadaných v adminu. Tento systém nabízí přehlednou tvorbu webových stránek a jejich jednoduchou údržbu. Technologicky ale renderování webových stránek pokulhává, protože stále zůstává naprogramované v zastaralé technologii ASP 3.0. Mezi novější nástroje pro tvorbu webových stránek je framework Blazor, který s novou verzí .NET 8 přinesl novou možnost statického server-side renderingu. Díky tomu se Blazor stal nástrojem pro tvorbu statických webových stránek i interaktivních webových aplikací s architekturou klient-server.

Proč je vůbec dobré zabývat se vývojem komerčního CMS systémů? Je technicky možné vytvořit CMS systém ve frameworku Blazor? Tato práce nabídne odpověď na tyto otázky.

Bakalářská práce je rozdělena do dvou částí. První část se zabývá teorií. Napřed dojde k představení CMS systémů a porovnání open source a komerčních CMS systémů. Dojde k představení platformy .NET Core jako základ pro vývoj nového systému, a nakonec se práce bude zabývat frameworkem Blazor. Vývoj systému bude probíhat v tomto frameworku, proto dojde k jeho podrobnému představení a ukázání jeho možností. Ve druhé části se práce bude zabývat analýzou požadavků na CMS systém a následnou realizací na základě provedené analýzy. Hlavním východiskem této práce je návrh a implementace nové verze server-side renderingu CMS systému webManager. Na závěr bude představena ukázková implementace webové stránky na nový systém.

I. TEORETICKÁ ČÁST

1 PŘEHLED CMS SYSTÉMŮ

Systémy pro správu obsahu (CMS) jsou v dnešní době nepostradatelnou součástí tvorby webových stránek. V této kapitole si nejprve představíme klíčové vlastnosti CMS systémů a podávíme se, na jaké systémy můžeme v dnešní době narazit.

Systém pro správu obsahu umožňuje uživateli tvorbu webových stránek bez nutnosti znalostí web designu a vývojových jazyků jako jsou JavaScript, HTML a CSS. Dále umožňuje ukládat veškeré zdroje (obrázky, PDF soubory, videa) využívané v obsahu webu. [1]

Zde je seznam několika klíčových vlastností, které jsou důvodem pro hojné využívání CMS systémů [2]:

- Efektivita: Díky tomu, že systémy umožňují jednoduše vytvářet, upravovat a publikovat obsah bez technických znalostí nebo kódovacích dovedností, vedou k značné časové úspoře.
- Udržitelnost: Díky ukládání veškerého obsahu a médií a sledování historie jejich změn, nabízejí redakční systémy praktický systém pro údržbu webu po dlouhou dobu.
- Škálovatelnost: Díky schopnosti vytvářet nové stránky, obsah a rozšiřovat funkce webu, umožňují CMS systémy škálovat web v souladu s růstem jeho potřeb.
- Konzistence: Systémy pro správu obsahu pomáhají v celém webu udržovat jednotný design díky jasně definovaným fontům, barvám a stylům. [2]

1.1 Open source CMS

Mezi open source CMS řadíme například WordPress, Drupal a Joomla. Tyto systémy vyvíjí rozsáhlá komunita programátorů. Redakční systém si kdokoliv může stáhnout a upravit v souladu s jeho licencí. [3]

Open source CMS nesou několik výhod:

- Jelikož za vývojem těchto systémů stojí komunita programátorů, jsou tyto systémy pravidelně aktualizovány. Open source CMS s velkou uživatelskou komunitou jsou v tomto obzvláště efektivní. Pravidelná aktualizace vede k využívání nejnovějších technologií, což uživatelům přináší více možností při tvorbě jejich webových stránek.

- Open source CMS jsou zdarma a dostupné všem, bez licenčních poplatků. Z tohoto důvodu mohou být poměrně cenově dostupné.
- Open source platforma umožňuje jakýkoli zásah do zdrojového kódu. Je možné provádět změny bez omezení, a díky tomu jsou tyto systémy velice flexibilní.
- Velká komunita vývojářů umožňuje bez problému přejít s vývojem celého webu k jinému vývojáři. [4]

Open source systémy mají tedy řadu výhod, ale je třeba uvést, že to neznamená, že se hodí pro tvorbu jakéhokoli webu. Výběr redakčního systému je třeba zvážit na základě všech potřeb webu. [3]

Nyní si uvedeme některé z klíčových nevýhod těchto systémů:

- Open source platformy mohou představovat jistá bezpečnostní rizika. Programátor, který se ve zdrojovém kódu tohoto systému dobře orientuje, může využít svých znalostí a snadno tak identifikovat potenciální zranitelnosti a bezpečnostní rizika webu.
- Spolupráce s více vývojáři, která je běžná u open source CMS systémů, může vést k nižší kvalitě a konzistenci kódu. Kvalifikace a přístupy jednotlivých programátorů se mohou lišit. Toto může vést k nejednotnému stylu a technickému řešení webu a ztížit jeho údržbu.
- Poměrně velikou nevýhodou je omezená podpora. Za těmito systémy nestojí žádná firma, která by zajišťovala zákaznickou podporu. [4]

1.2 Komerční CMS

Komerční systémy CMS (někdy také proprietární CMS systémy) se odlišují od open source tím, že jsou vyvíjeny a vlastněny konkrétními společnostmi nebo organizacemi. Zdrojový kód pro tyto systémy není veřejně dostupný a přístup k nim vyžaduje zakoupení licence. [5]

Výhody komerčních CMS systémů:

- Hlavním rozdílem mezi komerčními a open-source CMS je jejich zaměření. Komerční CMS se často vyvíjejí se zaměřením na konkrétní odvětví, zatímco open-source řešení nabízí univerzální platformu s širokou škálou funkcí. [5]

- Poskytovatelé komerčních CMS systémů v praxi disponují komplexní zákaznickou podporou, která zahrnuje podrobnou dokumentaci, školící zdroje a přímou technickou podporu. Tyto zdroje napomáhají uživatelům efektivně se orientovat v sytému a řešit případně vzniklé problémy. [5]
- Komerční CMS systémy jsou v praxi vyvíjeny jednou společností, čímž mají tendenci být konzistentní a poskytovat uživatelsky přívětivější interakci. To zahrnuje například jednotný design a intuitivní rozhraní, které usnadňuje práci uživatelům. [4]

Komerční CMS systémy s sebou přináší i řadu nevýhod, které je třeba před výběrem CMS systému zvážit:

- Nevýhodou těchto systémů jsou náklady za licenci – u komerčních CMS si v podstatě pronajímáte software. Nekupujete si zdrojový kód a po skončení licence nebudete vlastníkem systému.
- Převod webu na jiný CMS systém může být problematický kvůli migraci dat. Každý systém přistupuje k tvorbě obsahu a ukládáni dat trochu jinak. [17]

2 .NET CORE

.NET Core je bezplatná, open-source platforma pro vývoj aplikací napříč operačními systémy, která dává vývojářům možnost vytvářet širokou škálu programů. [6] Platforma nabízí možnost vyvíjet v několika programovacích jazycích – C#, F# a Visual Basic. [7]

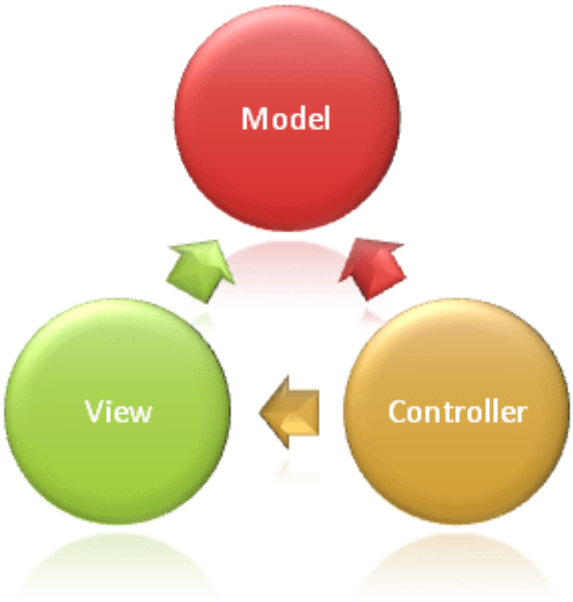
V roce 2002 Microsoft představil .NET Framework, ve kterém bylo možné programovat ve více než 60 programovacích jazycích, ale mohl běžet pouze na operačním systému Windows. To vedlo ke vzniku platformy .NET Core v roce 2016, která je schopná běžet na různých operačních systémech jako jsou Windows, macOS a Linux. [8]

Platforma .NET Core se odlišuje od svého předchůdce .NET Framework tím, že je distribuována formou balíčků NuGet. Tento přístup umožňuje vývoj .NET Core agilním způsobem. Jakýkoliv upgrade lze jednoduše provést aktualizací NuGet balíčku. Není tedy rozdíl mezi systémovými knihovnami a knihovnami třetích stran – všechno jsou to NuGet balíčky. Díky tomu je výběr knihoven mnohem snazší – pokud si do programu stáhnete NuGet balíček, který vyžaduje novější verzi systémové knihovny, systém vám nabídne aktualizaci příslušné systémové knihovny. Dále je potřeba zmínit, že modulární přístup NuGet balíčků zajišťuje, že finální aplikace bude obsahovat pouze ty balíčky, které potřebuje. [9]

2.1 ASP.NET Core

ASP.NET Core je multiplatformní, open-source framework pro vývoj moderních webových a cloudových aplikací. [10] Tento framework umožňuje vývoj široké škály aplikací za užití těchto nástrojů:

- ASP.NET Core Razor Pages – framework pro tvorbu jednoduchých webových stránek s využitím server-side renderingu. [10]
- AS.NET Core MVC – framework pro vytváření webových aplikací a API, který využívá Model-View-Controller (MVC) architekturu. Pomocí této architektury jsou uživatelské požadavky přesměrovány na Controller, který načte příslušné View a předá mu datový Model, který potřebuje. [11]



Obrázek 1. MVC architektura přejato z [11]

- ASP.NET Core Blazor – framework pro tvorbu webových stránek a webových aplikací. Umožňuje spouštět C# kód v prohlížeči. Nabízí možnost server-side i client-side renderingu. [10]

2.1.1 ASP.NET Core Razor Pages

Framework Razor Pages umožňuje tvorbu webových stránek s možností zobrazovat její obsah dynamicky. Umožňuje uvnitř HTML kódu stránky psát C# kód, který je při renderování zpracován. Tento framework je multiplatformní a využívá server-side renderingu. Můžeme ho vnímat jako implementaci MVC architektury, která usnadňuje tvorbu webů s touto architekturou. [12]

Nyní si uvedeme některé z kladných vlastností tohoto frameworku:

- Umožňuje renderovat webové stránky dynamicky.
- Dá se používat jako obdobný nástroj místo PHP nebo klasického ASP díky svému zaměření na jednotlivé stránky.
- Využívá jednoduchý princip pro tvorbu jednotlivých stránek – jedna stránka webu je reprezentována jedním .cshtml souborem, který je zodpovědný za její obsah a logiku. [30]

2.1.1.1 Základní syntaxe

Soubory s koncouvkou .cshtml slouží jako šablony pro generování výsledného HTML stránky na webu. Pro psaní C# kódu uvnitř HTML existuje soubor pomocných značek, které je třeba používat pro správné vyrenderování požadovaného obsahu.

Značka označující začátek C# kódu je @, pokud chceme napsat blok kódu ve více řádcích, musíme ho umístit uvnitř závorek takto: @{…}

Zde je příklad definice proměnné Title a vypsání její hodnoty do nadpisu H1:

```
@{
    string Title;
    Title = "Nadpis H1";
}

<h1>@Title</h1>
```

Pokud chceme vypsat hodnotu ve smyčce for, uděláme to takto:

```
@for (int number = 0; number < 5; number++)
{
    @number
}
```

Pro vypsání výsledku výrazu uvnitř smyčky použijeme tuto syntaxi:

```
@for (int number = 0; number < 5; number++)
{
    @(number * 5)
}
```

Může se ale stát, že chceme vypsat do textu znak @ a nechceme, abychom tím indikovali kód. V tomto případě použijeme zdvojení znaku @:

```
<p>email@@domena.cz</p>
```

Máme také možnost psát komentáře. Jsou tři varianty – použití dvou lomítek // pro jednořádkový komentář nebo víceřádkový komentář se dvěma možnostmi syntaxí /*…*/ nebo @*…*@:

```
@{
    //  jednořádkový komentář
    /*
        víceřádkový
        komentář
    */
    @*
        víceřádkový
        komentář
    *@
}
```

Výhodou této syntaxe @*…*@ je, že se tyto komentáře můžou psát mimo blok kódu a neobjeví se ve finálním renderu:

```
@*Toto je nadpis H1*@
<h1>Nadpis H1</h1>
```

2.1.2 Dependency Injection

Dependency injection je softwarový návrhový vzor, který má za cíl oprostit třídy od jejich závislostí na jiných třídách. Snaží se odstranit jakoukoliv závislost mezi třídou a službou, kterou využívá. Dependenci injection také napomáhá plnit zásady SOLID objektově orientovaného návrhu. Konkrétně naplňuje princip dependenci inversion. Díky tomu se stává náš kód více recyklovatelný. Tento návrhový vzor je realizován pomocí těchto čtyř tříd:

- Třída reprezentující potřebnou službu (service).
- Třída, která tuto službu využívá (klient).
- Interface, na který má klient vazbu a tento interface je zároveň implementován pomocí service.
- Injektor, který se stará o vytvoření instance service a injektování do třídy klient. [13]

ASP.NET Core podporuje tento návrhový vzor a má v sobě implementovaný injektor, takže se při využívání dependenci injection v aplikaci o tuto část nemusíme starat. [14]

Pojďme se nyní podívat na konkrétní příklad implementace dependenci injection v ASP.NET Core. Jako první vytvoříme interface pro naši službu:

```
public interface ILogger
{
	void LogMessage(string message);
}
```

Dále vytvoříme třídu, která bude toto rozhraní implementovat:

```
public class Logger : ILogger
{
	public void LogMessage(string message) {
		Console.WriteLine(message);
	}
}
```

Aby byla tato služba dostupná, je třeba v souboru Program.cs službu zaregistrovat. Při registraci nové služby se zpravidla uvádí název interface a jeho implementace:

```
//create builder

var builder = WebApplication.CreateBuilder(args);

// Add Logger service

builder.Services.AddSingleton<ILogger, Logger>();
```

V tuto chvíli už můžeme vytvořit třídu, která bude tuto službu využívat:

```
public class IndexModel : PageModel
{
    private readonly ILogger _logger;

    public IndexModel(ILogger logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogMessage("Hello from IndexModel");
    }
}
```

Nyní si podrobně rozebereme ukázku kódu. Máme třídu `IndexModel`, která dědí z třídy `PageModel`. Tato třída tedy bude model pro nějakou stránku na webu. Aby byla tato třída schopná logovat nějaké informace, vytvořili jsme privátní readonly property s názvem

`_logger`. V konstruktoru poté dojde k injektování závislostí do této třídy. Jako parametr konstruktoru je `ILogger`, kterým se naplní property naší třídy `_logger`. Poslední, co v této třídě máme, je funkce `OnGet`, která využívá injektovaného loggeru a volá jeho funkci `LogMessage` pro zalogování informace.

V kódu také vzniklo rozhraní `ILogger`. V tomto rozhraní dochází k definici všech metod, které musí jeho třídy implementovat.

Třída `Logger` je implementací interface `ILogger`. Implementuje funkci `LogMessage` a to tak, že vypíše text do konzole. Třída `IndexModel` je závislá na interface `ILogger`, tato třída nemá tedy žádnou přímou vazbu na třídu `Logger`. Pokud se změní implementace funkce `LogMessage`, nijak to neovlivní třídu `IndexModel`, dá se jednoduše říct, že třídu `IndexModel` vůbec nezajímá, jestli se logování vypíše pouze do konzole, nebo jestli dojde k logu do databáze.

Jako poslední je příkaz `builder.Services.AddSingleton`. Toto je příkaz, který je potřeba napsat do souboru `Program.cs` a stará se o zaregistrování dané služby. Díky tomuto příkazu dojde při volání konstruktoru třídy `IndexModel` k předání instance služby `ILogger`.

Ve frameworku ASP.NET Core existuje několik způsobů, jak službu zaregistrovat.

Existují tři základní typy:

- Singleton
- Scoped
- Transient [15]

V následujících odstavcích si popíšeme životní cyklus každého typu.

Singleton – pokud v programu zeregistrujeme službu jako singleton, vytvoří se v programu vždy jen jedna instance. V aplikacích, které zpracovávají requesty se pro každý request využívá stále stejná instance této služby. Její životní cyklus je po celou dobu běhu aplikace. [15] Zde je příklad registrace sužby jako singleton:

```
services.AddSingleton<ILogger, Logger>();
```

Scoped – služba zaregistrovaná pomocí metody scoped vytváří vždy novou instanci pro každý request. V rámci zpracování jednoho requestu je vždy sdílena jedna instance této služby. Její životní cyklus je po celou dobu jednoho requestu. [15] Zde je příklad registrace služby jako scoped:

```
services.AddScoped<ILogger, Logger>();
```

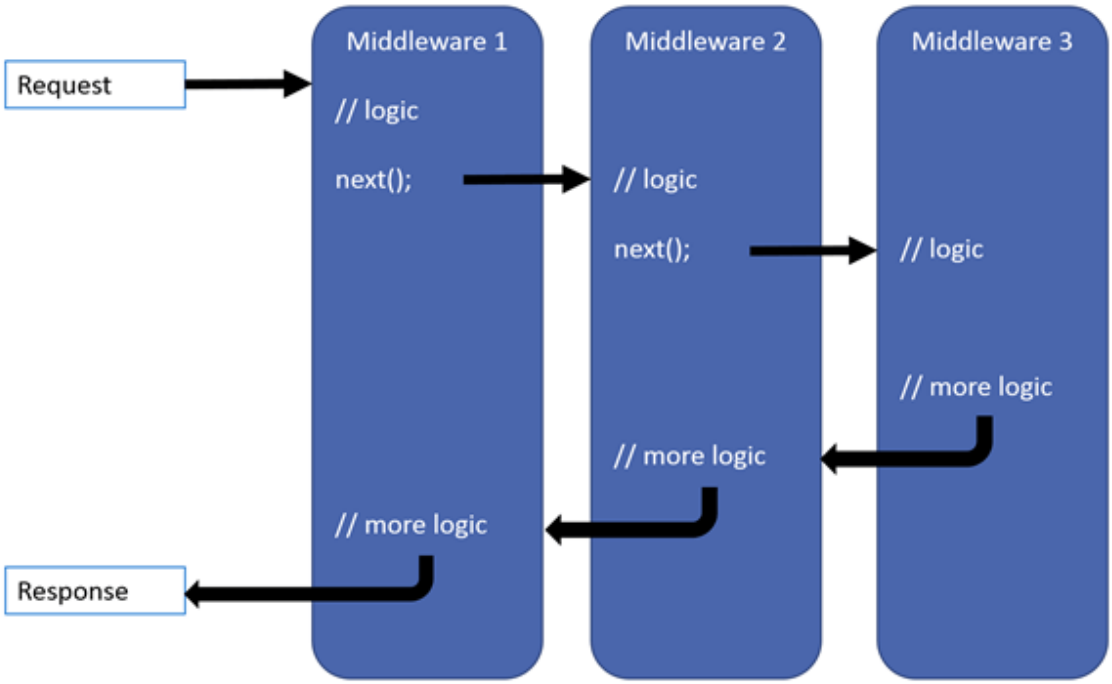
Transient – pokud nechceme v systému žádný sdílený stav služby, použijeme způsob transient. Pro každou třídu, která využívá takto zaregistrovanou službu, vznikne vlastní instance. [15] Zde je příklad registrace služby jako transient:

```
services.AddTransient<ILogger, Logger>();
```

2.1.3 ASP.NET Core Middleware

Middleware v ASP.NET Core je software, který se skládá z komponent (tříd) a je součástí pipeline požadavků. Tyto komponenty se aktivují sekvenčně při každém requestu. [16]

Zde je diagram, který demonstruje tento koncept:



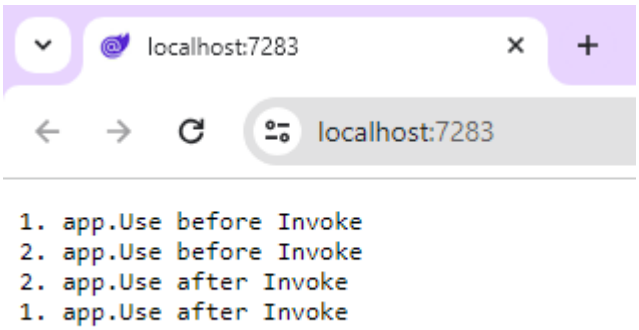
Obrázek 2. Pipeline requestu v ASP.NET Core přejato z [16]

V pipeline programu můžeme komponentu middleware definovat a konfigurovat pomocí funkcí `Run`, `Map` a `Use`. Řetězení více middleware za sebou provádíme pomocí příkazu `next()`, kde `next` reprezentuje další middleware v pořadí. Komponenta middleware má také možnost zkratovat další zpracování v pipeline požadavků tím, že nezavolá tento příkaz. Toto se využívá v případě, kdy se chceme vyhnout provádění nepotřebného kódu. Například, když přijde požadavek na statický soubor jako je obrázek nebo CSS soubor. [16]

Pojďme si nyní ukázat příklad, kde budeme demonstrovat řetězení více middleware za sebou:

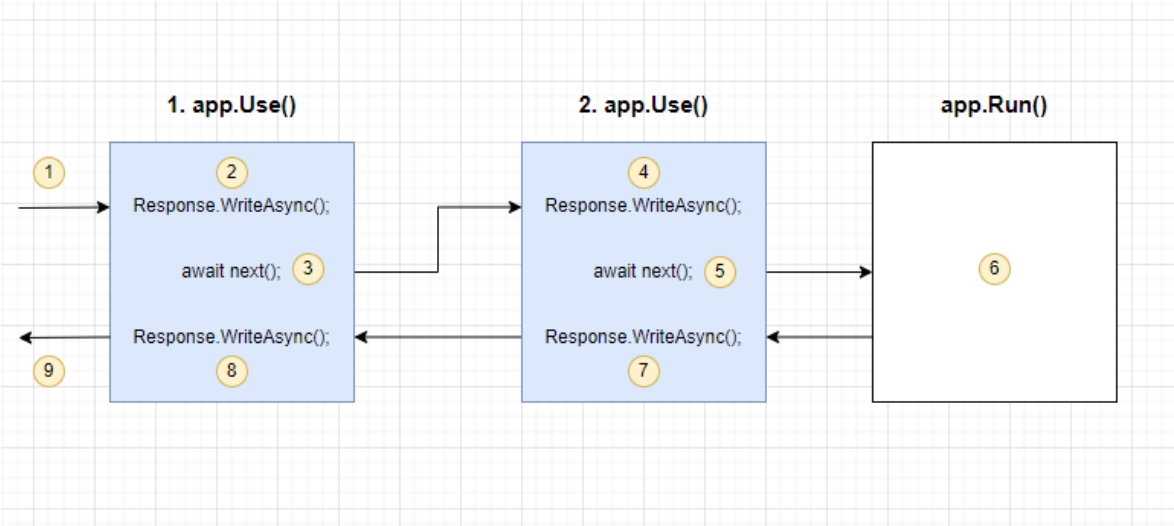
```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("1. app.Use before Invoke\n");
    await next();
    await context.Response.WriteAsync("1. app.Use after Invoke\n");
});

app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("2. app.Use before Invoke\n");
    await next();
    await context.Response.WriteAsync("2. app.Use after Invoke\n");
});

app.Run();
```

V příkladu jsou vytvořeny dva middleware, které pouze vypisují text na klienta. Zde je obrázek vypsaného textu:



Obrázek 3. Vypsaný text pomocí middleware

Zde je diagram pro grafické zobrazení pipeline této jednoduché aplikace:



Obrázek 4. Diagram pipeline ukázkové aplikace

3 BLAZOR

Blazor je web UI framework, který nám umožňuje používat C# a .NET Core na front-endu. Umožňuje nám to vytvářet front-end logiku aplikace různými způsoby s využitím C# programovacího jazyka. Framework Blazor využívá syntaxi Razor pro tvorbu jednotlivých komponent aplikace. [32] Pojďme si nyní říct jaký je zásadní rozdíl mezi Razor a Blazor. Zásadní rozdíl je, že C# kód napsaný ve frameworku Razor bude proveden pouze jednou, a to ve chvíli, kdy je načtena webová stránka, ale ve frameworku Blazor může být kód proveden při načtení stránky a také, když nastanou různé eventy jako jsou onclick a onchange. [18]

Možnost používání platformy .NET a spouštění C# kód na straně klienta přináší několik výhod. Můžeme se takto vyhnout JavaScript frameworkům a můžeme psát klientskou i serverovou část v jazyce C#. Aplikace, kde je klientská i serverová část napsaná v jazyce C# zvyšuje produktivitu při vývoji a údržbě. Máme také možnost sdílení serverového a klientského kódu v aplikační logice. Díky ekosystému .NET knihoven máme k dispozici širokou škálu nástrojů a můžete těžit z jejich výkonu, spolehlivosti a zabezpečení. [19]

3.1 Komponenty

Blazorové aplikace se skládají z komponent, které jsou samostatnými a opakovaně použitelnými bloky kódu. Blazor využívá syntaxe Razor pro renderování komponent (viz Základní syntaxe). [19] Jak už jsme si dříve řekli, kód pro jednotlivé stránky ve frameworku Razor se píše do souboru s koncovkou .cshtml. V Blazoru píšeme kód pro jednotlivé komponenty do souborů s koncovkou .razor. Těmto komponentám říkáme Razor komponenty. [18]

Pomocí komponenty v Blazoru můžeme vytvořit navigovatelné stránky, layouty a opakovaně použitelné bloky UI. [20] V následujících stránkách se na každý typ podíváme detailněji.

3.1.1 Navigovatelné stránky

Razor komponenty se mohou chovat jako webové stránky a mohou být navigovatelné, pokud použijeme direktivu @page a za ní relativní cestu, pod kterou bude tato stránka zobrazena. Můžeme klidně použít direktivu @page vícekrát a potom bude stránka zobrazena pod různými cestami. [20]

Zde je ukázkový příklad:

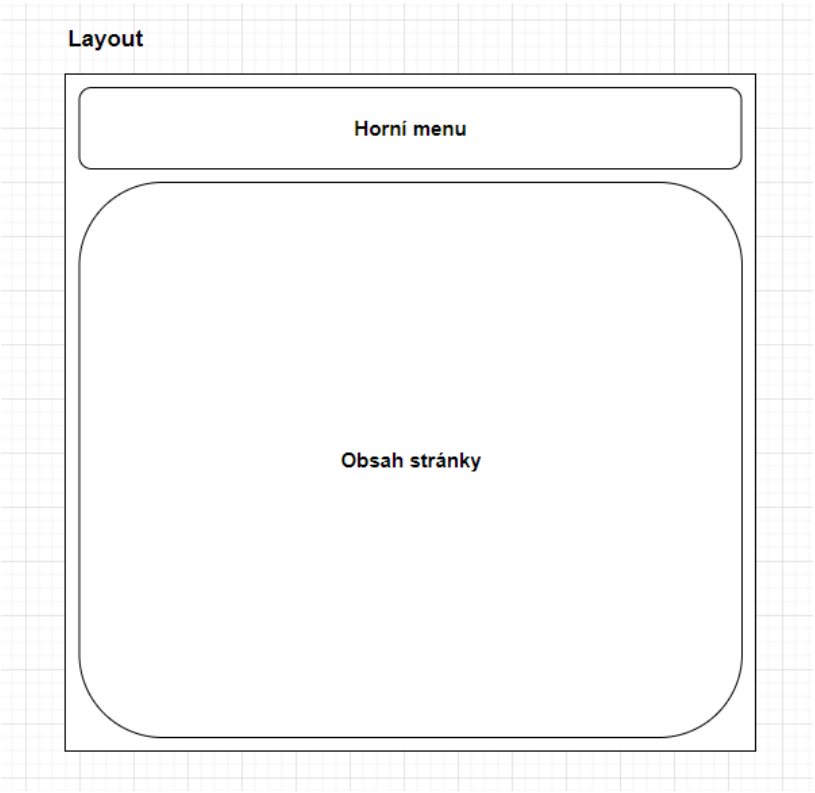
```
@page "/o-nas"
@page "/informace-o-nas"

<h1>O nás</h1>
```

Vytvořili jsme takto stránku o nás, která obsahuje pouze nadpis H1. Řekněme, že naše webová aplikace poběží na doméně www.testaplikace.cz. Stránka o nás se tedy zobrazí pod URL www. testaplikace.cz/o-nas nebo www.testaplikace.cz/informace-o-nas.

3.1.2 Layouty

Komponentu můžeme nazývat layout, když dědí ze třídy LayoutComponentBase a vykresluje obsah property Body. Tato property obsahuje obsah stránky. [20]

Vytvoříme si jednoduchý ukázkový layout. Budeme chtít, aby rozložení stránky bylo takové, že nahoře bude pásek s menu a pod ním uvidíme obsah stránky. Diagram by vypadalo takto:

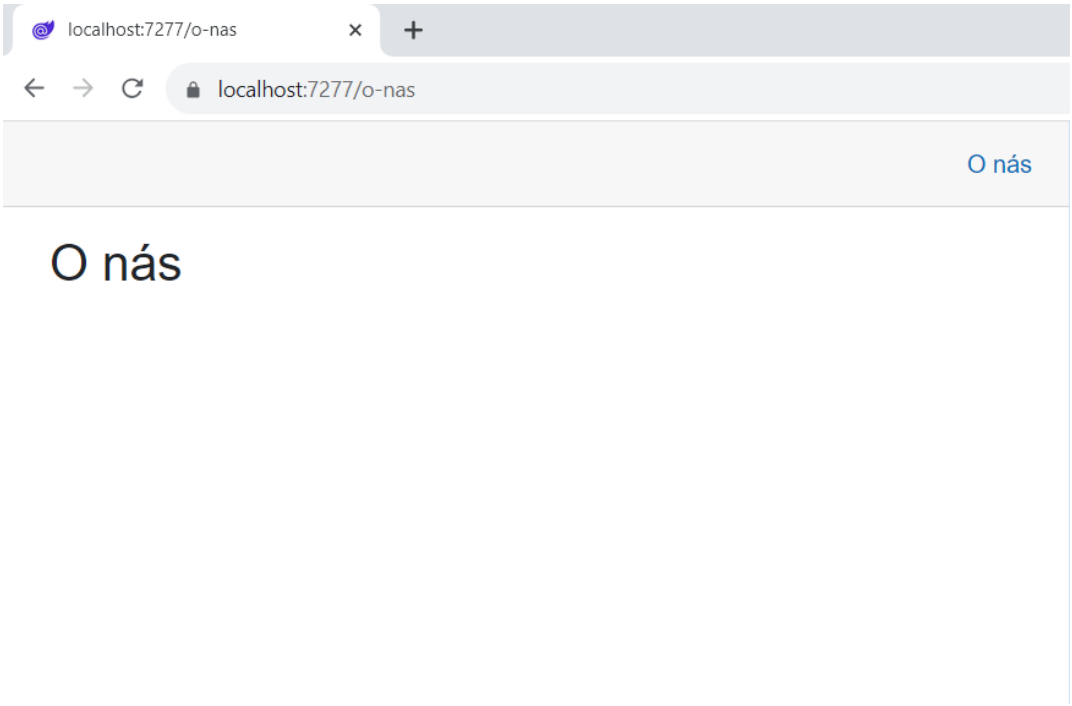


Obrázek 5. Ukázka layoutu

Zde je odpovídající kód:

```
@inherits LayoutComponentBase

<div class="page">
    <main>
        <div class="top-row px-4">
            <a href="o-nas">O nás</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

A takto vypadá výsledný render stránky O nás z předchozího příkladu s využitím nového layoutu:



Obrázek 6. Render stránky O nás s použitým layoutem

3.1.3 Opakovaně použitelné bloky UI

Komponenty mohou zahrnovat další komponenty jako součást svého obsahu, což znamená, že mohou být použity k reprezentaci jednoduchého nebo složitého uživatelského rozhraní, které se používá na více místech v rámci jedné aplikace. [20]

Komponentu můžeme vložit dovnitř jiné komponenty umístěním tagů podobných HTML. Tento tag musí obsahovat název komponenty a umístíme ho na místo, kde chceme, aby se komponenta vykreslila. [20]

Upravíme kód layoutu tak, aby do sebe vkládal obsah jiné komponenty. V tuto chvíli máme odkaz na stránku O nás vytvořený jako klasický odkaz, takže při kliknutí dojde k přenačtení celé stránky. Blazor ale umožňuje interaktivní navigaci mezi stránkami. Místo klasického HTML tagu `<a href="o-nas">` můžeme použít komponentu `<NavLink href="o-nas">`. Tato komponenta zajistí, že při kliknutí na odkaz nedojde k přenačtení celé stránky, ale pouze k výměně obsahu jiné stránky za obsah stránky O nás. Nedojde takto k přenačtení společného layoutu.

Upravený kód vypadá takto:

```
@inherits LayoutComponentBase

<div class="page">
    <main>
        <div class="top-row px-4">
            <NavLink href="o-nas">
                O nás
            </NavLink>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

V příkladu jsme využili jednu ze systémových komponent Blazoru, ale můžeme vytvářet i vlastní komponenty.

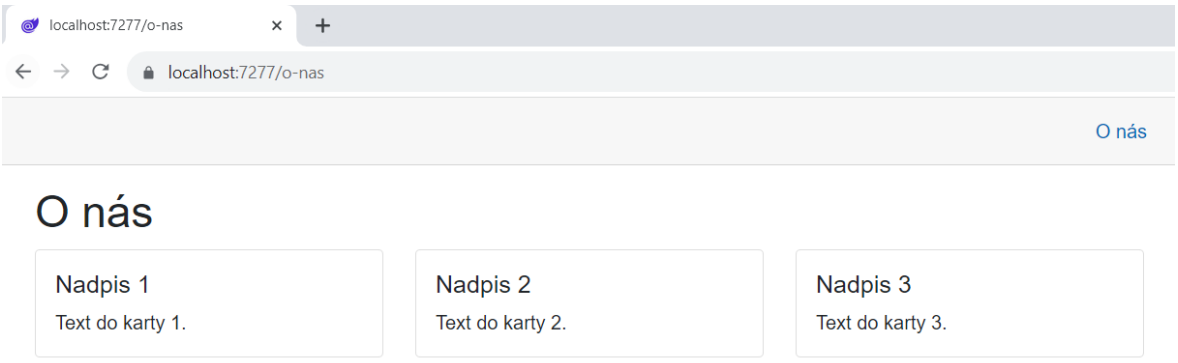
Dále také musíme zmínit, že Razor komponenty si navzájem umí předávat data pomocí parametrů. Parametry komponenty jsou definovány jako public C# property ve třídě komponenty a musí mít atribut `[Parameter]`. Pro příklad si vytvoříme jednoduchou komponentu `Card`, která zobrazí kartu s nadpisem a textem. Nadpis a text se bude dát nastavit pomocí parametrů `Title` a `Text`.

Takto vypadá kód uvnitř komponenty `Card.razor`:

```
<div class="card" style="width: 18rem;">
    <div class="card-body">
        <h5 class="card-title">@Title</h5>
        <p class="card-text">@Text</p>
    </div>
</div>


@code {
    [Parameter]
    public string? Title { get; set; }
    [Parameter]
    public string? Text { get; set; }
}
```

Vytvořenou komponentu využijeme na stránce O nás a zobrazíme tak nějaké informace formou karet:

```
@page "/o-nas"
@page "/informace-o-nas"

<h1>O nás</h1>

<div class="row">
      <div class="col-4">
            <Card Title="Nadpis 1" Text="Text do karty 1." />
      </div>
      <div class="col-4">
            <Card Title="Nadpis 2" Text="Text do karty 2." />
      </div>
      <div class="col-4">
            <Card Title="Nadpis 3" Text="Text do karty 3." />
      </div>
</div>
```



Obrázek 7. Render stránky O nás s využitím komponenty Card

3.2 Módy renderování

Framework Blazor umožňuje vývoj webových aplikací s různými módy renderování. Komponenty je možné renderovat v prohlížeči na straně klienta pomocí technologie WebAssembly, nebo na straně serveru. [22] Ve verzi .NET 8 Microsoft zkombinoval výhody obou frameworků Razor a Blazor a přidal do frameworku Blazor možnost server-side renderingu bez interakce s front-endem. Tento mód je nyní nastaven pro Blazor projekty defaultně. Můžeme si však vybrat mód renderování, a to jak na globální úrovni celé aplikace, nebo zvlášť pro každou Razor komponentu. Obrovskou výhodou je, že syntaxe pro tvorbu komponent není nijak závislá na módu renderování. [23]

V této kapitole si projdeme všechny módy renderování, které framework Blazor nabízí.

3.2.1 Blazor WebAssembly

Technologie WebAssembly nám umožňuje kompilovat kód do formátu zvaném WASM, který si stáhne prohlížeč. V prohlížeči se tento kód následně zkompiluje Just-In-Time do nativního kódu. Můžeme tak na straně klienta spouštět C# kód. [31]

Nyní si uvedeme některé výhody, který tento mód renderování nabízí:

- Blazor WebAssembly nabízí uživatelům rychlou a plynulou interakci. Toho je dosaženo díky okamžitým aktualizacím uživatelského rozhraní (UI) přímo v prohlížeči, bez nutnosti komunikace se serverem. [24]
- Po stažení aplikace do prohlížeče může aplikace běžet bez připojení k internetu. [24]
- Tento model renderování také šetří serverové zdroje. Aplikace komunikuje se serverem při prvotním stažení aplikace, ale dále už komunikace se serverem není potřeba. [22]

Uvedeme si také nějaké nevýhody:

- Prvotní spuštění Blazor WebAssembly aplikace může být pomalejší, protože si prohlížeč musí stáhnout celou aplikaci. [22]
- Blazor WebAssembly sice umožňuje spouštět nativní kód v prohlížeči, ale složitější výpočty pak mohou být pomalejší, jelikož se provádí na straně klienta. U mobilních zařízení a slabších počítačů se může projevit zpoždění v reakci aplikace. [24]

- Pokud budeme chtít v aplikaci přistup k datům z databáze uložené na serveru, nevyhneme se volání API. [22]
- Aplikace je omezena možnostmi prohlížeče. Ne všechny prohlížeče podporují tento model. [18]

Dále si ukážeme, jak v Blazor aplikaci nastavit mód renderování WebAssembly. Zde je příklad, jak nastavit v souboru Program.cs tento mód renderování:

```
builder.Services.AddRazorComponents()
    .AddInteractiveWebAssemblyComponents();


var app = builder.Build();

app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode();
```

Tímto se nastaví pro celou Blazor aplikaci tento mód renderování. Pokud chceme módy renderování kombinovat, musíme v souboru Program.cs nastavit více módů renderování, a pak můžeme u každé komponenty zvlášť specifikovat mód jejího renderování, a to hned dvěma způsoby.

- Nastavení módu renderování pro instanci komponenty [31]:

```
<Card @rendermode="InteractiveWebAssembly" />
```

- Nastavení módu renderování uvnitř definice komponenty [31]:

```
@page "/o-nas"
@rendermode InteractiveWebAssembly
```

3.2.2 Blazor Server

V tomto modelu se veškerý kód komponenty spouští na serveru a do prohlížeče se odesílá pouze hotový HTML a CSS kód. Komunikace mezi serverem a klientem probíhá pomocí SignalIR. SignalIR spojení se stará o aktualizaci uživatelského rozhraní na základě volaných eventů nebo aktualizaci dat z databáze. [22]

Výhody Blazor Server:

- Aplikace, která využívá tento model hostování nemusí stahovat do prohlížeče celou aplikaci jako je tomu u WebAssembly. Prvotní načtení aplikace bude tedy rychlejší než u WebAssembly. [22]
- Jelikož aplikace běží na serveru, můžeme data uložené v databázi načítat přímo, bez nutnosti volání API. [22]
- Aplikace není závislá na možnostech prohlížeče. Aplikace s tímto módem renderování bude fungovat v prohlížečích, kde nebude fungovat WebAssembly. [18]

Nevýhody:

- Tento mód renderování spotřebovává spoustu serverových zdrojů, kvůli sledování stavu každého klienta. [22]
- Aktualizace uživatelského rozhraní je závislá na stabilním připojení k inernetu. [22]
- Vytvoření složitějších interakcí v uživatelském rozhraní může být v tomto režimu o něco těžší. Je to proto, že se změny v UI zobrazují pomaleji než u Blazor WebAssembly. Důvodem je, že se informace pro aktualizaci UI odesílají ze serveru do prohlížeče.

Zde je příklad nastavení Blazor Server módu renderování v souboru Program.cs:

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();


var app = builder.Build();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

3.2.3 Blazor Auto

Blazor Auto je nový mód renderování v Blazoru, který umožňuje dynamické přepínání mezi Blazor Server a Blazor WebAssembly za běhu aplikace. Prvotní načtení aplikace proběhne na straně serveru, na pozadí zatím dojde ke stažení WASM kódu do prohlížeče a další interakce poté probíhá na straně klienta za použití WebAssembly. [25]

Tento model se snaží kombinovat výhody obou módů renderování. Pojďme si tedy uvést jeho výhody:

- Prvotní načtení aplikace bude stejně rychlé jako u Blazor Server.
- Pokud je to možné, Blazor Auto se snaží využívat WebAssembly pro rychlou a plynulou interakci uživatele.
- Pokud aplikace potřebuje přistupovat k datům uloženým na serveru, dá se využít Blazor Serveru a tím se odstraní potřeba pro volání API. [25]

Nevýhody:

- Všechny komponenty, které mají běžet v módu WebAssembly, je nutné umístit do samostatného klientského projektu. Důvodem je, že na klienta chceme stahovat pouze tyto komponenty. Pokud by se komponenty pro WASM nacházely v serverovém projektu, museli by si uživatelé stahovat i serverový kód, který nepotřebují. [25]
- Tento režim se sice snaží o co nejrychlejší přepnutí na klientské vykreslování, může však dojít ke krátkému zpoždění. Důvodem je nutnost počáteční kontroly dostupnosti prostředků WebAssembly, nezbytných pro chod aplikace na straně klienta. [26]

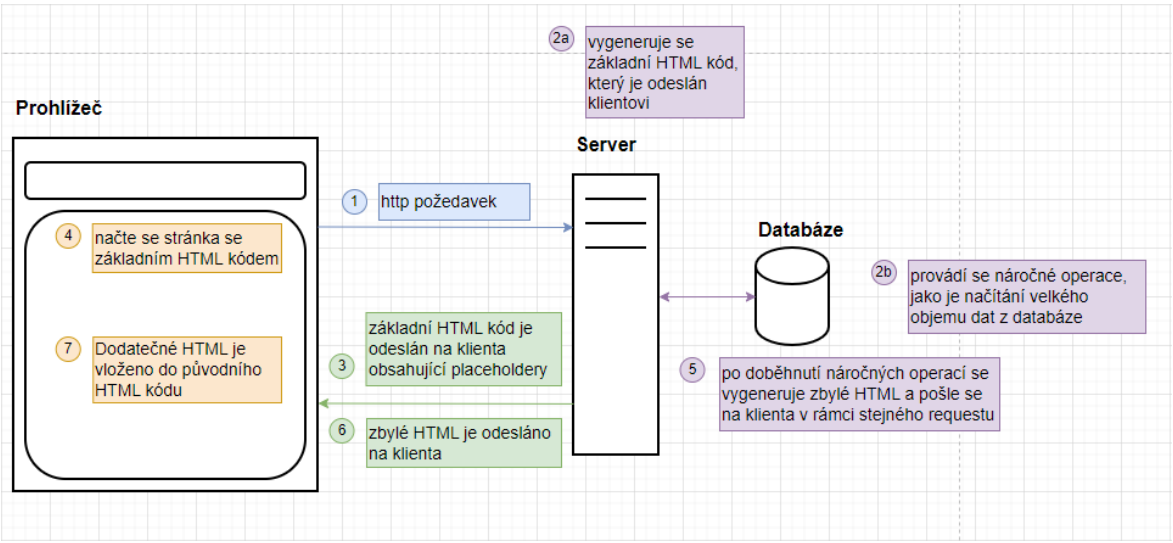
3.2.4 Blazor Static Server

Ve frameworku Blazor existuje ještě možnost aplikace bez interakce mezi klientem a serverem. Blazor Static Server funguje na stejném principu jako Razor Pages s tou výhodou, že můžeme využívat Razor komponent. Navíc se kdykoliv můžeme rozhodnout mód renderování změnit, protože syntaxe pro psaní Razor komponent je univerzální a nezávisí na zvoleném módu renderování. [27]

Velice zajímavou vlastností Blazor Static Server je Streaming Rendering. V základním nastavení Blazor Static Server pracuje tak, že přijde na server požadavek, ten je zpracován na straně serveru a vrátí HTML požadované stránky. Může se stát, že na straně serveru budeme provádět složitější operace nad databází a v důsledku toho se prodlouží čas odpovědi na požadavek. Streaming Rendering umožňuje stránkám, které potřebují provádět tyto složité operace, aby měly rychlou odezvu. Server nejprve odešle základní HTML kód stránku obsahující placeholdery, zatímco na pozadí probíhá zpracování dat.

Jakmile se data načtou, server pomocí streamu vrátí zbylé požadované HTML a vloží ho do původního HTML kódu. Toto se děje v rámci jednoho requestu a odpovědi na něj. [27]

Zde je obrázek, který ukazuje tento scénář:



Obrázek 8. Diagram popisující události Streaming Renderingu

Výhody Blazor Static Server:

- Umožňuje renderovat webové stránky dynamicky.
- Umožňuje jednoduché optimalizace pro SEO.
- S využitím Streaming Rendering lze dosahovat velice rychlé odezvy ze serveru.
- Má stejné vlastnosti jako Razor Pages s výhodou Razor komponent a možnosti přepnout mód renderování. [27]

Nevýhody:

- Není vhodný pro webové stránky s větší interaktivitou se serverem.
- Tento mód renderování využívá pouze serverových zdrojů a nevyužívá zdrojů klienta. [27]

CMS systém, který byl vytvořen v rámci této práce, využívá tohoto módu renderování. Framework Blazor byl pro vývoj systému zvolen kvůli jeho možnosti tvořit komplexní komponenty, psaní kódu v jazyce C# a nově přidané možnosti server-side renderingu. Díky tomu můžeme v systému přehledně a jednoduše vytvářet komponenty, můžeme také

načítat data z databáze bez nutnosti volání API a optimalizovat rychlost načítání stránek s využitím Streaming Renderingu.

3.3 Zabezpečení

3.3.1 Cross-Site Scripting

Cross-Site Scripting (XSS) je typ bezpečnostní chyby, která umožňuje útočníkovi vložit skripty na straně klienta (obvykle JavaScript) do webových stránek. Tento kód se pak spustí v prohlížeči uživatelům, kteří navštíví napadenou stránku a může tak dojít k ohrožení jejich bezpečnosti a soukromí. [28]

Útočníci se snaží škodlivý kód vložit například skrz formuláře umístěné na webu, nebo zasíláním škodlivých dat v URL Query Stringu. Abychom se o tuto hrozbu nemuseli starat, Razor komponenty automaticky enkódují veškeré vstupy. [29]

Zde je příklad enkódování v Razor komponentě:

```
@{
        var text = "<script />";
}

@text
```

Text vypsaný na klienta:

```
&lt;script /&gt;
```

Došlo k enkódování znaků < a >, takže prohlížeč tento text nezaregistroval jako HTML script tag.

3.3.2 SQL Injection

SQL Injection je typ útoku, kdy se útočník snaží na server vložit nebezpečný SQL kód a spustit ho v databázi. Útočník se takto snaží dostat k citlivým datům. Webová aplikace, která načítá data z databáze by měla těmto útokům zabránit. Ve Frameworku Blazor můžeme tomuto zabránit využitím technologií, které nespoléhají na přímé dotazování databáze pomocí SQL dotazů. Příkladem je Entity Framework, použití parametrizovaných dotazů nebo použití procedur uložených v databázovém projektu. [29]

II. PRAKTICKÁ ČÁST

4 ANALÝZA POŽADAVKŮ NA CMS

4.1 Současný systém

CMS systém webManager se skládá ze tří částí. API a klientská část pro admin rozhraní a systém pro renderování webových stránek z dat vytvořených přes admin systém. Admin systém proběhl naprostým přepracováním a nyní je API naprogramované v technologii ASP.NET Core a klientská část využívá javascriptový framework Vue. Systém, který se stará o renderování webových stránek zatím zůstává v technologii ASP 3.0, proto se tato práce bude zabývat přepracováním této části systému do nových, soudobých technologií.

4.1.1 Hlavní nedostatky

Zastaralá technologie ASP 3.0 – Zastaralá technologie brzdí další rozvoj systému. Protože je jednodušší některé funkcionality systému vyvíjet v technologii .NET, systém využívá COM objektů a tím se snaží dohánět své nedostatky. Využívání COM objektů ale zabírá spoustu režie.

Nepřehledná a pomalá tvorba šablon – Starý systém využívá vlastní šablonovací systém. Šablony jsou uloženy v XML souboru. Každá šablona obsahuje HTML obsah a speciální značky pro volání funkcí. Při tvorbě šablon tedy nejde využít žádných našeptávačů, nebo jiných pokročilých funkcionalit, které vývojová prostředí nabízí.

Pro vyvíjení webu je třeba projekt rozchodit na IIS serveru – Pokud chce vývojář začít s implementací nového webu, je třeba, kromě vytvoření nového projektu, rozjet projekt na lokálním testovacím IIS serveru. Tento úkon je náchylný k chybám a zabírá spoustu času. Oproti tomu aplikace v technologii ASP.NET je možné spustit pomocí IIS express serveru přímo v aplikaci Visual Studio.

4.2 Funkční požadavky

Funkční požadavky vycházejí ze stávajícího systému, protože funkcionalita nového systému by měla zůstat stejná.

R001: Systém umožní renderovat stránky na základě dat z databáze.

R002: Systém umožní seskládat obsah webové stránky z vytvořených pluginů.

R003: Systém umožní vykreslit na webovou stránku seznam článků.

R004: Systém umožní zobrazit detail článku.

R005: Systém umožní vykreslit pro všechny stránky na webu stejnou hlavičku s navigačním menu.

R006: Systém umožní renderovat webové stránky ve více jazykových mutacích.

R007: Systém umožní renderovat pluginy na základě vytvořených šablon.

R008: Systém umožní při nenalezení stránky zobrazit stránku 404 s vlastním obsahem.

R009: Systém umožní hostovat webové stránky pro více zákazníků.

4.3 Nefunkční požadavky

Nyní se budeme zabývat nefunkčními požadavky pro CMS systém. Tyto požadavky byly navrženy tak, aby zaručily bezpečnost a základní vlastnosti pokročilého CMS systému.

R010: Systém bude držet historii aliasů stránek.

R011: Systém musí podporovat nástroje pro SEO.

R012: Systém umožní pro webovou stránku lokalizovat veškeré texty do relevantních jazyků.

R013: Systém musí být chráněn před pokusy o SQL injection.

R014: Systém musí být chráněn před pokusy o cross-site scripting.

R015: Systém umožní jednoduchou a přehlednou tvorbu HTML šablon.

5 POPIS REALIZOVANÉHO SYSTÉMU

V této kapitole se zaměříme na podrobný popis realizovaného systému pro dynamické vykreslování obsahu webových stránek. Systém zobrazuje obsah z dat uložených v databázi a umožňuje tak flexibilní zobrazování obsahu dle požadavků uživatele.

Systém implementuje soubor pluginů pro tvorbu prezentačních webů, které stávající systém obsahuje. Pro každý web hostovaný na tomto systému je možné vytvořit sadu šablon, které se starají o vzhled a chování vykreslených pluginů na webu.

V této kapitole podrobně rozebereme logické rozčlenění projektu a představíme klíčové komponenty systému. Součástí kapitoly budou i ukázky zdrojového kódu pro ilustraci principů fungování.

5.1 Vytvoření projektu

Při vytváření nového projektu ve frameworku Blazor je potřeba vybrat správný mód renderování. Systém byl vyvíjen s použitím pouze server-side renderingu, takže jako mód renderování byl vybrán Blazor Static Server. V projektu následně vznikly tři části – Core, Data a Components.

5.1.1 Core

Tento projekt obsahuje Blazor Static Server. Dochází zde ke zpracování veškerých HTTP požadavků. Je zde implementován tenant middleware, který na základě požadované domény načte data odpovídajícího tenanta.

Aby byl obsah webových stránek v systému dynamický, nejsou zde implementovány jednotlivé end pointy pro požadované stránky. V systému existuje pro zobrazení obsahu stránky pouze jeden end point. Kvůli tomu je zde implementován vlastní rewrite middleware, který se stará o převedení požadované URL na query string a nasměrování na zmíněný end point, kde dojde k zobrazení obsahu na základě načteného query stringu.

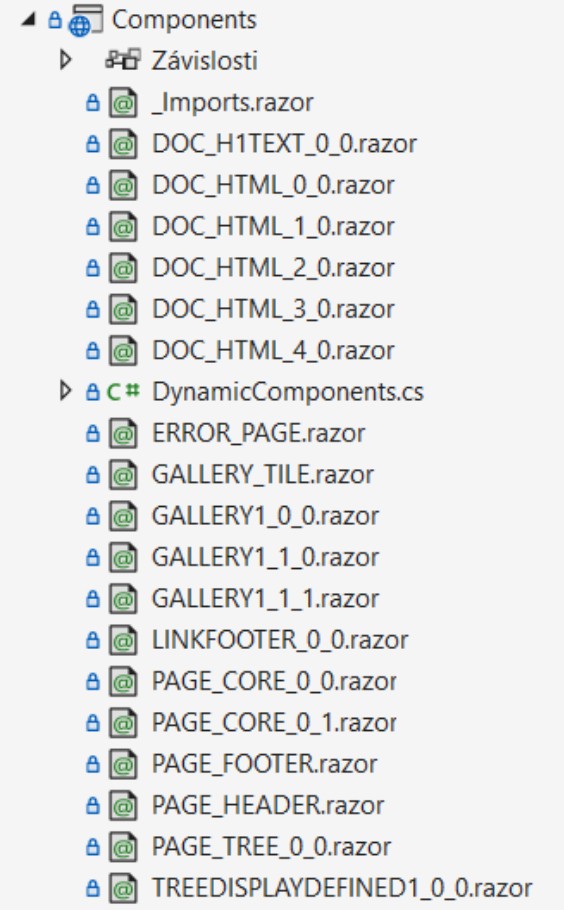
Každý tenant v systému má vlastní sadu komponent, které slouží k zobrazení obsahu webu. Komponenty jsou uložené pro každého tenanta zvlášť jako dll soubory. Pro načtení těchto dll souborů do paměti slouží služba `IComponentService`, kterou také najdeme v tomto projektu.

5.1.2 Data

Tato knihovna se stará o načítání dat z databáze. Systém je napojený na Microsoft SQL Server databázi stávajícího CMS systému, která mimo jiné obsahuje sadu procedur pro načítání dat. Pro načítání veškerých dat najdeme v této knihovně také interface, takže je možné systém napojit i na jiný zdroj dat. Tato knihovna nepoužívá Entity Framework. Najdeme zde funkce pro volání procedur a modely, na které se načtená data transformují. Knihovna také implementuje možnost cáchování dat.

5.1.3 Components

CMS systém umožňuje načíst pro každý hostovaný web vlastní sadu komponent. Pro každý web musí tedy vzniknou samostatný projekt, který obsahuje blazor komponenty. Tento projekt musí vždy obsahovat třídu dědící z třídy `IDynamicComponents`. Tato třída obsahuje dictionary, který mapuje název šablony pluginu s konkrétní blazor komponentou. Na následujících dvou obrázcích můžeme vidět, jaké komponenty vznikly pro ukázkový web a třídu implementující rozhraní `IDynamicComponents`.



Obrázek 9. Obsah projektu Components

```
public class DynamicComponents : IDynamicComponents
{
    public IDictionary<string, Type> Components => new Dictionary<string, Type>
    {
        { "TREE_CORE_0_0", typeof(PAGE_TREE_0_0) },
        { "PAGE_CORE_-1_0", typeof(ERROR_PAGE) },
        { "PAGE_CORE_0_0", typeof(PAGE_CORE_0_0) },
        { "PAGE_CORE_0_1", typeof(PAGE_CORE_0_1) },
        { "DOC_HTML_0_0", typeof(DOC_HTML_0_0) },
        { "DOC_HTML_1_0", typeof(DOC_HTML_1_0) },
        { "DOC_HTML_2_0", typeof(DOC_HTML_2_0) },
        { "DOC_HTML_3_0", typeof(DOC_HTML_3_0) },
        { "DOC_HTML_4_0", typeof(DOC_HTML_4_0) },
        { "TREEDISPLAYDEFINED1_0_0", typeof(TREEDISPLAYDEFINED1_0_0) },
        { "DOC_H1TEXT_0_0", typeof(DOC_H1TEXT_0_0) },
        { "GALLERY1_0_0", typeof(GALLERY1_0_0) },
        { "GALLERY1_1_0", typeof(GALLERY1_1_0) },
        { "GALLERY1_1_1", typeof(GALLERY1_1_1) },
        { "LINKFOOTER_0_0", typeof(LINKFOOTER_0_0) }
    };
```

Obrázek 10. Třída implementující rozhraní IDynamicComponents

5.2 Klíčové části systému

Tato kapitola se zaměřuje na klíčové části a principy fungování implementovaného systému. Cílem této kapitoly je detailně popsat vnitřní fungování systému a demonstrovat jeho schopnosti pro dynamické vykreslování webového obsahu.

V kapitole si ukážeme implementovanou multitenantní architekturu, dynamické načítání komponent, komunikaci systému s databází a zpracování HTTP požadavku. Kapitola obsahuje obrázky s ukázkami zdrojových kódů klíčových částí systému.

5.2.1 Multitenantní systém

Začneme tím, jak je v systému implementováno načtení tenanta z požadované domény. Projekt obsahuje soubor appsettings.json, který obsahuje globální nastavení celého systému. Tento soubor obsahuje seznam všech tenantů v systému. Každý tenant uvedený v tomto souboru musí mít uvedené informace, které vidíte na obrázku:

```
"ApplicationSettings": {
  "WebDevelopmentBehaviorEnabled": true,
  "Tenants": {
    "localhost": {
      "IdWWW": 1177,
      "RootFullPath": "D:\\Internet\\testProject\\",
      "ComponentsFolderName": "Components",
      "IsHttpSecured": true
    }
  }
}
```

Obrázek 11. Seznam tenantů v appsettings.json

5.2.1.1 ITenant

V systému je rozhraní `ITenant`. Toto rozhraní obsahuje mimo jiné nastavení webu, název domény, id webu v databázi a komponenty dostupné pro vykreslování obsahu webu. Nyní si uvedeme ty nejdůležitější property a vysvětlíme si jejich význam:

- `int IdWWW` – reprezentuje id webu uloženého v databázi, z databáze se načítají informace pouze spojené s tímto id
- `IDynamicComponents DynamicComponents` – jedná se o třídu, která obsahuje seznam dostupných komponent pro vykreslení obsahu na webu
- `WWWSettings WWWSettings` – tato třída obsahuje nastavení webu a informace o zobrazované jazykové mutaci

5.2.1.2 TenantMiddleware

Systém obsahuje vlastní middleware, který se stará o vytvoření a naplnění instance `ITenant`. Podle požadované domény se načte z appsettings příslušný tenant a následně se načtou všechny jeho data. Zde je ukázka funkce, která se o načtení tenanta stará:

```
public async Task Invoke(HttpContext context, IOptionsSnapshot<ApplicationSettings> options)
{
    context.Items[ConstHttpContextItemKeyTenant] = null;

    var applicationSettings = options?.Value;

    if (applicationSettings == null || applicationSettings.Tenants == null)
        throw new InvalidOperationException("Unable to read tenants from the application settings.");

    var domain = context.Request.Host.Value;
    var tenantKey = domain;
    string? mutationAlias = context.Request.Query[ConstRequestQueryStringKeyMutationAlias];
    if (mutationAlias == null)
        mutationAlias = "";

    //when developing, look for localhost tenant
    if (applicationSettings.WebDevelopmentBehaviorEnabled)
        tenantKey = "localhost";

    ITenant tenant = null;

    if (applicationSettings.Tenants.ContainsKey(tenantKey))
        tenant = applicationSettings.Tenants[tenantKey];
        tenant.WebDevelopmentBehaviorEnabled = applicationSettings.WebDevelopmentBehaviorEnabled;

    if (tenant == null)
        throw new AccessDeniedException($"Unable to get a tenant for the [{tenantKey}] domain - bad domain or tenant data.");
    else
    {
        tenant.DomainName = domain;
        tenant.DynamicComponents = this._componentService.GetTenantDynamicComponent(tenant.IdWWW);
        tenant.Components = (Dictionary<string, Type>)tenant.DynamicComponents.Components;
        tenant.WWWSettings = this._dataStorageAccess.SystemDataStorage.GetWWWSettings
                                                                        (
                                                                            tenant.IdWWW,
                                                                            tenant.GetWebBaseUrl(),
                                                                            mutationAlias,
                                                                            applicationSettings.WebDevelopmentBehaviorEnabled
                                                                        );
    }

    context.Items[ConstHttpContextItemKeyTenant] = tenant;
    await this._next.Invoke(context);
}
```

Obrázek 12. Funke Invoke ve třídě TenantMidleware

5.2.2 Dynamické načítání komponent

Multitenantní systém umožňuje pro každý web zobrazovat obsah pomocí vlastně definovaných Razor komponent. Jak bylo zmíněno již dříve, pro každý web je potřeba, aby vzniknul projekt s knihovnou Razor komponent, která se stará o zobrazení obsahu webu. Jakmile jsou šablony hotové, je potřeba provést kompilaci, a následně kompilovaný soubor dll nahrát do složky, která je uvedená v nastavení odpovídajícího tenanta viz. obrázek Seznam tenantů v appsettings.json. O načítání komponent pro všechny tenanty se stará služba s názvem ComponentService.

5.2.2.1 ComponentService

Služba ComponentService je při startu aplikace zaregistrována jako typ singleton. Tato služba se stará o to, aby pro všechny tenanty v systému načetla jejich knihovnu komponent.

Pro načtení komponent najdeme v této službě funkci `LoadDynamicComponents`. Tato funkce projde seznam všech tenantů v systému, načte jejich knihovnu komponent a uloží si tyto data do `Dictionary`, kde klíč je `IdWWW` tenanta a hodnota je načtená knihovna. Jedná se o poměrně náročnou operaci, proto je potřeba tuto funkci zavolat pouze jednou při startu aplikace. Při zpracování požadavků na zobrazení obsahu stránky jsou díky singleton službě tyto data uložené v paměti a není třeba tuto funkci znovu volat.

```
public void LoadDynamicComponents()
{
    IEnumerable<Type> component;
    string path;
    Assembly assembly;

    foreach (KeyValuePair<string, Tenant> tenant in _applicationSettings.Tenants)
    {
        if (!DynamicComponents.ContainsKey(tenant.Value.IdWWW))
        {
            path = tenant.Value.GetComponentsPath();
            assembly = LoadAssembly(path);

            component = GetTypeWithInterface(assembly);
            DynamicComponents.Add(tenant.Value.IdWWW, component);
        }
    }
}
```

Obrázek 13. Implementace funkce LoadDynamicComponents

Dále služba obsahuje funkci pro vrácení instance třídy `IDynamicComponents`, která obsahuje seznam všech dostupných Razor komponent v knihovně. Parametr pro načtení této instance je `IdWWW` tenanta. Díky této instanci jsme schopní vykreslovat pro každý web jiné komponenty. Tato funkce je volána ve třídě `TenantMiddleware` při načítání dat pro tenanta.

```
public IDynamicComponents GetTenantDynamicComponent(int IdWWW)
{
    return DynamicComponents[IdWWW].Select(x => (IDynamicComponents)Activator.CreateInstance(x))
        .SingleOrDefault(x => x.Components.Count > 0);
}
```

Obrázek 14. Implementace funkce GetDynamicComponent

5.2.3 Načítání dat z databáze

Aplikace načítá data z Microsoft SQL Server databáze. V projektu jsou implementované funkce, které volají SQL procedury s využitím technologie ADO.NET. Díky využití této technologie je systém chráněn před pokusy o SQL injection.

V systému je pro načítání veškerých dat z databáze vytvořena singleton služba `MsSqlDataStorageAccess`. Tato služba obsahuje přístup do databáze pro načítání obsahu webových stránek, přístup do databáze pro logování a obsahuje také přístup do cache paměti.

```
public IEnumerable<PageContentPlugin> LoadPageContent(int pageId, int contentColumnId, PageContentPluginType? onlyOnePlugin)
{
    using (var cmd = this.NewCommandProc("dbo.pubSelectWWWPageContent"))
    {
        cmd.AddParam("@IDWWW", this.IdWWW);
        cmd.AddParam("@IDWWWPage", pageId);
        cmd.AddParam("@IDWWWContentCols", contentColumnId);
        cmd.AddParam("@IDLanguage", this.IdLanguage);
        if (onlyOnePlugin != null) cmd.AddParam("@OnlyOneClass", onlyOnePlugin.Value);
        cmd.AddParam("@IsAdmin", this.IsAdminView ? 1 : null);

        using (var dataReader = this.ExecReader(cmd))
        {
            int templateNumber;
            while (dataReader.Read())
            {
                templateNumber = (int)dataReader["IDTemplateNum"];
                if (templateNumber < 0) templateNumber = 0;

                yield return new PageContentPlugin(
                        (PageContentPluginType)Enum.ToObject(typeof(PageContentPluginType), (int)dataReader["IDClassCollection"]),
                        templateNumber,
                        0,
                        (int)dataReader["IDWWWPageContent"],
                        (int)dataReader["IDWWWPage"],
                        dataReader["Title"] as string,
                        dataReader["Subtitle"] as string,
                        dataReader["DESCR"] as string,
                        dataReader["PictureFileAlias"] as string,
                        null
                    );
            }
        }
    }
}
```

Obrázek 15. Ukázková funkce pro načtení obsahu webové stránky z databáze

5.2.3.1 Cache paměť

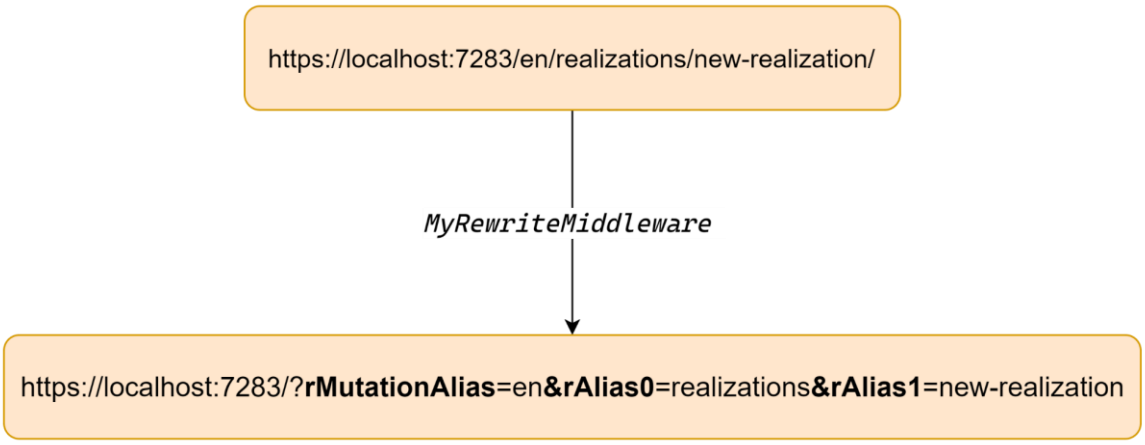
Pro optimalizaci rychlosti načítání webových stránek byla v systému implementovaná možnost ukládat a načítat data z cache paměti. Pro tuto funkcionalitu byla použita knihovna LazyCache, která umožňuje do cache paměti uložit data v páru klíč a hodnota. Systém implementuje ukládání dvou základních typů dat do cache paměti:

- Site scoped data – Jedná se o data, která jsou sdílená pro celý web hostovaný na tomto systému. Každý web obsahuje sadu systémových textů, které nejsou editovatelné přímo v obsahu stránky. Tyto textu jsou v systému načteny jednou procedurou pro celý web a jsou uloženy do cache paměti.
- Site part scoped data – Jedná se o data, která jsou sdílená pouze pro část webu hostovaném na tomto systému. Každá jazyková mutace webu obsahuje vlastní strom stránek, který slouží k vykreslení hlavičky webu a načítání hypertextových odkazů pro navigaci na webu. Z tohoto důvodu jsou data pro strom stránek uložena pro každou jazykovou mutaci zvlášť.

5.2.4 Zpracování HTTP požadavku pro zobrazení webové stránky

V této podkapitole si vysvětlíme, jak vypadá pipeline požadavku pro vykreslení obsahu webové stránky. Při vytváření klasického blazor projektu je pro každou stránku na webu vytvořená Razor komponenta obsahující direktivu @page. Musíme tedy v takovém projektu přesně nadefinovat stránky, které na webu existují. V tomto CMS systému má uživatel možnost vytvářet podstránky na webu bez omezení. Z tohoto důvodu není možné k vývoji přistupovat stejným způsobem.

V CMS systému dochází při zpracování HTTP požadavku k převedení požadované URL na query string. Systém obsahuje jeden end point pro vykreslování obsahu, který obsah vykreslí na základě query stringu. V produkčním prostředí je možné nastavit pravidla pro přepis URL například v systému IIS. Aby bylo možné webové stránky vyvíjet bez nutnosti konfigurovat web na IIS server, v systému vznikl vlastní middleware, který se stará o přepsání požadované URL na query string. Middleware má název `MyRewriteMiddleware` a dojde k jeho aplikování pouze ve vývojovém prostředí. Zde je příklad požadované URL a výsledná URL po aplikování `MyRewriteMiddleware`:



Obrázek 16. Příklad aplikování MyRewriteMiddlewre

Po přepsání URL dojde ke spuštění TenantMiddleware. Role, a to, jak tento middleware funguje jsme si už vysvětlili v kapitole TenantMiddleware.

Vykreslení obsahu následně vždy probíhá v Razor komponentě `App.razor`. Tato komponenta obsahuje pouze direktivu `@page "/"` a zavolnání funkce `@PageContent.RenderPage()`. Toto je náš end point pro vykreslení obsahu.

5.2.4.1 App.razor.cs

V projektu je vytvořen soubor `App.razor.cs`, který obsahuje kód odpovědný za správné načtení dat požadované stránky. Třída `App` je závislá na těchto službách:

- `IDataStorageAccess` – Obsah stránky je načítán z databáze.
- `ITenantAccess` – Obsah stránky je třeba vykreslit s použitím komponent definovaných pro konkrétního tenanta.
- `IHttpContextAccessor` – Alias požadované stránky je třeba načíst z query stringu HTTP požadavku.
- `IPageContentPluginParameters` – Tato služba obsahuje seznam objektů sdílených mezi všemi vykreslovanými pluginy obsahu. Více informací o této službě je uvedeno v podkapitole IPageContentPluginParameters

Tato třída obsahuje funkci `OnInitialized`, která inicializuje property `PageContent`.

```
protected override void OnInitialized()
{
    var page = GetPage();
    var urlAliases = GetUrlAliases(page);
    var pageTree = new PageTree(DataStorageAccess
                                .WebContentDataStorage
                                .LoadPagesDictionary(true),
                                TenantAccess);

    PluginParameters.Init(  DataStorageAccess,
                            TenantAccess,
                            httpContextAccessor,
                            page,
                            pageTree,
                            urlAliases);

    if (urlAliases.CheckAllData())
    {
        PageContent = new PageContent(  page.TemplateNum,
                                        urlAliases.ActState,
                                        PluginParameters,
                                        DataStorageAccess);
    }
    else
    {
        CheckAliasHistory(urlAliases.QueryAliases[0]);

        page = DataStorageAccess?.WebContentDataStorage.GetHomePage();
        PluginParameters.CurrentPage = page;
        PageContent = new PageContent(  PageContent.ErrorTemplateNum,
                                        PageContent.ErrorTemplateState,
                                        PluginParameters,
                                        DataStorageAccess);
    }
}
```

Obrázek 17. Funkce OnInitializet ve třídě App

Pokud se nepodaří najít v systému požadovanou webovou stránku, systém se nejprve podívá do historie aliasů stránek. O načtení historie se stará funkce `CheckAliasHistory`. Pokud dojde k nalezení záznamu v historii, systém provede redirect na aktuální webovou stránku. Pokud nedojde k nalezení záznamu v historii, systém zobrazí stránku s obsahem 404.

5.2.4.2 PageContentPlugin

Toto je třída, která implementuje základní property a funkce všech pluginů pro zobrazování obsahu na webu. Každý obsahový plugin je potomkem této třídy. Nyní si uvedeme některé základní property a funkce.

Každý obsahový plugin má k sobě odpovídající šablonu (Razor komponentu), která se stará o vykreslení. Třída `PageContentPlugin` proto obsahuje property `Template`, která reprezentuje název Razor komponenty. Název komponenty se skládá vždy ze tří parametrů:

- `PageContentPluginType TemplateName` – Odpovídá názvu obsahového pluginu
- `int TemplateNum` – Odpovídá čísle šablony použité pro vykreslení dat pluginu. Pokud budeme na webu zobrazovat galerii, můžeme chtít dva různé vzhledy galerie. První vzhled bude mít číslo šablony 0 a druhý vzhled bude mít číslo šablony 1.
- `int TemplateState` – Odpovídá hloubce zanoření obsahu. Pokud zobrazujeme obsah standardní stránky, `TemplateState` bude roven 0. Pokud zobrazujeme detail stránky (např. obsah stránky /realizations/new-realization/) bude `TemplateState` roven 1.

```
public string Template
{
    get
    {
        return $"{TemplateName}_{TemplateNum}_{TemplateState}";
    }
}


public PageContentPluginType TemplateName { get; protected set; }
public int TemplateNum { get; set; }
public int TemplateState { get; set; }
```

Obrázek 18. Ukázka pojmenování šablon pro vykreslení obsahu

Každý obsahový plugin potřebuje přístup k systémovým textům. Proto tato třída obsahuje implementaci funkce `GetLocalizedText`, která z databáze načte systémový text.

```
public string? GetLocalizedText(int idLanguage, string textCode)
{
    var identifier = LocalizedText.GetIdentifier(idLanguage, textCode);
    PluginParameters.DataStorageAccess.SystemDataStorage
        .GetLocalizedTexts(PluginParameters.TenantAccess.IdWWW, true)
        .TryGetValue(identifier, out var localizedText);

    if (localizedText != null)
        return localizedText.Text;

    return null;
}

public string? GetLocalizedText(SystemLanguageType languageType, string textCode)
{
    return GetLocalizedText((int)languageType, textCode);
}

public string? GetLocalizedText(string textCode)
{
    return GetLocalizedText(PluginParameters.TenantAccess.IdLanguage, textCode);
}
```

Obrázek 19. Ukázka funkce GetLocalizedText a její přetížení

5.2.4.3 PageContent

Třída `PageContent` se stará o vykreslování obsahu webové stránky. Tato třída je potomkem třídy `PageContentPlugin` a obsahuje rozšiřující funkce pro vykreslování obsahu. Tyto funkce mají vždy návratovou hodnotu `RenderFragment`, kterou je Razor komponenta schopná zpracovat a vykreslit její HTML obsah.

Nyní si popíšeme funkci `RenderPageContent`, která vykresluje obsah webové stránky vytvořený uživatelem pomocí admin rozhraní. Tato funkce z databáze načte seznam pluginů v obsahu stránky. Seznam pluginů je datového typu `IEnumerable<PageContentPlugin>`. Při procházení tohoto seznamu je vždy převeden `PageContentPlugin` na konkrétní instanci obsahového pluginu a následně dojde

k vykreslení odpovídající Razor komponenty. Razor komponentě je vždy předána instance pluginu jako parametr s názvem `Objref`.

```
public RenderFragment RenderPageContent(int contentColumnId) => builder =>
{
    var content = _dataStorageAccess.WebContentDataStorage
                                    .LoadPageContent(PluginParameters.CurrentPage.IdDB,
                                                     contentColumnId, null);
    PageContentPlugin? pageContentPlugin;
    Type? component;

    foreach (var plugin in content)
    {
        plugin.PluginParameters = PluginParameters;
        pageContentPlugin = GetPageContentPlugin(plugin);
        if (pageContentPlugin != null)
        {
            component = _dataStorageAccess
                            .TenantAccess
                            .Tenant
                            .GetComponent(pageContentPlugin.Template);

            if (component != null || _dataStorageAccess.TenantAccess.Tenant.WebDevelopmentBehaviorEnabled)
            {
                builder.OpenComponent(0, component);
                builder.AddAttribute(1, "Objref", pageContentPlugin);
                builder.CloseComponent();
            }
        }
    }
};
```

Obrázek 20. Ukázka implementace funkce RenderPageContent

5.2.4.4 IPageContentPluginParameters

Tato služba je v systému registrovaná jako ty scoped. Obsahuje instance objektů, které chceme zpřístupnit všem obsahovým pluginům v systému. Slouží k zjednodušení při vytváření instancí obsahových pluginů.

```
public interface IPageContentPluginParameters
{
    public IDataStorageAccess DataStorageAccess { get; set; }
    public ITenantAccess TenantAccess { get; set; }
    public IHttpContextAccessor ContextAccessor { get; set; }
    public Page CurrentPage { get; set; }
    public IPageTree PageTree { get; set; }
    public IUrlAliases UrlAliases { get; set; }

    public void Init(IDataStorageAccess dataStorageAccess,
                     ITenantAccess tenantAccess,
                     IHttpContextAccessor contextAccessor,
                     Page currentPage,
                     IPageTree pageTree,
                     IUrlAliases urlAliases);
}
```

Obrázek 21. Obsah rozhraní IPageContentPluginParameters

6 UKÁZKOVÁ IMPLEMENTACE WEBU

V rámci bakalářské práce byla zpracována implementace jednoho ukázkového webu, kde je demonstrována funkčnost systému. Webová stránka obsahuje dvě jazykové mutace. Budeme si demonstrovat implementaci layoutu webu a některé z implementovaných pluginů. Hotové CSS, JS soubory a HMTL markup byl poskytnut firmou Studio 9 s.r.o.

6.1 Šablona pro layout webové stránky

Jako první byla vytvořena Razor komponenta s názvem `MAIN_LAYOUT`, která obsahuje základní strukturu webové stránky. Tato komponenta také vykresluje meta tagy pro SEO. Data pro SEO jsou načítána z injektované služby `IPageContentPluginParameters`. Komponenta také vykresluje tagy pro načtení CSS a JS skriptů.

Standardně bychom tuto komponentu označili za layout a napsali bychom do ní direktivu `@inherits LayoutComponentBase`. Pak bychom mohli do komponenty pro vykreslení obsahu napsat direktivu `@layout MAIN_LAYOUT` a tím by se pro vykreslené stránky aplikoval tento layout. Layout je ale možné použít pouze v komponentách, které obsahují direktivu @page [20]. V našem projektu ale tyto direktivy nepoužíváme, proto není možné využít funkce @layout. Z toho důvodu tato komponenta neobsahuje direktivu `@inherits LayoutComponentBase`. Místo toho tato komponenta obsahuje parametr `RenderFragment ChildContent`, která umožňuje předat HTML obsah dovnitř komponenty.

6.2 Hlavička webu

Všechny stránky na tomto webu mají společnou hlavičku. Pro vykreslení hlavičky vznikla komponenta `PAGE_HEADER`. Komponenta přebírá instanci třídy `PageContent` a při vykreslování volá funkci `@PageContent.RenderTree(0, 0)`. Funkce `RenderTree(0,0)` vykresluje šablonu s názvem `PAGE_TREE_0_0`, která vykresluje navigační menu.

```
@foreach (var item in @Objref.PluginParameters.PageTree.GetPagesForMenu())
{
    <li>
        <a href="@Objref.PluginParameters.PageTree.GetPageUrl(item)" class="@GetActiveClass(item.Id)">
            @item.Name.ToLower()
        </a>
    </li>
}

@code {
    [Parameter]
    public webManagerCMS.Core.PageContentNS.Plugins.PageTree? Objref { get; set; }

    public string? GetActiveClass(int id)
    {
        if (id == Objref.PluginParameters.CurrentPage.Id) return "is-active";
        return null;
    }
}
```

Obrázek 22. Obsah Razor komponenty PAGE_TREE_0_0

6.3 Šablona pro vykreslení obsahu webové stránky

Ukázková webová stránka zahrnuje běžné pluginy a sekci realizací s detailním zobrazením pro každou realizaci. Pro zobrazení realizací byl použit plugin galerie, který automaticky generuje stránky s detaily pro každou z nich. Vznikly tedy dvě šablony pro vykreslení obsahu. První, s názvem `PAGE_CORE_0_0`, zobrazuje běžnou stránku a její obsah. Druhá, s názvem `PAGE_CORE_0_1`, zobrazuje detail realizace.

```
<MAIN_LAYOUT>
    <PAGE_HEADER Objref="Objref" />

    <div class="content-wrapper" id="content">
        @Objref.RenderPageContent(1)
    </div>

    <PAGE_FOOTER Objref="Objref" />
</MAIN_LAYOUT>

@code {
    [Parameter]
    public PageContent? Objref { get; set; }
}
```

Obrázek 23. Obsah Razor komponenty PAGE_CORE_0_0

```
<MAIN_LAYOUT>
    <PAGE_HEADER Objref="Objref" />

    <div class="content-wrapper" id="content">
        @Objref.RenderPageDetailContent()
    </div>

    <PAGE_FOOTER Objref="Objref" />
</MAIN_LAYOUT>

@code {
    [Parameter]
    public PageContent? Objref { get; set; }
}
```

Obrázek 24. Obsah Razor komponenty PAGE_COER_0_1

6.4 Plugin pro výpis realizací

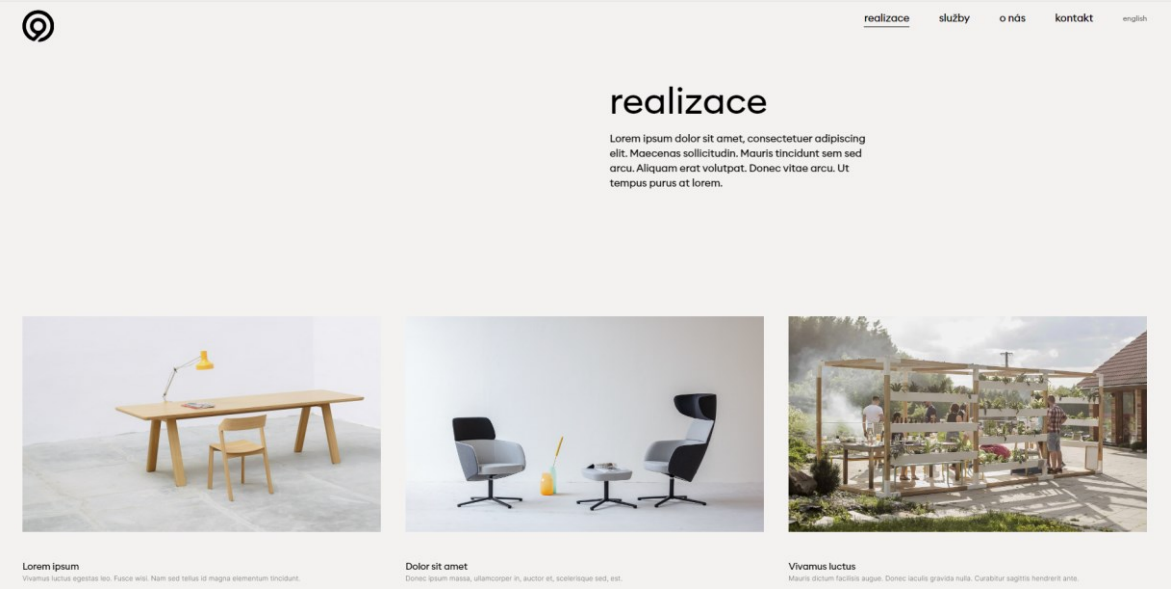
Na ukázkovém webu je možnost zobrazit realizace. Každá realizace má vlastní detail, proto byl pro zobrazení těchto dat použit plugin galerie, který umí automaticky generovat detail pro každou položku. Kdybychom nepoužili tento pluginy, uživatel by musel pro každou realizaci vytvářet stránku a v ní vyplnit její obsah. Nyní stačí pro vytvoření realizace vytvořit v pluginu galerie nový záznam a do záznamu vyplnit všechny data. Pro zobrazení realizací vznikly v pluginu dvě šablony. První šablona vykresluje seznam realizací a druhá vykresluje detail realizace.

6.4.1 Výpis realizací

Šablona s názvem `GALLERY1_1_0` slouží k zobrazení seznamu realizací. Pro optimalizaci doby načítání a plynulejší zobrazení využívá stream rendering, kdy samotné vykreslení dlaždic probíhá asynchronně.

```
<section class="section cards" id="@("C" + Objref.Id)">
    <div class="container">
        @if (!string.IsNullOrEmpty(Objref.Title))
        {
            <p class="section__title gs_reveal gs_reveal_fromLeft">@Objref.Title</p>
        }
        <div class="row">
            @if (gallery == null)
            {
                <div class="col-12">
                    <p><em>Loading...</em></p>
                </div>
            }
            else
            {
                @foreach (var item in gallery)
                {
                    <GALLERY_TILE   Picture="@item.Picture1" Url="@Objref.GetRowURL(item)"
                                    Title="@item.Title" Perex="@item.Perex" />
                }
            }
        </div>
    </div>
</section>

@code {
    [Parameter]
    public webManagerCMS.Core.PageContentNS.Plugins.Gallery? Objref { get; set; }

    public IEnumerable<GalleryRow>? gallery;

    protected override async Task OnInitializedAsync()
    {
        gallery = Objref.GetGalleryRows(100000);
    }
}
```

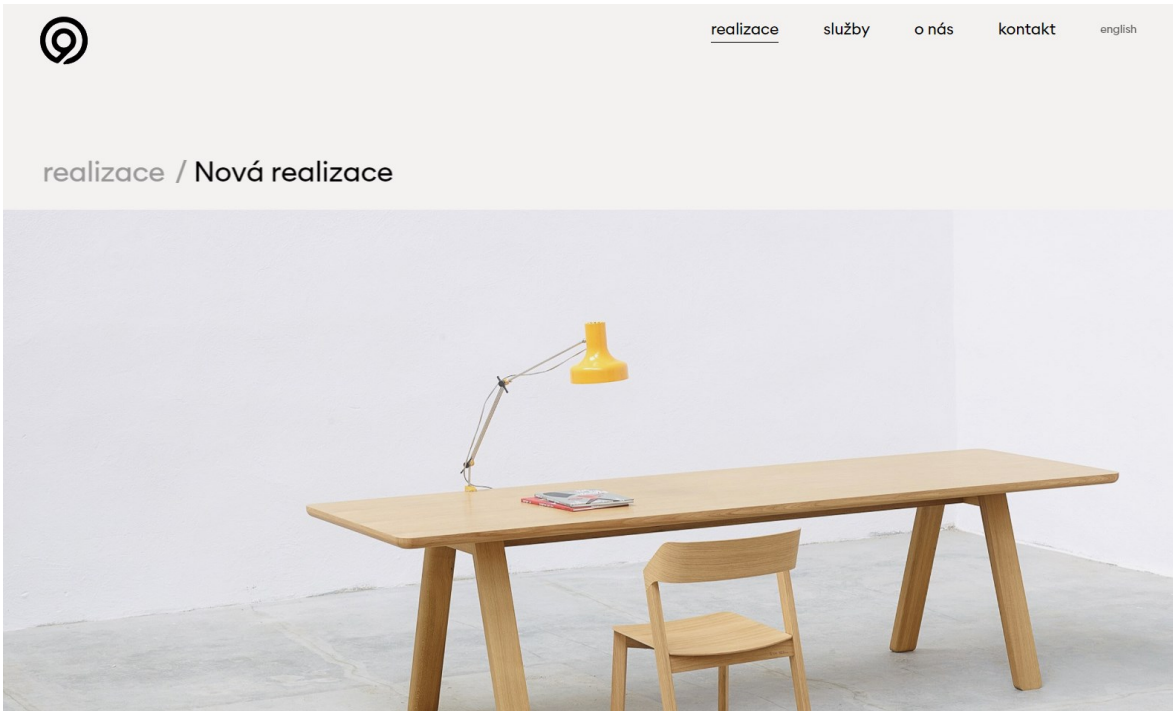
Obrázek 25. Obsah Razor komponenty GALLERY1_1_0



Obrázek 26. Stránka s výpisem realizací

6.4.2 Detail realizace

Pro detail realizace není potřeba vytvářet v systému stránku. Plugin galerie umožňuje zobrazit automaticky generovaný detail. URL detailu realizace obsahuje dvě úrovně. První úroveň je alias stránky, na které je plugin galerie vytvořen. Druhá úroveň je alias realizace. Zde je příklad URL detailu realizace: https://localhost:7283/realizace/nova-realizace/. Šablona pro vykreslení detailu má název `GALLERY1_1_1`.



Obrázek 27. Stránka s detailem realizace

ZÁVĚR

Výstupem praktické části bakalářské práce je systém pro renderování webových stránek vytvořených CMS systémem webManager. Systém byl vyvinut pomocí technologie ASP.NET Core Blazor Static Server. Toto téma bylo zvoleno, aby došlo k prozkoumání nové možnost statického server-side renderingu ve frameworku Blazor a možnosti tohoto frameworku při dynamickém načítání Razor komponent. Přínosem této práce je otevření možnosti pro aktualizaci zastaralého způsobu server-side renderingu systému webManager.

V teoretické části byl představen přehled CMS systémů. Dále byla představena platforma ASP.NET Core a její možnosti pro vývoj webových aplikací. Následně byla představena technologie Razor Pages a její základní syntaxe. Teoretická část se také z velké části zabývala frameworkem Blazor. Byl zde popsán princip tvorby webových stránek s použitím Razor komponent. Dále byl v práci uveden přehled módů renderování, které framework Blazor nabízí, a jejich porovnání. Také byl v práci uveden přístup frameworku Blazor k zabezpečení proti Cross-Site Scripting a SQL Injection.

V praktické části byla provedena analýza požadavků na vytvářený systém. Na základě analýzy byly zpracovány funkční a nefunkční požadavky. Dále bylo popsáno vytvoření projektu a jeho rozčlenění. Následně práce popisuje klíčové části systému a doplňuje tento popis obrázky s ukázkami kódu. Mezi klíčové části systému byla zařazena implementace multitenantního systému, systém pro dynamické načítání Razor komponent, spojení systému s databází a zpracování HTTP požadavku pro zobrazení webové stránky. V poslední kapitole je ukázána implementace webu pro demonstraci funkčnosti systému. Byla zde popsána tvorba základních Razor komponent pro vykreslení layoutu, hlavičky webu a výpis obsahu webové stránky. Jako poslední byla předvedena ukázková implementace pluginu s použitím stream renderingu pro výpis realizací na webu.

Nově vzniklý systém umožňuje tvorbu prezentačních webových stránek s ohledem na požadavky pro SEO a rychlost načítání. Stávající systém nabízí mnohem robustnější škálu funkcionalit pro tvorbu webových stránek, proto by budoucí vývoj systému mohl zahrnout implementaci některých z nich. Stávající systém nabízí například pluginy pro vývoj E-commerce aplikací, nebo pluginy pro rozdělení webu na prezentační a business část.

Při vývoji systému bylo největší výzvou implementace dynamického načítání Razor komponent. Framework Blazor není stavěný na tento koncept, kdy jsou pro několik webů načítány naprosto rozdílné komponenty, proto bylo náročné dojít k vhodnému řešení.

Implementace ukázkového webu byla v porovnání s implementací na starý systém velice pohodlná. Psaní Razor komponent, oproti psaní šablon ve formátu XML, nabízí spoustu nových možností, jako je debugování, napovídání našeptávačem IntelliSense a validace HTML kódu. Při vytváření systému pro server-side rendering autor došel k závěru, že tento framework je vhodný pro vývoj CMS systému.

SEZNAM POUŽITÉ LITERATURY

[1] *What is a CMS? A Guide to Content Management Systems.* Online. Mailchimp. C2001-2024. Dostupné z: https://mailchimp.com/resources/what-is-a-cms/. [cit. 2024-03-23].

[2] *What Is a Content Management System and How to Choose the Best CMS for Your Needs – Go WordPress.* Online. WordPress. 2018. Dostupné z: https://wordpress.com/go/tutorials/what-is-a-cms/. [cit. 2024-03-23].

[3] ŠTRÁFELDA, Jan. *Redakční systém (CMS) – vše, co chcete vědět [2024].* Online. Jan Štráfelda. 2024. Dostupné z: https://www.strafelda.cz/cms#open-source-cms. [cit. 2024-03-23].

[4] *Open source CMS platforms Pros & Cons*. Online. Agility CMS. 2017. Dostupné z: https://agilitycms.com/resources/posts/open-source-cms-the-benefits-and-the-risks. [cit. 2024-03-23].

[5] *Comparing Open-Source and Proprietary CMS Platforms*. Online. A WP Life. 2023. Dostupné z: https://awplife.com/comparing-open-source-and-proprietary-cms-platforms/. [cit. 2024-03-23].

[6] *Introduction to .NET*. Online. Microsoft. 2024, 10.1.2024. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/introduction. [cit. 2024-03-23].

[7] *.NET Managed languages strategy*. Online. Microsoft. 2023, 6.2.2023. Dostupné z: https://learn.microsoft.com/en-us/dotnet/fundamentals/languages. [cit. 2024-03-23].

[8] *.Net Core vs .Net Framework: Key Differences, Features, and more*. Online. Great Learning. C2013-2024, 14.7.2023. Dostupné z: https://www.mygreatlearning.com/blog/net-core-vs-net-framework/. [cit. 2024-03-23].

[9] *What is .NET Core*. Online. EDUCBA. 2023, 5.7.2023. Dostupné z: https://www.educba.com/what-is-dot-net-core/. [cit. 2024-03-23].

[10] *Overview of ASP.NET Core*. Online. Microsoft. 2023, 4.10.2023. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0. [cit. 2024-03-23].

[11] *MVC pattern*. Online. Microsoft. 2023, 26.9.2023. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0. [cit. 2024-03-23].

[12] *An Introduction To ASP.NET Razor Pages*. Online. BRIND, Mike. Learn Razor Pages. C2018-2024, 18.2.2021. Dostupné z: https://www.learnrazorpages.com/. [cit. 2024-03-23].

[13] *Design Patterns Explained – Dependency Injection*. Online. THORBEN, Janssen. Stackify. 2024, 15.3.2024. Dostupné z: https://stackify.com/dependency-injection/. [cit. 2024-03-26].

[14] *Dependency injection in ASP.NET Core*. Online. Microsoft. 2023, 11.7.2023. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0. [cit. 2024-03-26].

[15] *.NET dependency injection*. Online. Microsoft. 2024, 15.3.2024. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. [cit. 2024-03-29].

[16] *ASP.NET Core Middleware*. Online. Microsoft. 2023, 24.11.2023. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0. [cit. 2024-03-29].

[17] Open-source vs. proprietary software: How to choose the right CMS. Online. Brightspot. 2024, 14.3.2024. Dostupné z: https://www.brightspot.com/cms-resources/cms-selection-guide/open-source-vs-proprietary-cms. [cit. 2024-03-29].

[18] TAURIUS, Litvinavicius. *Exploring Blazor: Creating Hosted, Server-Side, and Client-Side Applications with C#.* Apress, 2019. ISBN 9781484254455. [cit. 2024-03-30].

[19] *ASP.NET Core Blazor*. Online. Microsoft. 2024, 9.2.2024. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0. [cit. 2024-03-30].

[20] *Components in Blazor.* Online. BRIND, Mike. Learn Blazor. 2023, 15.6.2023. Dostupné z: https://www.learnblazor.com/components. [cit. 2024-03-30].

[21] ASP.NET Core Razor components. Online. Microsoft. 2024, 9.2.2024. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-8.0. [cit. 2024-03-30].

[22] HIMSCHOOT, Peter. *Microsoft Blazor: Building Web Applications in .NET 6 and Beyond*. 3rd ed. Apress, 2021. ISBN 9781484278444. [cit. 2024-03-30].

[23] *.NET 8 and Blazor United / Server-side rendering*. Online. GIESEL, Steven. Steven Giesel. 2023, 4.10.2023. Dostupné z: https://steven-giesel.com/blogPost/3c71e553-490f-4d52-8104-eb8338c0207f. [cit. 2024-03-30].

[24] *Blazor Server vs. Blazor WebAssembly: Pros and cons of each approach*. Online. MOHANTY, Amit. C# Corner. 2023, 22.4.2023. Dostupné z: https://www.c-sharpcorner.com/article/blazor-server-vs-blazor-webassembly-pros-and-cons-of-each-approach/. [cit. 2024-03-31].

[25] *Exploring Blazor Changes in .NET 8 - Auto Render Mode*. Online. HILTON, Jon. Jon Hilton. 2023, 6.9.2023. Dostupné z: https://jonhilton.net/blazor-auto-render-mode/. [cit. 2024-03-31].

[26] *Seamless Rendering: The Magic of Blazor's Auto Render Mode in .NET 8.* Online. DONBAVAND, Daniel. Daniel Donbavand. 2023, 16.10.2023. Dostupné z: https://danieldonbavand.com/2023/10/16/blazors-auto-render-mode-in-net-8/. [cit. 2024-03-31].

[27] *Blazor in .NET 8: Server-side and Streaming Rendering*. Online. SAINTY, Chris. Chris Sainty. 2023, 22.8.2023. Dostupné z: https://chrissainty.com/blazor-in-dotnet-8-server-side-and-streaming-rendering/. [cit. 2024-03-31].

[28] *Prevent Cross-Site Scripting (XSS) in ASP.NET Core*. Online. Microsoft. 2023, 26.6.2023. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-8.0. [cit. 2024-03-31].

[29] *10 Best Practices to Secure ASP.NET Core MVC Web Applications*. Online. Syncfusion. 2023, 8.12.2023. Dostupné z: https://www.syncfusion.com/blogs/post/10-practices-secure-asp-net-core-mvc-app. [cit. 2024-03-31].

[30] *Introduction to Razor Pages in ASP.NET Core | Microsoft Learn* . Online. Microsoft. 2023, 10.7.2023. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-8.0. [cit. 2024-03-31].

[31] *Blazor School*. Online. C2021–2024. Dostupné z: https://blazorschool.com/. [cit. 2024-04-27].

[32] *Blazor | Build client web apps with C# | .NET*. Online. Microsoft. C2024. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor. [cit. 2024-04-27].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ASP	Active Server Pages
SQL	Structured Query Language
CMS	Content Management Systém
XML	Extensible Markup Language
HTML	Hyper Text Markup Language
COM	Common Object Model
API	Application Programming Interface
CSS	Cascading Style Sheets
PDF	Portable Document Format
MVC	Model View Controller
PHP	Hypertext Preprocesor
UI	User Interface
WASM	WebAssembly
SEO	Search Engine Optimization
JS	JavaScript

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

Příloha P I: CD s dokumentací práce a zdrojovým kódem vyvíjeného systému