Univerzita Hradec Králové

Fakulta informatiky a managementu

Webový Framework ASP.NET Core

Bakalářská práce

Autor: Tomáš Krámský

Studijní Obor: Aplikovaná Informatika

Vedoucí Práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové 2023

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne Tomáš Krámský

Poděkování

Rád bych poděkoval všem, v kruzích rodiny i přátel, kteří mě podporovali při psaní práce. Též bych rád poděkoval doc. Mgr. Tomáši Kozlovi, Ph.D. za velmi trpělivé vedení práce.

Anotace

Bakalářská práce poukazuje na důležitost webových frameworků pro vývoj backendových webových aplikací. Zaměřuje se na nástrahy tvorby vlastní implementace jednotlivých funkcí webových frameworků a vyzdvihuje použití již prověřených frameworků. Znázorňuje příklady některých těchto funkcí v ASP.NET Core, webovém frameworku pro platformu .NET od společnosti Microsoft. Na závěr ve zkratce zmíní některé další frameworky pro vývoj backendových webových aplikací pro další jazyky a platformy.

Klíčová slova: webové frameworky, dotnet, ASP.NET Core, webové backendy, webový vývoj

Annotation

Title: The ASP.NET Core Web Framework

This bachelor's thesis outlines the importance of backend web frameworks for the development of web applications. It focuses on the dangers of creating custom implementation of web framework features and promotes usage of battle-tested frameworks. It shows examples of this functionality in one such framework: ASP.NET Core, made by Microsoft for the .NET platform. Finally, it, in short, mentions other web frameworks for development of web backends for other languages and platforms.

Keywords: web frameworks, dotnet, ASP.NET Core, web backends, web development

Obsah

1. Úvod

V dnešní moderní době je stále více aspektů našich životů vedeno online skrze sociální sítě, fóra, e-shopy, webové portály a další webové služby stojící na technologiích, které byly jen pár desetiletí zpět zcela nemyslitelné. Přístup k internetu je dnes běžně považován za stejně důležitou utilitu, jako tekoucí voda nebo přístup k elektrické energii. Každou minutou vznikají a zanikají webové stránky a služby všech tvarů a velikostí, postavené na různých technologiích, s různými účely a požadavky.

V jejich vývoji jsou v ohromující většině případů využívány různé webové frameworky a knihovny, které poskytují hotová řešení pro běžné nebo komplikované problémy, jako je zpracovávání webových požadavků, vykreslování webových stránek nebo podpora autentizace. Jejich použití též umožňuje vývoj webových aplikací i lidem s menší úrovní zkušeností. Je sice dobré vědět, jak fungují http požadavky, ale pro použití některého z mnoha webových frameworků pro mnoho programovacích jazyků to zcela nutné není.

Jedním z populárních webových frameworků je ASP.NET Core, (neplést s ASP.NET nebo ASP). Právě tímto frameworkem se tato práce zabývá. Není však ani z daleka jediným a je vždy důležité vybrat správný nástroj pro daný úkol.

2. Webové frameworky a knihovny pro psaní backendů

Webový backend je název softwarové komponenty, která odbavuje webové požadavky. Běžné webové stránky se skládají z backendu, který běží na serveru, a frontendu, který běží u koncového klienta ve webovém prohlížeči. Frontend a backend spolu běžně komunikují za pomocí HTTP (Hyper Text Transfer Protocol). Backendy nemusí být vždy párovány s frontendy. Je možné mít např. backend poskytující HTTP API (Application Programming Interface) pro jiné aplikace. Stejně tak můžeme mít frontend, který nevyužívá žádný backend. (1)

Backendy jsou používány v případech, kdy potřebujeme webovým klientům poskytovat data nebo možnosti, ke kterým nemůžou přistoupit přímo z webového prohlížeče, nebo jim to naopak nechceme dovolit z bezpečnostních nebo jiných důvodů.

Úkolem webového backendu je odbavovat HTTP požadavky od klientů. Tento proces zahrnuje několik hlavních kroků a to:

- Naslouchání na TCP/UDP[1] portu pro příchozí požadavky
- Rozebrání příchozího požadavku od klienta na HTTP metodu, požadovanou cestu, verzi HTTP protokolu, jednotlivé HTTP hlavičky a případně tělo (2)
- Samotné zpracování požadavku
- Odeslání odpovědi klientovi

Práce se surovým HTTP protokolem ovšem může být dost náročná a do většiny projektů akorát přidává zbytečnou komplexitu a místa, kde může dojít k chybám. Proto je vhodné použít již existující implementaci.

Přijímání a odesílání HTTP požadavků není ale jediná věc, která se ve světě webových backendů stále opakuje. Mnoho dalších problémů je v případě vlastní implementace nutno řešit znovu a znovu, např. manipulace s cookies, autentizace, validace požadavků, práce s relacemi apod. Pokud si nechceme všechny tyto věci vyvíjet sami, je vhodné použít některý z mnoha webových frameworků nebo knihoven.

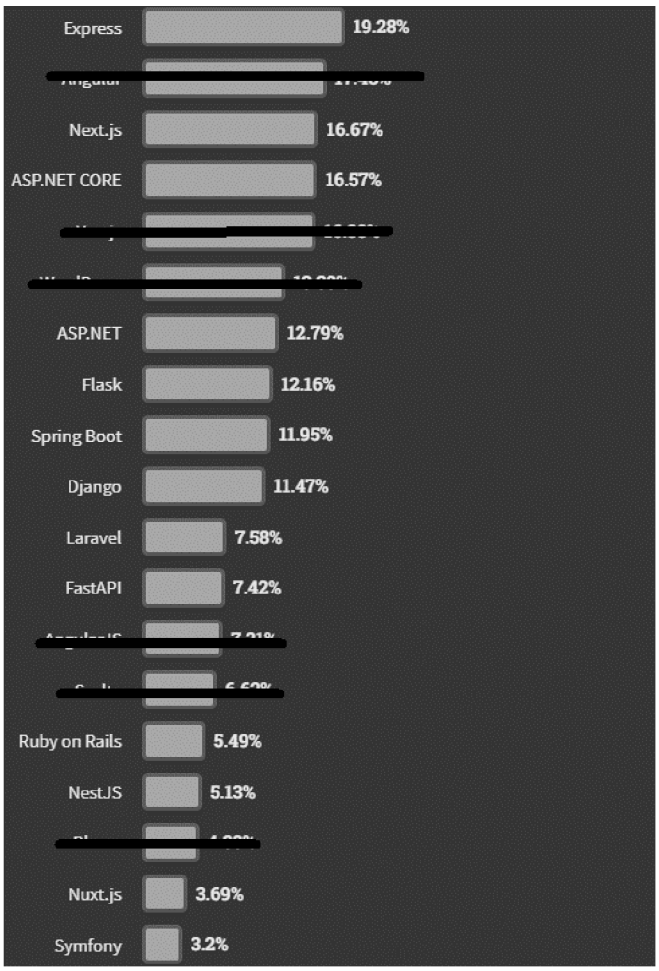
Různé webové frameworky/knihovny mohou poskytovat různé možnosti a různé úrovně abstrakce nad webovými požadavky, které odbavují. Některé, jako třeba ASP.NET Core, se snaží

[1] HTTP/3 využívá UDP pro komunikaci

popostrčit vývojáře do určitého způsobu psaní backendů, což je výhodné v tom, že různé projekty dělané ve stejném frameworku, vypadají podobně/stejně. To zjednodušuje vývoj a integraci nových vývojářů, kteří již mají zkušenost s daným frameworkem do existujících projektů.

Autentizace a autorizace (AuthN a AuthZ) je možnost velmi často buď integrovaná nebo dostupná formou pluginu/dodatkové knihovny. Použití hotových řešení je pro situace, kdy takové věci potřebujeme, velmi výhodné, jelikož vyžadují pro dobrou implementaci znalosti kryptografie. Pokus o vlastní implementaci by mohl velice jednoduše vést k problémům s bezpečností. Populární frameworky jsou používány velkým množství vývojářů, díky čemuž jsou výrazně testovány. (1)

Obrázek 1 ukazuje nejpopulárnější backendové webové frameworky. Po vyškrtání technologií, které nepatří mezi backendové webové frameworky můžeme vidět, že nejpopulárnější je javascriptový framework Express.js, blízce následován frameworkem ASP.NET Core.



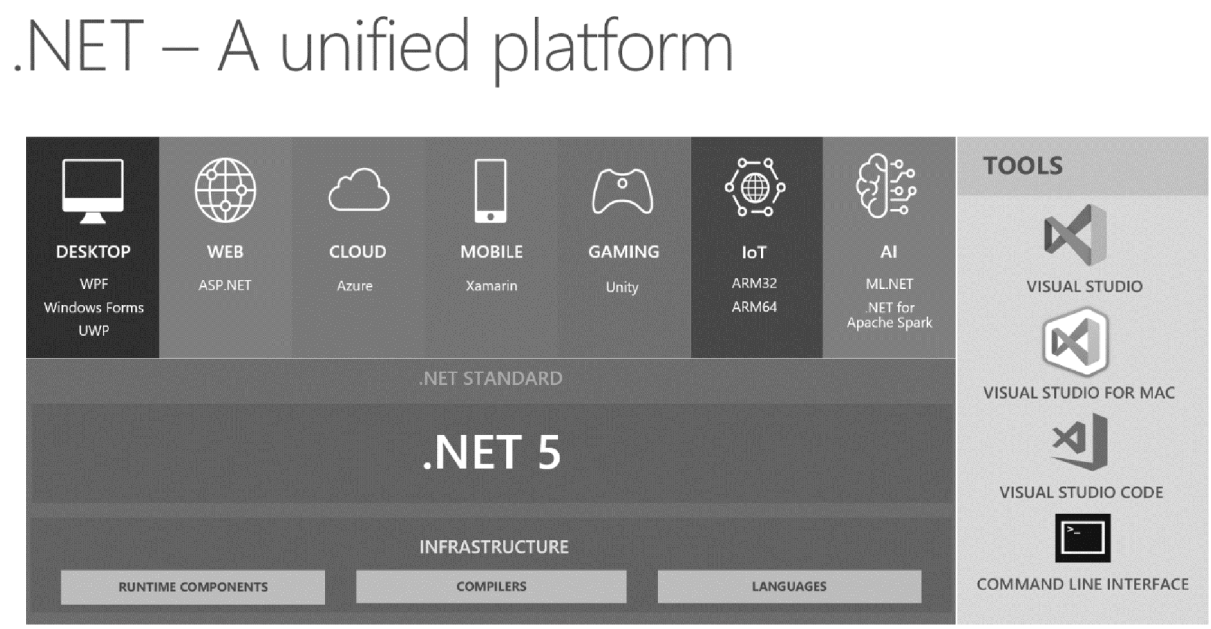
Obrázek 1 Stack Overflow Developer Survey 2023, most populat Web frameworks and technologies (3)

3. ASP.NET Core

ASP.NET Core je multi-platformní (často též nazýván cross-platform) framework pro platformu Dotnet od společnosti Microsoft určený pro tvorbu webových aplikací a služeb.

3.1. Dotnet

Dotnet (.NET) je open-source platforma vyvíjená primárně společností Microsoft pro vývoj multi-platformích[2] aplikací. Lze použít k vývoji mnoha typů aplikací, např.: desktopové aplikace, webové aplikace, mobilní aplikace, hry, IoT (Internet of Things) aplikace, aplikace využívající AI (Artificial Intelligence). Skládá se z nástrojů, programovacích jazyků a knihoven. Podporuje tři programovací jazyky: C# je objektově orientovaný a nejznámější ze všech tří, F# je funkcionální/hybridní (primárně funkcionální, ale podporuje objektově orientovaný přístup) (4) a VisualBasic.NET je objektově orientovaný jako C#, ale není zdaleka tak populární a jeho vývoj již skončil. (5) S Dotnet platformou je také často spojováno vývojové prostředí Visual Studio (též vyvíjeno společností Microsoft), které umožňuje rychlejší a snazší vývoj a ladění aplikací. (4)



Obrázek 2 .NET Platforma (4)

[2] Multi-platformní aplikace mohou být zkompilovány/spuštěny na více jak jedné platformě. V případě .NET jsou podporované platformy Windows, Linux, macOS, Android, Apple IOS, Apple TV a Apple IPad (70)

3.1.1. Historie

Dotnet (původně .NET Framework) byl nejdříve vyvíjen Microsoftem pro tvorbu aplikací pro operační systém Microsoft Windows. Velkou výhodou této skutečnosti byla vysoká úroveň integrace a velice dobré vývojové nástroje. Nevýhodou je fakt, že .NET Framework není cross-platformní, takže aplikace psané nad ním nejsou kompatibilní s jinými operačními systémy. Existují možnosti, jak kompilovat a pouštět Dotnetové aplikace na jiných operačních systémech ať už díky vrstvě kompatibility Wine (6), nebo platformě Mono (open-source implementace .NET) (7), ovšem tyto možnosti nemají zdaleka tak dobrou podporu jako .NET Framework samotný.

Dalším důležitým milníkem bylo představení .NET Core. Nové verze Dotnet platformy nevázané na operačním systému Windows. Její vývoj běžel nějakou dobu paralelně s vývojem .NET Framework. .NET Core 1.0 byl představen v době .NET Framework 4.6 a v době .NET Core 2.2 byla vydána poslední verze .NET Framework, a to verze .NET Framework 4.8. .NET Core je také první verze Dotnetu, která je open-source. .NET Framework open-source není. (8)

Posledním krokem je .NET 5 (a .NET 6, který je v době psaní v módu preview), což je první verze .NET Core, která má úplně nahradit .NET Framework. (9) Je velice jednoduché se ve verzování ztratit. .NET Frameworku se říkalo .NET, .NET Core se naštěstí vždy říkalo .NET Core a jednu dobu se .NET 5 říkalo .NET vNext (obecné označení používané pro další verzi Dotnetu). Teď se .NET Frameworku říká .NET Framework, .NET Core zůstává .NET Core a novému .NET 5 se říká jen .NET.

Důležité je také zmínit existenci .NET Standard, který byl představen společně s .NET Core a byl primárně určen k psaní knihoven kompatibilních jak s .NET Core, tak s .NET Framework. (10)

3.1.2. Kompilace

Jazyky platformy Dotnet se kompilují do CIL (Common Intermediate Language), ve starších verzích .NET známý také jako MSIL (Microsoft Intermediate Language), nebo obecněji pouze IL (Intermediate Language) (11), který je poté možno spustit pomocí CLR (Common Language Runtime). (12) Fakt, že výsledkem kompilace je IL, a nikoliv nativní strojový kód jako např. v případě jazyku C++, umožňuje spuštění jednoho kompilačního artefaktu (tj. výstup kompilace zahrnující položky jako .dll soubory obsahující IL kód, konfigurační soubory a další soubory

potřebné k běhu aplikace) na jakémkoliv operačního systému a architektuře procesoru která je podporována platformou .NET. (13)

Nevýhodou je ovšem nutnost mít na cílovém systému nainstalovaný Dotnet Runtime, bez kteréhož takto zkompilovanou aplikaci není možné spustit. Výjimkou tohoto pravidla je tzv. self-contained build jehož výsledkem je nejen IL, ale i runtime pro vybranou platformu potřebný ke spuštění aplikace, což umožní aplikaci spustit i na systému, kde Dotnet Runtime není nainstalovaný za cenu podstatně většího objemu souborů. (13)

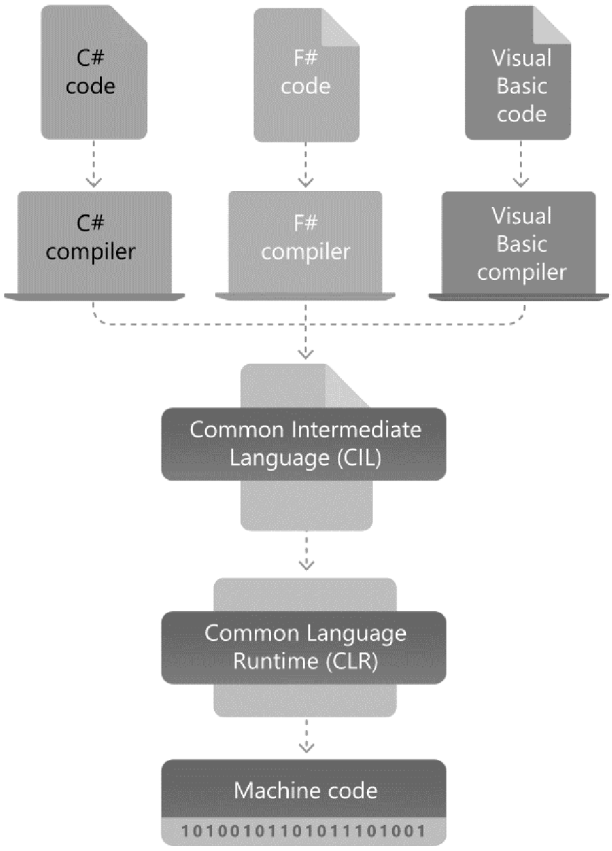
Další možností kompilace je vyprodukovat pouze jeden soubor. Standardní kompilace vyprodukuje pro každý projekt i použitou knihovnu jeden nebo více .dll souborů. Je proto běžné že kompilačních artefaktů kompilace jsou desítky až stovky. Toto nemusí být např. pro koncové uživatele úplně přívětivé, a proto existuje tato možnost, která umožňuje snížit počet kompilačních artefaktů na pouhý jeden. Nevýhodou této možnosti je, že výsledný spustitelný soubor je vždy závislý na určité platformě, protože není možné vygenerovat jeden soubor, který by byl spustitelný na více platformách. (14)

3.1.3. Běh aplikace

Dotnet aplikace se spouštějí v běhovém prostředí CLR (Common Language Runtime), jehož funkcionalita je srovnatelná s běhovým prostředím pro Java bytekód JVM (Java Virtual Machine). Má na starost automatickou správu paměti pomocí GC (Garbage Collector/Garbage Collection), poskytuje základní datové typy, poskytuje infrastrukturu pro podporu výjimek, a především prostředky pro zpracování CIL kódu. Výhodou GC je automatické uvolňování nepoužívané paměti, což zjednodušuje proces vývoje aplikace. Nevýhodou je nutnost čas od času pozastavit běh aplikace, aby ke GC mohlo dojít a pokud z nějakého důvodu ke GC dochází velmi často, může se v některých extrémních situacích stát, že CLR stráví více času prováděním GC než prováděním programu samotného. (15)

Hlavním úkolem CLR je vykonávat instrukce specifikované v CIL kódu. Ten není prováděn přímo, ale je dále kompilován pomocí komponentu CLR nazývaného JIT (Just In Time) kompilátor, který převede CIL kód na nativní strojový kód platformy, na které je aplikace spuštěna, přičemž může provést různé dodatečné optimalizace specifické pro danou platformu. Poté je výsledný nativní strojový kód teprve spuštěn. Výhodou takovéhoto přístupu je portabilita IL kódu, ovšem nevýhodou je nutnost kompilace IL kódu do nativního strojového kódu před jeho použitím, což

znamená pomalejší start aplikace. Aplikace není zkompilována JIT kompilátorem celá najednou, ale po částech v závislosti na tom, co je zrovna potřeba. (16) Tento problém s rychlostí startu je možné do určité míry vyřešit tzv. R2R (Ready to Run) kompilací, cože je forma AOT (Ahead Of Time) kompilace, při které se provede část práce JIT kompilátoru jíž při kompilaci ze zdrojového kódu do IL. Výsledkem je větší aplikace, protože musí obsahovat jak IL kód, tak R2R nativní kód, což také znamená, že pokud používáme R2R kompilaci, tak musíme udělat pro každou platformu, kterou cílíme, zvlášť build. (17)



Obrázek 3 Proces kompilace od zdrojového kódu ke strojového kódu (18)

3.2. Funkcionalita

3.2.1. Dependency injection (DI)

ASP.NET Core je celý postaven na dependency injection (injekci dependencí). Jedná se o návrhový vzor implementující princip *Inversion of Control* (IoC), který říká, že komponenty aplikace by měly záviset na abstrakci místo na specifických implementacích. (19)

DI se, ale nepoužívá jen kvůli splnění jednoho principu softwarového vývoje. Jeho hlavní výhodou je možnost jednoduché centralizované správy služeb a závislostí mezi nimi. Představme si, že máme službu *Database* reprezentující spojení s databází, dále službu *Repository* poskytující funkcionalitu pro provádění operací nad databází a službu *UserService* implementující operace s uživatelem. Závislosti mezi těmito třídami můžou vypadat nějak takto: *UserService -> Repository -> Database*. Pokud všechny služby akceptují svoje závislosti pomocí parametru v konstruktoru, tak by pak vytvoření validní instance třídy *UserService* vypadala nějak takto:

```
class Consumer
{
	private readonly UserService _userService;

	public Consumer()
	{
		_userService = new UserService(new Repository(new Database()));
	}
}
```

Tento příklad je ještě docela jednoduchý, ale s většími grafy závislostí si můžeme představit, jak by tento přístup rychle nabobtnal. S DI můžeme toho samého docílit takto:

```
services.AddScoped<Database>();
services.AddScoped<Repository>();
services.AddScoped<UserService>();

...

class Consumer
{
	private readonly UserService _userService;

	public Consumer(UserService userService)
	{
		_userService = userService;
	}
}
```

Takto DI vyřeší graf závislostí za nás a centralizuje logiku pro tvoření nových instancí. Velice jednoduše také můžeme pro *Repository* definovat rozhraní *IRepository,* což nám v kombinaci s DI umožní v případě potřeby nahradit službu jen na jednom místě:

```
interface IRepository
{
	...
}

class RepositoryImpl1 : IRepository
{
	...
}

class RepositoryImpl2 : IRepository
{
	...
}

...

//services.AddScoped<IRepository, RepositoryImpl1>();
services.AddScoped<IRepository, RepositoryImpl2>();
```

Díky již zmíněnému principu Inversion of Control je možné nahradit implementaci jakéhokoliv rozhraní, což v kombinaci s faktem že ASP.NET Core samotný systém DI vysoce využívá umožňuje jednoduše upravovat i chování samotného ASP.NET Core. (20)

3.2.1.1. Scope

V různých aplikacích může být DI scope kolem různých věcí. Většinou obaluje provádění nějaké operace. Zajišťuje, aby se po dokončení operace uvolnily zdroje asociované s jejím provedením. V ASP.NET Core je scope jeden webový požadavek. Každý webový požadavek dostane svůj scope, ve kterém se vykonává, a ze kterého si vyžaduje služby. (20)

3.2.1.2. Lifetimes

Lifetime (životnost) služeb určuje, jak dlouho bude instance služby existovat. V ASP.NET Core existují 3 životnosti (20):

- **Singleton** – po celou dobu běhu webové aplikace se používá stále jedna a ta samá instance. Instance je uvolněna na konci běhu aplikace
- **Scoped** – nová instance služby pro každý scope. Uvnitř jednoho scope se stále používá jedna a ta samá instance. Instance je uvolněna na konci života scope, ze kterého byla vyžádána
- **Transient** – nová instance pokaždé, když je žádána. I v rámci jednoho scope se vytvoří nová kdykoliv je žádána. Instance je uvolněna na konci života scope, ze kterého byla vyžádána

3.2.1.3. Registrace služeb

Služby se do DI registrují při startu aplikace přes rozhraní *IServiceCollection*. Přesné místo v kódu, kde se služby registrují bude zmíněno později.

Je důležité služby registrovat pod správným typem/rozhraním. V následujícím příkladu bude služba dostupná pod svým typem *MyService*, ale ne pod rozhraním *IMyService*:

```
interface IMyService
{
	void DoSomething();
}

class MyService : IMyService
{
	public void DoSomething()
	{
		...
	}
}

...

services.AddSingleton<MyService>();
```

Pokud chceme používat službu přes rozhraní *IMyService*, musíme pod ním také tuto službu zaregistrovat:

```
services.AddSingleton<IMyService, MyService>();
```

První generický parametr určuje, pod jakým typem/rozhraním bude služba zaregistrovaná a druhý generický parametr určuje samotnou implementační třídu služby.

Služby nejsou ničím speciální. Jsou to prosté .NET třídy implementující požadovanou funkcionalitu (20).

3.2.1.4. Metody injekce

Injekce dependencí je proces, při kterém DI Kontejner poskytne požadované služby nějakou z následující metod.

3.2.1.4.1. Injekce do konstruktoru

Nejběžnější a nejvíce používanou metodou injekce služeb z hlediska uživatele frameworku je injekce do konstruktoru. S touto metodou jednoduše deklarujeme služby, které chceme používat v konstruktoru třídy a DI Kontejner při konstrukci třídy automaticky požadované služby poskytne:

```
class Consumer
{
	private readonly ILogger<Consumer> _logger;
	private readonly IService _service;

	public Consumer(ILogger<Consumer> logger, IService service)
	{
		_logger = logger;
		_service = service;
	}
}
```

3.2.1.4.2. Injekce do metody

V některých případech můžeme použít injekci do metody s pomocí atributu *[FromServices]*. Tento přístup je ovšem omezení pouze na místa, kde je to podporováno jako akce kontrolérů nebo lambda funkce u Minimal APIs (vysvětleno později):

```
[HttpGet(“/“)]
public IActionResult Index(
	[FromServices]ILogger<MyController> logger,
	[FromServices]IService service)
{
	...
}
```

3.2.1.4.3. Manuálně

V některých případech můžeme mít potřebu si o nějakou službu zažádat manuálně bez použití automatické injekce do konstruktoru nebo metody. V těchto případech můžeme použít rozhraní *IServiceProvider*, které je možné injektovat jako dependency, nebo je dostupná třeba z *HttpContext.RequestServices*:

```
// získání IServiceProvider přes injekci do konstruktoru a použití
class Consumer
{
	private readonly IServiceProvider _serviceProvider;

	public Consumer(IServiceProvider serviceProvider)
	{
		_serviceProvider = serviceProvider;
	}

	public void DoSomething()
	{
		var service = _serviceProvider.GetRequiredService<IService>();
	}
}

...

// získání IServiceProvider z HttpContextu v kontroléru a použití
class MyController : Controller
{
	[HttpGet(“/“)]
	public IActionResult Index()
	{
		var service = HttpContext.RequestServices
					.GetRequiredService<IService>();
		...
	}
}
```

3.2.1.5. DI Kontejnery

Této “krabici“ která obsahuje všechny naše služby se také často říká DI Container (Dependency Injection Container) nebo IoC Container (Inversion of Control Container). V případě ASP.NET Core je výchozí implementací DI Kontejneru *Microsoft.Extensions.DependencyInjection*, ale je možné používat i jiné DI Kontejnery poskytující extra funkcionalitu nebo výkon. (20)

3.2.2. Middleware

Middleware jsou základní stavební bloky, ze kterých se v ASP.NET Core a mnoha dalších webových frameworcích skládá pipeline pro zpracování webových požadavků. Tyto bloky si můžeme představit jako sekvenci kroků, kde každý může provést nějakou operaci nad webovým požadavek a buď ho odbavit a poslat zpátky skrz předchozí middleware v opačném pořadí (kde každý middleware, který zpracoval webový požadavek při cestě tam dostane příležitost zpracovat webový požadavek také při cestě zpátky) nebo ho předat dalšímu middleware v pořadí.

Pro každý webový požadavek je vytvořena instance třídy *`HttpContext`*, která obsahuje webový požadavek (*`HttpRequest`*), odpověď na něj (*`HttpResponse`*) a další relevantní data jako třeba informace o podléhajícím síťovém spojení přes které je požadavek prováděn nebo informace o uživateli, který požadavek provádí.

Middleware může dělat cokoliv. Má plný přístup, jak k webovému požadavku, tak k případné odpovědi na něj. Může také využívat služeb z dependency injection.

Příkladem je třeba autentizační middleware, který z webového požadavku zjistí identitu uživatele v závislosti na nakonfigurovaných možností autentizace (např. pomocí autentizační cookies) a výsledek zapíše do vlastnosti *`HttpContext.User`*, logovací middleware, který loguje webové požadavky nebo routovací middleware, který má na starost nasměrování požadavku ke správnému kusu uživatelského kódu, který požadavek odbaví. (21)

3.2.3. Konfigurace Aplikace

3.2.3.1. Starý způsob

Do .NET verze 6 se běžné webové aplikace konfigurovaly pomocí tzv. *`Startup`* třídy. Příklad takové třídy je možné vidět zde:

```
class Startup
{
	public void ConfigureServices(IServiceCollection services)
	{
		services.AddRazorPages();
		services.AddAuthorization()
		   .ConfigureApplicationCookie(cookie =>
		   {
		       cookie.ExpireTimeSpan = TimeSpan.FromHours(1);
		   });

		services.AddMvc()
		   .AddCookieTempDataProvider()
		   .AddXmlSerializerFormatters()
		   .AddMvcLocalization();
	}

	public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
	{
		app.UseDeveloperExceptionPage();

		app.UseStaticFiles();

		app.UseRouting();

		app.UseAuthentication();
		app.UseAuthorization();

		app.UseEndpoints(endpoints =>
		{
			endpoints.MapRazorPages();
			endpoints.MapControllers();
		});
	}
}
```

Ve třídě Startup máme 2 metody. Metoda *ConfigureServices(IServiceCollection services)* slouží k přidání služeb do dependency injection. ASP.NET Core poskytuje mnoho rozšiřujících metod pro jednoduchou registraci služeb potřebných k použití jednotlivých komponentů. V příkladu můžeme vidět třeba metodu *AddRazorPages(),* která přidá služby potřebné k používání Razor Pages v naší aplikaci. Tyto metody jsou konvencí ve formátu *Add** např. *AddRazorPages*, *AddMvc* nebo *AddAuthentication*.

V ASP.NET Core se na mnoha místech používají pro konfiguraci lambda funkce. Toto je možné vidět ve volání metody *ConfigureApplicationCookie,* kde se pomocí konfiguračního lambda funkce k nastavení doby validity autentizační cookie nebo ve volání metody UseEndpoints, která přidává kontroléry a Razor stránky do systému pro směrování webových požadavků. Také se

často v konfiguraci využívá návrhový vzor *fluent interface*, který spočívá v řetězení volaných metod. Toto je možné vidět u volání metody *AddMvc()* a volání následujících metod.

Metoda *Configure(IApplicationBuilder app, IWebHostEnvironment env)* dovoluje nakonfigurovat samotný pipeline pro zpracování webových požadavků. Příchozí požadavky budou postupně zpracovány pomocí middleware v takovém pořadí, v jakém jsou přidány v této metodě. V příkladu si můžeme povšimnout např. middleware pro odbavování požadavků na statické soubory (*UseStaticFiles*), middleware pro routing (*UseRouting*) nebo pár middlewarů pro autentizaci a autorizaci (*UseAuthentication*, *UseAuthorization*).

Startup třída je sama o sobě k ničemu. Slouží pouze jako konfigurace, podle které bude webová aplikace vytvořena. Toho je ve výchozím ASP.NET Core vzoru docíleno následovně (22):

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

3.2.3.2. Nový způsob

Od .NET verze 6 existuje nový, jednodušší způsob, jak nakonfigurovat a zapnout webovou aplikaci. Stále je možné používat starý způsob, ale Microsoft doporučuje používat tento nový který vypadá takto (23):

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

// Add services to the container.
builder.Services.AddRazorPages();
builder.services.AddAuthorization()
       .ConfigureApplicationCookie(cookie =>
       {
              cookie.ExpireTimeSpan = TimeSpan.FromHours(1);
       });

builder.Services.AddMvc()
       .AddCookieTempDataProvider()
       .AddXmlSerializerFormatters()
       .AddMvcLocalization();

var app = builder.Build();

// Configure the HTTP request pipeline.
App.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapControllers();

app.Run();
```

Kód v tomto příkladu docílí stejného výsledku jako kód ve starém příkladu, ale je podstatně kompaktnější.

3.2.4. Logování

Logování je zprostředkováno balíčky *Microsoft.Extensions.Logging.**. Tyto balíčky poskytují užitečnou abstrakci nad logováním v podobě rozhraní *ILogger<T>,* kde generický parametr T je typ, pro který chceme logovat, což poskytuje logovacímu systému informaci o zdroji logové zprávy (tzv. kategorii), která je ve formátu celého názvu typu T i se jmenným prostorem, např. typ *HomeController* ve jmenném prostoru *ExampleSite.Controllers* by měl kategorii *ExampleSite.Controllers.HomeController*. Tato informace je poté využívána v konfiguraci logování k filtrování logových zpráv. Je tak možné jednoduše nastavit, že třeba chceme, aby logové zprávy ze jmenného prostoru *Microsoft* byly zapisovány, jen pokud mají úroveň varování nebo vyšší a logové zprávy z naší aplikace, pokud mají úroveň informace nebo vyšší.

Existuje celkem 6 úrovní logových zpráv s postupně stoupající důležitostí (24):

- *Trace*
- *Debug*
- *Informational*
- *Warning*
- *Error*
- *Critical*

Logování je samozřejmě také zaintegrováno do dependency injection systému, takže pokud máme např. třídu *HomeController* a chceme v ní využít logování, tak stačí pomocí injekce do konstruktoru požádat o službu typu *ILogger<HomeController>* následovně:

```
public class HomeController
{
	private readonly ILogger<HomeController> _logger;
	public HomeController(ILogger<HomeController> logger)
	{
		_logger = logger;
	}
}
```

Microsoft přímo vyvíjí některé cíle pro logování jako např. konzole nebo Windows Event Log. Je také možné implementovat si vlastní cíl pro logování.

Tento logovací systém je velmi flexibilní a umožňuje použití logovacích knihoven třetích stran jako *Log4Net* a *Serilog*, které mají svoje vlastní cíle pro logování a často i více možností než samotný *Microsoft.Extensions.Logging*.

Důležité je zmínit podporu tzv. strukturovaného logování, které umožňuje do logových zpráv umístit pojmenované hodnoty, se kterými je pak možné v podporovaných logovacích cílech pracovat. Takto obohacenou logovou zprávu je možno zapsat takto (24):

```
int value = 123;
_logger.LogInformation(“The value is {Value}“, value);
```

Příkladem logovacího cíle, který podporuje strukturované logové zprávy je systém Seq (25) od společnosti Datalust. Systém zaznamená nejen zformátovanou logovou zprávu, ale hodnotou,

se kterou byla zaslána a následně je možné podle této hodnoty vyhledávat a filtrovat logy. V případě logování do konzole není strukturované logování moc užitečné, ale stéle aspoň způsobí zapsaní hodnoty do konzole jinou barvou.

3.2.5. Konfigurace

Konfigurace je zprostředkována balíčky *Microsoft.Extensions.Configuration.**. Tyto balíčky poskytují možnosti pro čtení konfigurace z různých zdrojů a bindování konfigurace na .NET objekty. Je tak možné nadefinovat následující třídu:

```
class MyConfiguration
{
	public string Url { get; set; }
	public int Port { get; set; }
	public bool UseSSL { get; set; }
}
```

A poté načíst konfiguraci z následujícího JSON souboru:

```
{
	“url“: “https://example.com“,
	“port“: 8080,
	“useSSL“: true
}
```

JSON samozřejmě není jediný podporovaný zdroj konfigurace. Existuje mnoho dalších jako např. proměnné prostředí, XML, parametry z příkazové řádky, Azure Key Vault a mnoho dalších. Některé zdroje konfigurace podporují automatické znovunačtení konfigurace při změně. Také je možné implementovat si vlastní zdroj konfigurace (26).

Konfiguraci poté můžeme použít pomocí injekce do konstruktoru následovně:

```
public class HomeController
{
	private readonly IOptions<MyConfiguration> _options;

	public HomeController(IOptions<MyConfiguration> options)
	{
		_options = options;
	}
}
```

V místě *IOptions<T>* můžeme použít jedno z následujících možností (26):

- `IOptions<T>` - získá instanci konfigurace tak jak vypadala při startu aplikace
- `IOptionsSnapshot<T>` - získá instanci konfigurace při každém webovém požadavku znovu. Pokud se, ale konfigurace změní v průběhu požadavku, tak nedojde k aktualizaci tohoto snapshotu. Toto je užitečné, když chceme zajistit, aby se respektovalo znovunačtení konfigurace, ale nechceme, aby se konfigurace změnila uprostřed zpracovávání požadavku
- `IOptionsMonitor<T>` - získá aktuální stav konfigurace a podporuje znovunačtení konfigurace

3.2.6. Možnosti hostování

Samotné zpracování webových požadavků by bylo k ničemu, kdybychom je nemohli přijímat z venkovního světa. K tomu účelu slouží servery/moduly implementující HTTP protokol. ASP.NET Core nabízí oficiálně 3.

3.2.6.1. Kestrel

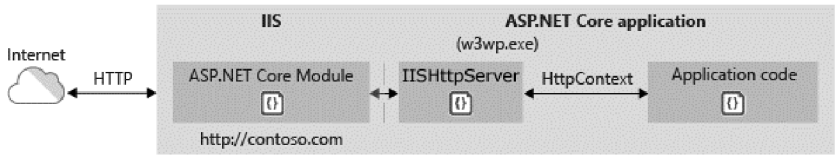
První z možností pro hostování naší webové aplikace psané v ASP.NET Core je webový server Kestrel od společnosti Microsoft, který je distribuován společně s frameworkem. ASP.NET Core.

Implementuje protokoly HTTP verze 1.1, verze 2 a nově od ASP.NET Core 7 i verzi 3. V tomto případě je webový server přímo vestavěn do naší webové aplikace a je distribuován jako její součást. (27)

3.2.6.2. IIS

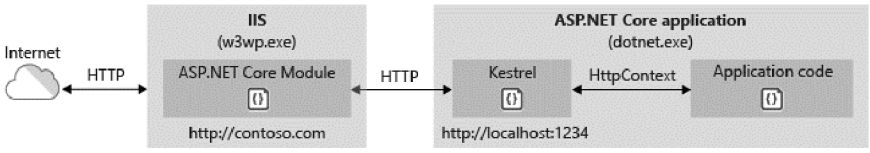
Internet Information Services (IIS) je webový server od společnosti Microsoft pro OS Microsoft Windows. Pro hostování ASP.NET Core webových aplikací je nutné doinstalovat dodatečný modul. Aplikace může být hostována buď “in-process“ nebo “out-of-process“. (28)

V hostovacím modelu in-process je instance naší webové aplikace spuštěna uvnitř pracovního procesu IIS serveru díky čemuž jí může IIS server předávat příchozí webové požadavky přímo viz. Obrázek 4 IIS in-process hostování. Tento hostovací model je výkonnější než out-of-process model. (29)



Obrázek 4 IIS in-process hostování (29)

V hostovacím modelu out-of-process je instance naší webové aplikace spuštěna v odděleném procesu za pomocí serveru Kestrel a webové požadavky jsou jí přeposílány přes loopback síťové rozhraní (speciální rozhraní které vrací odchozí data zpět tomu samému systému. Dá se říct, že si tak systém "povídá sám se sebou") na náhodný port viz. Obrázek 5 IIS out-of-process hostování. (30)



Obrázek 5 IIS out-of-process hostování (30)

3.2.6.3. HTTP.SYS

HTTP.SYS je webový server, který funguje pouze na OS Microsoft Windows a je s ním úzce spjatý. Seznam funkcí, které podporuje se odvíjí od verze OS. Rozdílů mezi serverem Kestrel a HTTP.SYS je sice více, ale z pohledu tématu je důležité jen to, že Kestrel je rychlejší, a hlavně není závislý na OS Microsoft Windows. (31)

3.2.7. Model binding

Model binding umožňuje navazovat data z webových požadavků přímo na .NET typy bez nutnosti manuálního čtení dat z požadavku. Je možné bindovat, jak jednoduché typy (int, string, bool…), tak komplexní typy (třídy), ale ne všechny typy jde bindovat ze všech zdrojů. ASP.NET Core je v mnoha situacích dostatečně inteligentní, aby sám určil odkud se má jaká hodnota bindovat. V následujících příkladech ale bude vždy využito `[From*]` atributů, která přesně specifikují odkud se má hodnota brát.

Data je možné bindovat z (32):

- Cesty
- Těla
- Query parametrů
- HTTP hlaviček
- Formuláře

3.2.7.1. Z cesty

V následujícím příkladu bude parametr nastaven z URL z místa, kde je zmíněn pomocí *{id}*, takže pro cestu */people/123* by parametr id měl hodnotu *123* (32):

```
[HttpGet(“/people/{id}“)]
public IActionResult Get([FromRoute]int id)
{
	//...
}
```

Je také možno specifikovat tzv. *route constraint*, který omezí, co je možné do cesty vsadit např.:

```
[HttpGet(“/people/{id:int}“)] //omezení pouze na hodnoty datového typu int
...

[HttpGet(“/people/{id:min(1)}“)] //omezení an minimální hodnotu 1
...

[HttpGet(“/people/{id:regex(^[0-9]*$)}“)] //omezení pomocí regex výrazu
...
```

3.2.7.2. Z těla

U typů požadavků, které tělo podporují, jako HTTP POST, PUT, PATCH atd. V následujícím případu uvažujme, že někde máme definovanou třídu *Person* a anotací parametru atributem *[FromBody]* vyjádříme, že chceme, aby se na instanci třídy *Person* nabindovali data těla z webového požadavku. Nutno podotknout, že pokud chceme získat hodnoty z odeslaného formuláře, musíme použít atribut *[FromForm]* popsaný níže. *[FromBody]* slouží k získání dat např. z JSON nebo XML těla (32):

```
class Person
{
	public string Name { get; set; }
	public int Age { get; set; }
}

...

[HttpPost(“/people“)
public IActionResult Create([FromBody]Person person)
{
	//...
}
```

3.2.7.3. Z query parametrů

Všechny primitivní parametry, které nejsou nabindovány na parametry z cesty jsou nabindovány na query parametry. V následujícím příkladu by požadavek na cestu */people?page=2&searchQuery=bob* způsobil zavolání této metody s parametry *page = 2* a *searchQuery = “bob“* (32):

```
[HttpGet(“/people“)]
public IActionResult List([FromQuery]int page, [FromQuery]string searchQuery)
{
	// ...
}
```

3.2.7.4. Z HTTP Hlaviček

```
[HttpGet(“/people“)]
Public IActionResult List([FromHeader(Name = “X-My-Header“)]string header)
{
	// ...
}
```

3.2.7.5. Z Formuláře

```
class Person
{
	public string Name { get; set; }
	public int Age { get; set; }
}

...

[HttpPost(“/people“)
public IActionResult Create([FromForm]Person person)
{
	//...
}
```

3.2.8. Razor

Razor syntax umožňuje generovat HTML v ASP.NET Core webových aplikacích. Skládá se z HTML a C# kódu. Razor soubory mají buď příponu *.cshtml* (Razor View), nebo *.razor* (Razor Component). Při kompilaci je tento kód přeložen na .NET třídu.

V následujícím kódu je možno vidět jednoduchý příklad Razor syntaxe. Na začátku je kódový blok, do kterého můžeme psát běžný C# kód. Pod ním se nachází HTML kód, v kombinaci s proměnnými a smyčkou *foreach*. Další syntaxe jako *while*, *for*, *if* nebo *switch* jsou také podporovány. I při výpisu proměnné do HTML dokumentu můžeme využívat běžných C# metod jako třeba metody *ToUpper()* která změní všechny písma v textovém řetězci na velká. Na konci je demonstrován komentář (33):

```
@{
	string name = “bob“;

	string[] items = new []
	{
		“item 1“,
		“item 2“,
		“item 2“
	};
}

<p>Name: @bob</p>

<ul>
	@foreach (var item in items)
	{
		<li>@item.ToUpper()</li>
	}
</ul>

@* razor comment *@
```

Pokud používáme Razor syntaxe pro Razor Pages, MVC nebo Minimal APIs, můžeme specifikovat tzv. model, který reprezentuje typ instance třídy, kterou později specifikujeme při renderování HTML:

```
@model Person

<p>Name: @Model.FirstName @Model.LastName</p>
```

Razor automaticky nahrazuje nebezpečné znaky za bezpečné enkódované alternativy. Třeba znak ostré závorky „*<*“ by byl nahrazen bezpečně enkódovaným znakem „*<*“. Pokud toto chování chceme obejít, můžeme využít pomocné metody *Html.Raw()* (33):

```
@{
	string text = “<h1>hello world</h1>“;
}

@Html.Raw(text)
```

Podporované jsou také tzv. layouts (rozložení) a partials (částečné zobrazení), která umožňují znovupoužití částí HTML. Částečné zobrazení dovoluje definici částí HTML v oddělených souborech (ve výchozí konfiguraci ve složce */Views/Shared/*), které je následovně možné využívat v jakýchkoliv jiných Razor souborech. Konvencí se částečné zobrazení prefixují podtržítkem. Takový soubor může vypadat třeba takto (34):

```
@* _Header.cshtml *@
<header>
	<nav>
		<a href=“/“>Index</a>
		<a href=“/o-nas“>O nás </a>
		<a href=“/kontakt“>Kontakt</a>
	</nav>
</header>
```

A lze ho použít takto:

```
@Html.RenderPartial(“_Header“)
```

Rozložení dovoluje definovat “šablonu“, do které se pak dosadí další HTML. Stejně jako částečná zobrazení se ve výchozí konfiguraci dávají do složky */Views/Shared*. Takový soubor může vypadat třeba takto (35):

```
@* _Layout.cshtml *@
<!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="utf-8" />
</head>
<body>
	@RenderBody()
</body>
```

Rozložení lze použít následovně:

```
@{
       Layout = “_Layout“;
}

<p>hello world</p>
```

Paragraf s textem *hello world* bude pří renderování umístěn tam, kde jsme v rozložení zavolali metodu *RenderBody()*.

Existují také dva speciální soubory a to *_ViewImports.cshtml* a *_ViewStart.cshtml*.

_ViewImports.cshtml slouží k definici sdílených importů jmenných prostorů pro všechny Razor soubory. Může vypadat třeba takto (36):

```
@* _ViewImports.cshtml *@
@using MyWebApplication
@using MyWebApplication.Models
```

Díky tomuto souboru budou jmenné prostory *MyWebApplication* a *MyWebApplication.Models* dostupné ve všech ostatních Razor pohledech automaticky.

_ViewStart.cshtml slouží k puštění kódu na začátku každého Razor souboru. Běžně se používá k definici výchozího rozložení, aby nebylo nutné ho definovat v každém Razor souboru zvlášť (37):

```
@{
    Layout = "_Layout";
}
```

Pokud takto definujeme rozložení, tak bude použito ve všech Razor souborech, což zahrnuje i soubor rozložený samotný. Abychom předešli problému, kde se soubor rozložený snaží používat sám sebe jako šablonu, tak je nutné na začátek souboru rozložení nastavit vlastnost *Layout* na hodnotu *null*:

```
@* _Layout.cshtml *@
@{
	Layout = null;
}
<!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="utf-8" />
</head>
<body>
	@RenderBody()
</body>
```

3.2.8.1. Tag Helpers

Takzvané “Tag Helpery“ umožňují modifikaci chování HTML tagů v Razor syntaxi. Několik vestavěných Tag Helperů je dostupných ve jmenných prostorech *Microsoft.AspNetCore.Mvc.TagHelpers* a *AuthoringTagHelpers*. (38)

Abychom je však mohli použít, musíme je nejprve zpřístupnit použitím razor direktiva @addTagHelper:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

Příkladem Tag Helperu, který nyní lze využít je třeba Script Tag Helper, který umožní použít na <script> tagu extra atributy *asp-src-include* pro zahrnutí všech scriptů, které odpovídají poskytnutému vzoru (38):

```
<script asp-src-include=“scripts/*.js“></script>
```

Kdyby složka *scripts* obsahovala soubory *script.js*, *hello.js* a *test.js,* tak by výsledek vykreslení Razor stránky vypadal následovně:

```
<script src=“~/scripts/script.js“></script>
<script src=“~/scripts/hello.js“></script>
<script src=“~/scripts/test.js“></script>
```

Tag Helpery můžeme definovat vlastní. Stačí vytvořit třídu která dědí ze třídy *Microsoft.AspNetCore.Razor.TagHelpers.TagHelper* a Implementovat metodu *Process* nebo *ProcessAsync* (39):

```
namespace MyWebApp;

[HtmlTargetElement("div", Attributes = "asp-hello")]
public class HelloTagHelper : TagHelper
{
    [HtmlAttributeName("asp-hello")]
    public string Name { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.SuppressOutput();
        output.Content.AppendHtml($"<div>hello {Name}</div>");
    }
}
```

Atributem *[HtmlTargetElement]* specifikujeme, jaký html tag, s jakými atributy je cílem tohoto Tag Helperu. Pouze takto definované HTML tagy budou ovlivněny. Atributem *[HtmlAttributeName]* na vlastnosti *Name* definujeme, že hodnota html atributu *asp-hello* má být nastavena do vlastnosti *Name*.

Atribut poté můžeme použít následovně (39):

```
@addTagHelper *, MyWebApp

<div asp-hello="bob"></div>
```

Což vyprodukuje následující HTML kód:

```
<div>hello bob</div>
```

3.2.9. Ostatní

ASP.NET Core poskytuje spoustu dalších možností, jako například lokalizaci (podporu více jazyků), sessions (ukládání dočasných informací o uživateli na straně serveru), caching (ukládání výsledků výpočetně nebo časově náročných operací do mezipaměti a jejich následné využití), jednoduchou manipulaci s cookies a mnoho dalších. Velké množství informací lze nalézt přímo v oficiální dokumentaci dostupné na webové adrese:

https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core

3.3. Způsoby vývoje

Jeden způsob vývoje aplikace není vždy vhodný pro všechny problémy. Proto existuje víc možností jak framework ASP.NET Core používat. Některé způsoby jsou určené k vývoji webových API, některé pro tvorbu běžných webových stránek a jiné i k dalším účelům blíže popsaných v této kapitole.

3.3.1. Minimal APIs

Nejjednodušší možností, jak použít ASP.NET Core pro odbavování webových požadavků, jsou tzv. Minimal APIs. Jedná se o zápis velice podobný Javascriptovým frameworkům jako express.js dostupný od verze ASP.NET Core 6 a vypadá následovně (40):

```
app.MapGet(“/“, () => “Hello world“);

app.MapGet(“/add“, (int a, int b) => a + b);

app.MapGet(“/greet/{name}“, (string name) => $“Hello, {name}!“;

app.MapPost(“/form“,
	([FromForm]MyForm form, [FromServices]ILogger<Program> logger) =>
	{
		logger.LogInformation(
			“form submitted: {Email} {Message}“,
			form.Email,
			form.Message
	);
});
```

Existují *Map** metody pro všechny běžně HTTP metody jako GET, POST, PUT, DELETE atd. Můžeme si povšimnout využití již známého model binding atributu *[FromForm]* a dependency injection atributu *[FromServices]*.

Minimal APIs umí automaticky vhodně zformátovat odpověď podle vraceného datového typu, např. při vrácení datového typu string automaticky vyprodukuje odpověď HTTP 200 OK s obsahem textového řetězce v těle odpovědi, nebo při vracení komplexního typu (třídy) se jí pokusí automaticky serializovat do formátu JSON (40).

Někdy samozřejmě potřebujeme větší kontrolu nad odpovědí k čemuž slouží rozhraní *IResult* reprezentující výsledek webového požadavku. Statická třída *Results* poskytuje přístup k některým předdefinovaným odpovědím:

```
app.MapGet(“/people/{id}“, (int id) =>
{
	if (id == 1)
	{
		// vrátí HTTP 200 OK s objektem serializovaný do formátu JSON
		return Results.Ok(new Person { Id = 1, Name = “Bob“ });
	}
	else
	{
		// vrátí HTTP 404 Not Found
		return Results.NotFound();
	}
});
```

Můžeme si také udělat vlastní implementaci rozhraní *IResult*. Třeba tuto, která vrátí HTTP 200 OK se specifikovanou zprávou:

```
class MyResult : IResult
{
	private readonly string _message;

	public MyResult(string message)
	{
		_message = message;
	}

	public async Task ExecuteAsync(HttpContext httpContext)
	{
		httpContext.Response.StatusCode = 200;
		byte[] bytes = Encoding.ASCII.GetBytes(_message);
		await httpContext.Response.BodyWriter.WriteAsync(bytes);
	}
}
```

Pro jednoduchost používání pak můžeme vytvořit statickou třídu definující rozšiřovací metodu:

```
static class ResultsExtensions
{
	public static IResult MyResult(this IResultExtensions ext, string message)
	{
		return new MyResult(message);
	}
}
```

A pak ji použít takto:

```
// bez statické třídy s rozšiřovací metodou
app.MapGet(“/“, () => new MyResult(“hello world“));

// se statickou třídou s rozšiřovací metodou
app.MapGet(“/“, () => Results.Extensions.MyResult(“hello world“));
```

3.3.2. MVC

ASP.NET Core MVC umožňuje vyvíjet aplikace pomocí návrhového vzoru stejného jména, MVC (Model, View, Controller). (41)

V kontextu ASP.NET Core je tento návrhový vzor implementován následovně:

```
// MODEL
class LoginModel
{
	[DisplayName(“jméno“)]
	public string Username { get; set; }

	[DisplayName(“heslo“)]
	public string Password { get; set; }

	public bool LoginFailed { get; set; }
}
```

```
@* VIEW *@
@model LoginModel

<form asp-action=“Login“>
	<label asp-for=“Username“></label>
	<input asp-for=“Username“ />
	<label asp-for=“Password“></label>
	<input asp-for=“Password“ type=“password“/>
	<button type=“submit“>Log In</button>
</form>
```

```
// CONTROLLER
public class AuthController : Controller
{
	[HttpGet]
	Public IActionResult()
	{
		return View(new LoginModel());
	}

	[HttpPost]
	public IActionResult Login(LoginModel model)
	{
		var success = model.Username == “admin“ && model.Password == “admin“;
		if (!success)
		{
			return View(new LoginModel { LoginFailed = true };
		}

		return Redirect(“/“);
	}
}
```

Třída AuthController musí dědit ze třídy *Controller* (nebo *ControllerBase*). Její jméno končí slovem “Controller“, ovšem toto pojmenování je pouze jmenná konvence, kterou není nutné následovat pro správné fungování aplikace.

Můžeme si zde povšimnout využití tag helperu *asp-for*, který pro html tagy <label> automaticky nastaví popis a pro html tagy <input> automaticky nastaví html atributy *name* a *type*. V modelu si mimo jiné můžeme povšimnout atributu *DisplayName*, který umožňuje nastavit jméno, které se promítne v html tagu <label> díky tag helperu.

Směrování požadavků je pro controllery řešeno buď tzv. konvenčním směrováním (convention routing) nebo pomocí atributů (attribute routing). V obou případech musíme controllery zaregistrovat při startu aplikace:

```
services.AddControllers();

...

app.UseRouting();

// pro attribute routing
app.MapControllers();
//pro convention routing
app.MapControllerRoute("default", "{controller}/{action}");
```

V případě attribute routingu definujeme cesty pomocí atributů. Výsledná cesta je kombinací cesty definované na úrovni třídy a cesty definované na úrovni akce (42):

```
[Route(“/“)]
class HomeController : Controller
{
	[HttpGet]
	public IActionResult Route1() => Ok(); // cesta: GET /

	[HttpPost]
	public IActionResult Route2() => Ok(); // cesta: POST /

	[HttpGet(“hello“)]
	public IActionResult Route3() => Ok(); // cesta: GET /hello

}

[Route(“/subpath“)]
class SecondController : Controller
{
	[HttpGet]
	public IActionResult Route4() => Ok();  // cesta: GET /subpath

	[HttpGet(“hello“)]
	public IActionResult Route5() => Ok();  // cesta: GET /subpath/hello
}
```

V případě convention routingu definujeme cestu v registraci při startu a cesty k jednotlivým akcím jsou definovány konvencí (42):

```
//startup
app.MapControllerRoute("default", "{controller}/{action}");

...

class HomeController : Controller
{
	public IActionResult Route1() => Ok(); // cesta: /Home/Route1

	[HttpGet]
	public IActionResult Route2() => Ok(); // cesta: GET /Home/Route2

	[HttpPost]
	public IActionResult Route3() => Ok(); // cesta: POST /Home/Route3

	[ActionName(“Hello“)]
	public IActionResult Route4() => Ok(); // cesta: /Home/Hello

}
```

Při použití convention routingu stále můžeme používat atributy *Http** (*HttpGet*, *HttpPost* atd.), ovšem můžeme je použít pouze k omezení HTTP metody, nikoliv k úpravě cesty. Můžeme stále použít atribut *ActionName*, který umožní změnu názvu akce z výchozí hodnoty (názvu metody) na hodnotu vlastní. (42)

3.3.3. Web API

Minimal APIs je jedna možnost, jak vyvíjet API v ASP.NET Core, ovšem není jediná. Druhou možností je využít controllery z MVC s pár změnami. Přidáme atribut *ApiController*, controller dědí ze třídy *ControllerBase* místo *Controller* a místo *IActionResult* v mnoha případech používáme *ActionResult<T>* jako návratovou hodnotu akcí controlleru:

```
[ApiController]
[Route(“/api/todo“)
class TodoController : ControllerBase
{
	[HttpGet]
	public ActionResult<List<Todo>> GetAll() => Ok();

	[HttpPost]
	public ActionResult<Todo> Create() => Ok();

	[HttpGet(“{id:int}“)]
	public ActionResult<Todo> Get(int id) => Ok();

	[HttpDelete(“{id:int}“)]
	public IActionResult Delete(int id) => Ok();
}
```

Použití návratového typu *ActionResult<T>* umožňuje vracet komplexní typy přímo bez nutnosti explicitního použití metody, která vrací IActionResult a který by vyprodukoval odpověď v žádaném formátu (43):

```
public ActionResult<Todo> GetById(int id)
{
	var todo = _todos.FirstOrDefault(todo => todo.Id == id);
	if (todo is null)
	{
		return NotFound();
	}

	// vracíme todo přímo. Neobalujeme ho v metodě jako např. Json(todo)
	return todo;
}
```

Třída *Controller* dědí ze třídy *ControllerBase* a poskytuje navíc funkcionalitu, jako např. přístup k session, view datům, metodám k renderování razor View jako výsledku akce apod. Tato funkcionalita není potřeba pro vývoj API, takže se ve většině případů používá odlehčená základní třída *ControllerBase*.

Atribut *[ApiController]* na controlleru způsobí (43):

- Nutnost použití attribute routingu. Convention routing nebude fungovat
- Automatickou odpověď http 400 v případě, kdy model není validní
- Automatické vracení chybových http status kódů ve formátu definovaném RFC 7807 (44)
- Úpravu pravidel pro binding parametrů akcí controlleru

Některá tato chování lze upravit při startu aplikace:

```
builder.Services.AddMvc().ConfigureApiBehaviorOptions(api =>
{
    api.SuppressModelStateInvalidFilter = true;
    api.SuppressMapClientErrors = true;
    api.SuppressInferBindingSourcesForParameters = true;
    api.SuppressConsumesConstraintForFormFileParameters = true;
    api.DisableImplicitFromServicesParameters = true;
});
```

3.3.4. Razor Pages

Razor Pages jsou alternativní možností pro vývoj jednodušších webových aplikacích s Razor Views. Oproti MVC, kde je model a controller oddělen, je v Razor Pages model a "controller" to samé. Stejně jako MVC musíme Razor Pages zaregistrovat při startu aplikace: (45)

```
services.AddRazorPages();

...

app.MapRazorPages();
```

Každá Razor Page se skládá ze 2 částí, a to Razor View a tzv. Page Model:

```
@* Razor View *@
@page "/login"
@model HomeModel

<form method="post">
      <label asp-for="Username"></label>
      <input asp-for="Username" />
      <label asp-for="Password"></label>
      <input asp-for="Password" type="password"/>
      <button type="submit">Log In</button>
</form>
```

```
// Page Model
class HomeModel : PageModel
{
	public string Username { get; set; }

	public string Password { get; set; }

	public bool Failed { get; set; }

	public void OnGet()
	{
	}

	public PageResult OnPost()
	{
		Failed = Username == “admin“ && Password == “admin“;
		if (Failed)
		{
			return Page();
		}

		return Redirect(“/“);
	}

}
```

Cesta je definovaná v Razor View pomocí Razor direktiva *@page*. Page Model obsahuje dvě speciální metody, *OnGet* a *OnPost*. Tyto metody budou konvencí zavolány při GET a POST požadavku na tuto stránku. Můžeme přidat i metody pro další HTTP metody např. *OnDelete()*.

Tyto metody nemusejí nic vracet, přičemž se po jejich provedení vyrenderuje korespondující Razor View, nebo můžu vracet *IActionResult* stejně jako MVC controller akce. Důležité je zmínit, že Razor Pages používají metodu *Page()* pro vyrenderování Razor View, nikoliv metodu *View()* jako MVC. (45)

3.3.5. Ostatní

ASP.NET Core umí též poskytovat SignalR a gRPC služby. Také je možné použít ASP.NET Core Blazor k tvorbě klientských webových UI.

3.3.5.1. SignalR

SignalR je protokol umožňující komunikaci v reálném čase (real-time) a asynchronní komunikaci za pomocí zasílání zpráv mezi serverem a klientem. Tyto zprávy mohou být formátovány buď pomocí JSON, nebo MessagePack serializace. Možným využitím jsou real-time prvky na webech

(chat, sledování živých dat), nebo i real-time komunikace mezi serverem a mobilní či desktop aplikací. (46)

3.3.5.2. gRPC

gRPC je framework pro vzdálené volání procedur (remote procedure call, RPC) vytvořená společností Google. Využívá možností poskytnutých protokolem HTTP/2 pro poskytnutí rychlé, obousměrné komunikace. (47) (48) Většina využití gRPC je viděna v komunikaci mezi server nebo servery a newebovými aplikacemi, z důvodu omezené možnosti využití HTTP/2 s klientských webových aplikacích psaných v Javascriptu. (49)

3.3.5.3. Blazor

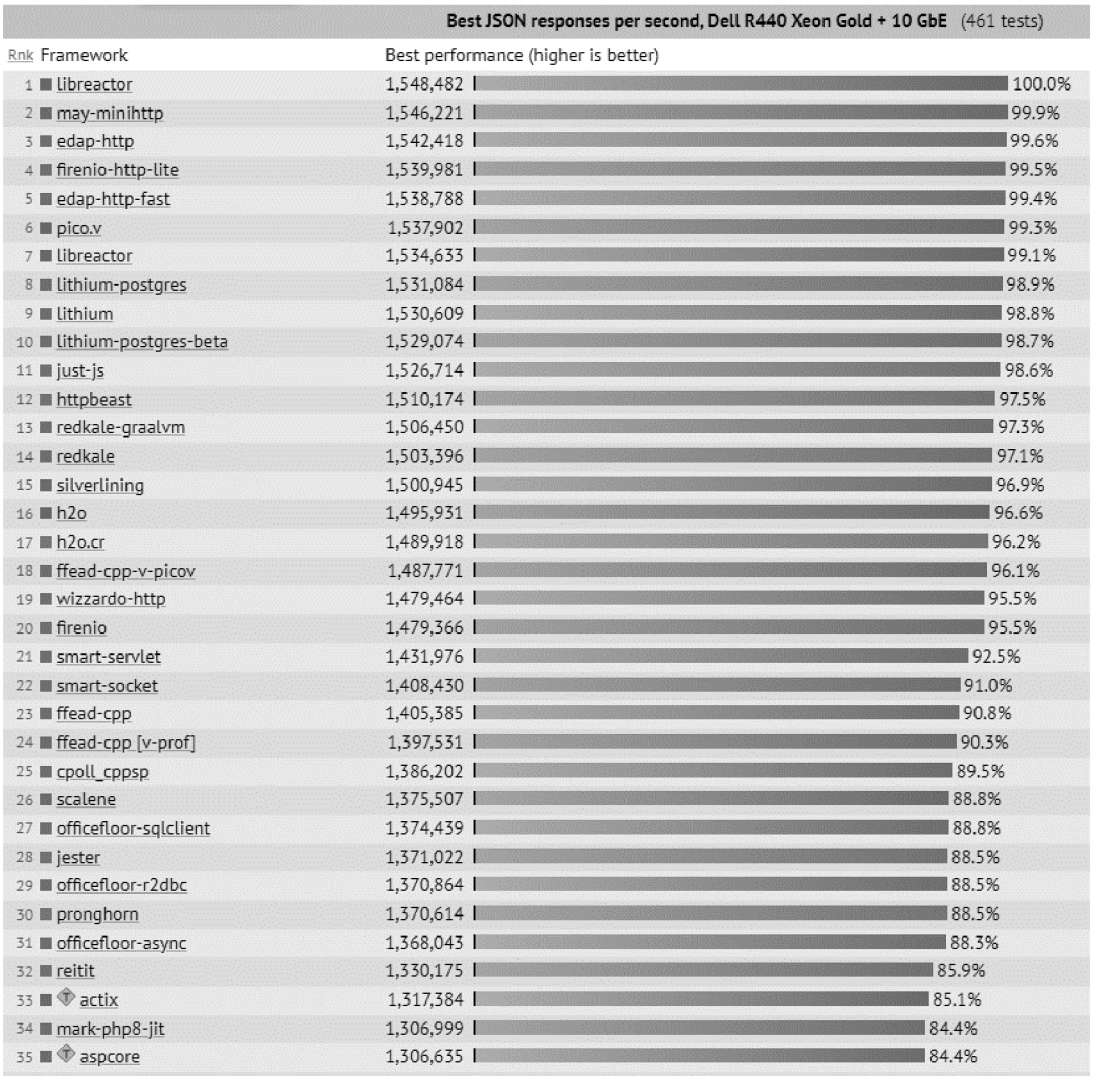
ASP.NET Core Blazor je framework pro tvorbu interaktivních webových UI pro webové klienty. Využívá buď technologie WASM (WebAssembly), standard definující binární formát souborů spustitelných ve webových prohlížečích, umožňující mimo jiné spuštění .NET programu v prohlížeči, nebo SignalR spojení k serveru, přes které je řešena interaktivita. Oba přístupy mají svoje výhody a nevýhody, které ovšem nespadají pod téma webových backendů. (50)

4. Alternativy

Při výběru webového frameworku není ASP.NET Core jediná možná volba. Pro některé aplikace nemusí být ani tou správnou. Proces výběru záleží na mnoha faktorech, které je nutno evaluovat před učiněním rozhodnutí.

Příkladem, kde ASP.NET Core není v tuto chvíli nejsilnější volba, je SSR (Server Side Rendering), tj. renderování komponentů webového UI, které by za běžných okolností byly vyrenderovány až po načtení scriptů ve webovém prohlížeči klienta, na straně serveru. (51) SSR bylo historicky nejlépe integrováno do JavaScriptových webových frameworků z důvodu použití stejného programovacího jazyku na straně serveru i klienta a to jazyku JavaScript, což umožňuje použití toho samého kódu k vykreslení komponentů webového UI jak na straně serveru, tak na straně klienta. Toto ale přestává být pravdou s nástupem frameworků využívajících technologii WebAssemby, díky které už není JavaScript jediným programovacím jazykem použitelným ve webovém prohlížeči. Součástí ASP.NET Core je sice technologie Blazor, zmíněná v kapitole 3.3.5.3, která ve verzi .NET 8 preview 3 získala první známky podpory SSR. Jedná se zatím pouze o omezenou preview verzi, která rozhodně není připravena na použití v produkčním prostředí. (52) Pokud dnes chceme framework s plnou a otestovanou podporou SSR připravený na použití v produkčním prostředí, sáhneme po některém z mnoha JavaScriptových frameworků, jako např. NextJS (53) nebo SvelteKit (54).

Dalším možným důvodem, proč se vyhnout ASP.NET Core je požadavek extrémního výkonu. ASP.NET Core samotný není pomalý (viz. Obrázek 6 Benchmark výkonu ASP.NET Core), ovšem můžeme narazit na situace, kdy je jeho výkon nedostačující. V těchto situacích můžeme využít některý z frameworků pro jazyky kompilované do nativního kódu, jako třeba Actix (55) nebo Axum (56) pro programovací jazyk Rust. Další výhodou Rustu je alternativní model správy paměti, který nevyužívá Garbage Collectoru, což eliminuje pauzy v provádění programu, které nevyhnutelně GC způsobuje. Tato výhoda je ovšem vyvážena složitějším psaním programu. (57)



Best JSON responses per second, Dell R440 Xeon Gold + 10 GbE (461 tests)

Rnk	Framework	Best performance (higher is better)
1	libreactor	1,548,482 — 100.0%
2	may-minihttp	1,546,221 — 99.9%
3	edap-http	1,542,418 — 99.6%
4	firenio-http-lite	1,539,981 — 99.5%
5	edap-http-fast	1,538,788 — 99.4%
6	pico.v	1,537,902 — 99.3%
7	libreactor	1,534,633 — 99.1%
8	lithium-postgres	1,531,084 — 98.9%
9	lithium	1,530,609 — 98.8%
10	lithium-postgres-beta	1,529,074 — 98.7%
11	just-js	1,526,714 — 98.6%
12	httpbeast	1,510,174 — 97.5%
13	redkale-graalvm	1,506,450 — 97.3%
14	redkale	1,503,396 — 97.1%
15	silverlining	1,500,945 — 96.9%
16	h2o	1,495,931 — 96.6%
17	h2o.cr	1,489,918 — 96.2%
18	ffead-cpp-v-picov	1,487,771 — 96.1%
19	wizzardo-http	1,479,464 — 95.5%
20	firenio	1,479,366 — 95.5%
21	smart-servlet	1,431,976 — 92.5%
22	smart-socket	1,408,430 — 91.0%
23	ffead-cpp	1,405,385 — 90.8%
24	ffead-cpp [v-prof]	1,397,531 — 90.3%
25	cpoll_cppsp	1,386,202 — 89.5%
26	scalene	1,375,507 — 88.8%
27	officefloor-sqlclient	1,374,439 — 88.8%
28	jester	1,371,022 — 88.5%
29	officefloor-r2dbc	1,370,864 — 88.5%
30	pronghorn	1,370,614 — 88.5%
31	officefloor-async	1,368,043 — 88.3%
32	reitit	1,330,175 — 85.9%
33	actix	1,317,384 — 85.1%
34	mark-php8-jit	1,306,999 — 84.4%
35	aspcore	1,306,635 — 84.4%

Obrázek 6 Benchmark výkonu ASP.NET Core (55)

Může nás také omezovat prostředí, do kterého budeme hotový web nasazovat. Ne všechny webové stránky běží na velkých serverech. Např. web na konfiguraci routeru musí mít též backend, který bude aplikovat uživatelem vybrané nastavení. Nemusíme pro příklad zacházet ani do tak velkých extrémů – pronajmutí serveru, ať už fyzického nebo virtuálního, není zdarma. Pokud dokážeme omezit kolik výpočetních zdrojů web potřebuje, můžeme tím i ovlivnit kolik budeme platit za běh naší aplikace. Problém ceny je ještě více amplifikován v případě využití cloudových služeb od poskytovatelů jako Microsoft Azure, GCP (Google Cloud Platform) nebo AWS (Amazon Web Services), kde je běžné platit za sekundy strávené prováděním programu na procesoru.

Servery nejsou ovšem jediná věc, za kterou musíme platit. Vývojový čas a nutná úroveň zkušeností developerů jsou důležitými faktory. Z tohoto hlediska je zajímavý programovací jazyk GO a webové frameworky postavené kolem něj. Programovací jazyk GO je úmyslně postaven tak, aby byl, co nejjednodušší. Zároveň je kompilován do nativního kódu, v důsledku čemuž je rychlý, ale stále disponuje garbage collectorem, díky kterému za cenu malého snížení výkonu nemusíme řešit manuální správu paměti. Po kompilaci produkuje pouze jeden soubor bez externích závislostí, což zjednodušuje nasazení a samotná rychlost kompilace je ve srovnání s jinými programovacími jazyky relativně vysoká. (59) Odlehčený webový framework má GO vystavěný přímo ve své standardní knihovně, nebo je možné použít některou z alternativ, např. GO Fiber (60) nebo GIN (61).

Ne všechny aplikace potřebují maximální výkon, efektivitu nebo speciální možnosti. V případech, kdy potřebujeme jen rychle udělat funkční API pro pár klientů nebo děláme malý osobní projekt, můžeme využít třeba webový framework Flask pro programovacý jazyk Python (62). Výhodou je jednoduchost a nepřítomnost kompilačního kroku, za což ovšem zaplatíme výkonem.

Pro některé druhy aplikací můžeme použít některý z frameworků pro programovací jazyk PHP. PHP je zde již velmi dlouhou dobu a bylo jedním z prvních jazyků pro psaní logiky na straně serveru (63). Velkou výhodou je možnost hostingu téměř kdekoliv, a to často i u poskytovatelů webového hostingu, kteří běžně pod názvem “Web Hosting“ podporují statické stránky a PHP s MySQL databází (64). Takováto služba je často dostupná za velice nízké částky a někdy i zadarmo v určité omezené kapacitě. PHP lze používat i bez webového frameworku, je ale vysoce doporučeno nějaký použít, abychom se vyhnuli tvoření bezpečnostních děr, což je v čistém PHP velice jednoduché. Nejpopulárnějším webovým frameworkem pro PHP je Laravel (65).

5. Závěr

Webové frameworky pro vývoj backendů jsou nezbytnou součástí moderního webového vývoje. Poskytují počáteční bod pro rapidní vývoj backendů všech možných druhů a velikostí. ASP.NET Core a C# jsou však pouze jedny z mnoha frameworků a programovacích jazyků, a i přestože řeší mnoho, ne-li ohromující většinu, aspektů vývoje webových backendů, stále nejsou ultimátním řešením všech problémů. Je důležité nenásledovat slepě jeden framework a prozkoumat, co nabízejí ostatní a vybrat ten správný pro daný projekt.

Vývoj jde stále kupředu. Nové frameworky se objevují a existující frameworky dostávají nové schopnosti a výkonnostní zlepšení. Zajímavé je, že specificky pro .NET platformu žádné další větší webové frameworky mimo ASP.NET Core neexistují, pokud teda nepočítáme ASP.NET, který je jeho předchůdcem.

6. Zdroje

1. **frameworks, Server-side web. *MDN Web Docs.* [Online] [Citace: 25. 7 2023.] https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.**

2. HTTP requests. *IBM Documentation.* [Online] [Citace: 25. 7 2023.] https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-requests.

3. Stack Overflow Developer Survey 2023. *Stack Overflow.* [Online] [Citace: 27. 7 2023.] https://survey.stackoverflow.co/2023/#section-most-popular-technologies-web-frameworks-and-technologies.

4. Introducing .NET 5. [Online] [Citace: 30. 7 2021.] https://devblogs.microsoft.com/dotnet/introducing-net-5/.

5. Visual Basic support planned for .NET 5.0. [Online] [Citace: 3. 7 2021.] https://devblogs.microsoft.com/vbteam/visual-basic-support-planned-for-net-5-0/.

6. WineHQ. [Online] [Citace: 1. 8 2021.] https://www.winehq.org/.

7. Mono. [Online] [Citace: 1. 8 2021.] https://www.mono-project.com/.

8. .NET is free. [Online] [Citace: 8. 8 2021.] https://dotnet.microsoft.com/platform/free.

9. .NET Versions. [Online] [Citace: 8. 8 2021.] https://dotnet.microsoft.com/download/dotnet.

10. .NET Standard. [Online] [Citace: 8. 8 2021.] 8. https://docs.microsoft.com/en-us/dotnet/standard/net-standard.

11. .NET Documentation: Compile into .NET Framework CIL. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/cs-cz/dynamicsax-2012/appuser-itpro/compile-into-net-framework-cil.

12. .NET Documentation: Common Language Runtime (CLR) overview. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/standard/clr.

13. .NET Documentation: .NET application publishing overview. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/core/deploying/.

14. .NET Documentation: Single file deployment and executable. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/core/deploying/single-file.

15. .NET Documentation: Automatic Memory Management. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/standard/automatic-memory-management.

16. .NET Documentation: Managed Execution Process. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process.

17. .NET Documentation: ReadyToRun Compilation. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/core/deploying/ready-to-run.

18. What is Dotnet Framework. [Online] [Citace: 8. 8 2021.] https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework.

19. Architectural principles. *Microsoft Documentation.* [Online] [Citace: 31. 1 2023.] https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion.

20. Dependency injection in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 31. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0.

21. ASP.NET Core Middleware. *Microsoft Documentation.* [Online] [Citace: 8. 8 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-7.0.

22. App startup in ASP.NET Core 5. *Microsoft Documentation.* [Online] [Citace: 8. 8 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-5.0.

23. App startup in ASP.NET Core 7.0. *Microsoft Documentation.* [Online] [Citace: 29. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-7.0.

24. Logging in .NET Core and ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 19. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-7.0.

25. *Datalust SEQ.* [Online] 20. 1 2023. https://datalust.co/seq.

26. Configuration in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 18. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-7.0.

27. Use HTTP/3 with the ASP.NET Core Kestrel web server. *Microsoft Documentation.* [Online] [Citace: 20. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/http3?view=aspnetcore-7.0.

28. Host ASP.NET Core on Windows with IIS. *Microsoft Documentation.* [Online] [Citace: 20. 12 2022.] https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/?view=aspnetcore-7.0.

29. In-process hosting with IIS and ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 20. 12 2022.] https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/in-process-hosting?view=aspnetcore-7.0.

30. Out-of-process hosting with IIS and ASP.NET Core. *Microsoft Documenation.* [Online] [Citace: 20. 12 2022.] https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/out-of-process-hosting?view=aspnetcore-7.0.

31. HTTP.sys web server implementation in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 8. 8 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/httpsys?view=aspnetcore-7.0.

32. Model Binding in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 8. 8 2023.] https://learn.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-7.0.

33. Views in ASP.NET Core MVC. *Microsoft Documentation.* [Online] [Citace: 30. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-7.0.

34. Partial Pages. *Learn Razor Pages.* [Online] [Citace: 30. 1 2023.] https://www.learnrazorpages.com/razor-pages/partial-pages.

35. Layout Pages. *Learn Razor Pages.* [Online] [Citace: 30. 1 2023.] https://www.learnrazorpages.com/razor-pages/files/layout.

36. The ViewImports File. *Learn Razor Pages.* [Online] [Citace: 30. 1 2023.] https://www.learnrazorpages.com/razor-pages/files/viewimports.

37. The ViewStart file. *Learn Razor Pages.* [Online] [Citace: 31. 1 2023.] https://www.learnrazorpages.com/razor-pages/files/viewstart.

38. Tag Helpers in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 2. 7 2023.] https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-7.0.

39. Author Tag Helpers in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 8. 8 2023.] https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring?view=aspnetcore-7.0.

40. Minimal APIs Quick Reference. *Microsoft Documentation.* [Online] [Citace: 10. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-6.0.

41. Overview of ASP.NET Core MVC. *Microsot Documentation.* [Online] [Citace: 4. 7 2023.] https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-7.0.

42. Routing to controller actions in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 8. 8 2023.] https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-7.0.

43. Create web APIs with ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 15. 7 2023.] ttps://learn.microsoft.com/en-us/aspnet/core/web-api.

44. RFC 7807. [Online] [Citace: 20. 7 2023.] https://datatracker.ietf.org/doc/html/rfc7807.

45. Introduction to Razor Pages in ASP.NET Core. *Microsoft Documentation.* [Online] [Citace: 22. 7 2023.] https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-7.0&tabs=visual-studio.

46. Introduction to SignalR. *Microsoft Documentation.* [Online] [Citace: 25. 6 2023.] https://learn.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr.

47. Overview for gRPC on .NET. ***Microsoft Documentation.*** **[Online] [Citace: 25. 6 2023.] https://learn.microsoft.com/en-us/aspnet/core/grpc/?view=aspnetcore-7.0.**

48. gRPC. ***gRPC.*** **[Online] [Citace: 25. 6 2023.] https://grpc.io/.**

49. gRPC Web. ***Github.*** **[Online] [Citace: 25. 6 2023.] https://github.com/grpc/grpc-web.**

50. ASP.NET Core Blazor. ***Microsoft Documentation.*** **[Online] [Citace: 27. 7 2023.] https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-7.0.**

51. Server-Side Rendering (SSR). ***Vue.JS.*** **[Online] [Citace: 27. 7 2023.] https://vuejs.org/guide/scaling-up/ssr.html#what-is-ssr.**

52. ASP.NET Core updates in .NET 8 Preview 3. ***Microsoft .NET Blog.*** **[Online] [Citace: 27. 7 2023.] https://devblogs.microsoft.com/dotnet/asp-net-core-updates-in-dotnet-8-preview-3/.**

53. ***NextJS.*** **[Online] [Citace: 27. 7 2023.] https://nextjs.org/.**

54. ***SvelteKit.*** **[Online] [Citace: 8. 8 2023.] https://kit.svelte.dev/.**

55. ***Actix.*** **[Online] [Citace: 27. 7 2023.] https://actix.rs/.**

56. ***Axum.*** **[Online] [Citace: 27. 3 2023.] https://github.com/tokio-rs/axum.**

57. What Is Ownership? ***Rust Docs.*** **[Online] [Citace: 8. 8 2023.] https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.**

58. Web Framework Benchmarks. ***TechEmpower.*** **[Online] [Citace: 27. 7 2023.] https://www.techempower.com/benchmarks/#section=data-r21&hw=ph&test=json.**

59. ***GO.*** **[Online] [Citace: 27. 7 2023.] https://go.dev/.**

60. ***GO Fiber.*** **[Online] [Citace: 27. 7 2023.] https://gofiber.io/.**

61. ***Gin Web Framework.*** **[Online] [Citace: 27. 7 2023.] https://gin-gonic.com/.**

62. ***Flask.*** **[Online] [Citace: 8. 8 2023.] https://flask.palletsprojects.com/en/2.3.x/.**

63. History of PHP. ***PHP.*** **[Online] [Citace: 8. 8 2023.] https://www.php.net/manual/en/history.php.**

64. webhosting. *forps.* [Online] [Citace: 8. 8 2023.] https://www.forpsi.com/webhosting/.

65. *Laravel.* [Online] [Citace: 8. 8 2023.] https://laravel.com/.

66. App startup in ASP.NET Core 5.0. *Microsoft Documentation.* [Online] [Citace: 29. 1 2023.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-5.0.

67. .NET Team. Introducing .NET 5. [Online] [Citace: 3. 7 2021.] https://devblogs.microsoft.com/dotnet/introducing-net-5/.

68. —. Visual Basic support planned for .NET 5.0. [Online] [Citace: 3. 7 2021.] https://devblogs.microsoft.com/vbteam/visual-basic-support-planned-for-net-5-0/.

69. WineHQ. [Online] WineHQ. [Citace: 1. 8 2021.] https://www.winehq.org/.

70. Mono. [Online] Mono Project. [Citace: 1. 8 2021.] https://www.mono-project.com/.

71. Microsoft. .NET is free. [Online] [Citace: 8. 8 2021.] https://dotnet.microsoft.com/platform/free.

72. —. .NET Versions. [Online] [Citace: 8. 8 2021.] https://dotnet.microsoft.com/download/dotnet.

73. —. .NET Standard. [Online] [Citace: 8. 8 2021.] https://docs.microsoft.com/en-us/dotnet/standard/net-standard.

74. —. .NET Documentation: Compile into .NET Framework CIL. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/cs-cz/dynamicsax-2012/appuser-itpro/compile-into-net-framework-cil.

75. —. .NET Documentation: Common Language Runtime (CLR) overview. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/standard/clr.

76. —. .NET Documentation: .NET application publishing overview. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/core/deploying/.

77. —. .NET Documentation: Single file deployment and executable. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/core/deploying/single-file.

78. —. .NET Documentation: Automatic Memory Management. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/standard/automatic-memory-management.

79. —. .NET Documentation: Managed Execution Process. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process.

80. —. .NET Documentation: ReadyToRun Compilation. [Online] [Citace: 30. 7 2021.] https://docs.microsoft.com/en-us/dotnet/core/deploying/ready-to-run.

81. What is HTTP/3? *Cloudflare.* [Online] [Citace: 27. 7 2023.] https://www.cloudflare.com/en-ca/learning/performance/what-is-http3/.

82. Target frameworks in SDK-style projects. *Microsoft Documentation.* [Online] [Citace: 7. 8 2023.] https://learn.microsoft.com/en-us/dotnet/standard/frameworks.

7. Seznam použitích symbolů a zkratek

HTML	Hyper Text Markup Language
DI	Dependency Injection
IoC	Inversion of Control
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
MVC	Model View Controller
API	Application Programming Interface
IIS	Internet Information Services
OS	Operating System
XML	eXtensible Markup Language
AuthN	Authentication
AuthZ	Authorization
JS	JavaScript
SSR	Server Side Rendering
WASM	WebAssembly

8. Obrázky

9. Zadání

Zadání bakalářské práce

Autor:	**Tomáš Krámský**
Studium:	I1900210
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Webový Framework ASP.NET Core**
Název bakalářské práce AJ:	The ASP.NET Core Web Framework

Cíl, metody, literatura, předpoklady:

Cíl: Práce vyzdvihne důležitost frameworků pro vývoj backendových webových aplikací a popíše podrobněji jeden z nich - ASP.NET Core. Zmíní také některé alternativní backendové webové frameworky.

Osnova:

1. Úvod
2. Webové frameworky - obecný přehled
3. Popis ASP.NET Core
4. Alternativy k ASP.NET Core
5. Výsledky a závěr

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Oponent:	prof. RNDr. PhDr. Antonín Slabý, CSc.
Datum zadání závěrečné práce:	26.1.2021