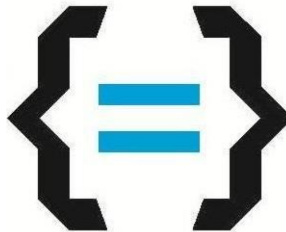
FAKULTA INFORMATIKY A MANAGEMENTU

UNIVERZITA HRADEC KRÁLOVÉ

Katedra informatiky a kvantitativních metod



Vývoj moderních webových aplikací v ASP.NET Core

Diplomová práce

Autor: Bc. Lukáš Bareš

Obor: Aplikovaná Informatika – kombinovaná forma

Vedoucí práce: Ing. Jiří Štěpánek

Hradec Králové 17.4.2018

Prohlášení:

Prohlašuji, že jsem tuto práci vytvořil samostatně s použitím uvedených zdrojů a literatury.

V Hradci Králové dne: Bc. Lukáš Bareš

………………………………… …………………………………..

Poděkování:

Velice děkuji svému vedoucímu diplomové práce Ing. Jiřímu Štěpánkovi za velmi inspirativní a zajímavé téma práce. Bylo mi oporou jeho metodické vedení a vlídný osobní přístup. Můj obrovský dík patří v neposlední řadě mé rodině a kolegům, za jejich podporu, trpělivost a motivaci během tvorby této diplomové práce.

Anotace práce:

Tato diplomová práce se zabývá tématikou vývoje moderních webových aplikací z pohledu moderních architektur a za pomoci pokročilého multiplatformního webového frameworku ASP.NET Core. Zkoumá rozdíly mezi monolitickými aplikacemi a systémy založenými na architektuře mikroslužeb. Vyhodnocuje klady a zápory každé z těchto uvedených architektur. Zabývá se vnitřní stavbou frameworku ASP.NET Core, jeho konfigurací, bezpečností, flexibilitou, kompatibilitou s vývojovými nástroji a porovnáním rozdílu oproti ostatním starším verzím ASP.NET. Tato práce dále řeší použitelnost frameworku ASP.NET Core v moderních webových architekturách, speciálně pak v architektuře mikroslužeb.

Annotation:

This Diploma Thesis deals with the development of modern web applications from the point of view of modern architectures and with the help of the advanced multiplatform web framework ASP.NET Core. It examines the differences between monolithic applications and systems based on microservices architecture. It evaluates the pros and cons of each of these architectures. It deals with the ASP.NET Core framework structure, its configuration, security, flexibility, compatibility with development tools and comparing the difference with other older versions of ASP.NET. This thesis further solves the usability of the ASP.NET Core framework in modern web architectures, especially in microservices architecture.

Obsah

1 Úvod 1
2 Moderní webové aplikace 4
2.1 Monolitické aplikace 6
2.2 Architektura mikroslužeb 8
2.2.1 Návrh architektury mikroslužeb 12
2.2.2 Migrace z monolitické aplikace na aplikaci mikroslužeb 17
3 Moderní podoba .NET platformy 20
3.1 Webové frameworky ASP.NET 24
4 ASP.NET Core 26
4.1 Architektura aplikace 29
4.2 Princip komunikace a práce s protokolem 31
4.2.1 Middlewares a filtry 33
4.2.2 Application Services 35
4.3 Konfigurace aplikace 37
4.3.1 Směrování požadavků v aplikaci 39
4.4 Práce s daty a jejich skladování 40
4.4.1 Entity Framework 41
5 Zabezpečení ASP.NET Core REST aplikací 44
5.1 Zabezpečení komunikace REST modulů 47
5.2 ASP.NET Core - DataProtector 49
5.3 Uložení citlivých informací v aplikaci 50
5.4 Autentizační a autorizační procesy 51
5.5 Práce s Cookies 55
5.6 ASP.NET Identity 55
6 Spojení ASP.NET Core s architekturou mikroslužeb 57

6.1 Virzualizační nástroj Docker ... 61
7 Praktická část práce – ukázková aplikace ... 62
7.1 Serverový cluster ... 63
7.1.1 Autentizace a autorizace ... 65
7.1.2 Analýza webových stránek ... 66
7.2 Klientský webový modul ... 68
7.3 Možnosti dalšího rozvoje vytvořené aplikace ... 69
8 Závěry a doporučení ... 70
9 Citované zdroje a literatura ... 73

1 Úvod

V rámci vývoje trendů moderní informatické společnosti je stále rozsáhlejší část lidského života přenášena do virtuálního prostoru. Stále více rutinních profesních činností lze dnes usnadnit, vylepšit nebo úplně nahradit digitálními platformami sdíleného virtuálního prostoru. Například objednat nákup potravin, zajistit vánoční dárky, rezervovat dovolenou, ubytování, letenky, spravovat finance, investovat, založit firmu, udržovat kontakt s klienty, propagovat nový produkt, analyzovat chování trhu nebo hodnotit efektivitu nabízené služby. Lze si jenom těžko představit, že by extrémně rychlý a intenzivní vývoj informačních technologií mohl být efektivní bez pokročilých technologií webového světa.

Základem zmíněného vzájemně propojeného virtuálního prostředí jsou inteligentní a flexibilní internetové platformy. Ty se díky vysokému zájmu odborné veřejnosti a komerčních společností neustále vyvíjejí a úzce spolupracují s ostatními moderními informačními technologiemi. Vzniklé spojení webových služeb, mobilních aplikací, internetu věcí, umělé inteligence nebo datové analytiky postupně formuje právě onen zmíněný „virtuální prostor". Tento svět se stává základem pro moderní průmyslová odvětví a pokročilé komplexní uživatelské služby. Mizí hranice mezi jednotlivými platformami a použitými technologiemi. Naopak stoupá snaha o vytvoření kooperujících služeb napříč informačními technologiemi ve spojení s aktuálními klientskými požadavky a trendy.

Moderní webové aplikace mají za úkol pracovat s uživatelskými daty, poutavě prezentovat výstupy systému koncovému uživateli a bezpečným způsobem data dlouhodobě archivovat. Neuspořádaná data je nutné kontinuálně statisticky analyzovat, hledat v nich neočekávané souvislosti, relace a vzory. Ty mohou vylepšovat vztahy mezi poskytovateli služeb a konzumenty.

Zároveň si moderní doba žádá po webových aplikacích vysokou míru odolnosti proti zátěži, rychlý a levný vývoj a provoz, co nejvyšší kompatibilitu s ostatními technologiemi, tvárnost pro dosažení cílů společnosti a implementaci aktuálních standardů. Současné komerční společnosti jsou na inteligentních informačních systémech závislé více, než kdykoli dříve.

Moderní informační platformy umožňují:

- Nové možnosti podnikání.
- Vytváření komunikačních kanálů s okolním světem.
- Nabídku nových produktů a služeb.
- Vyhodnocení efektivity činnosti soukromých společností.

Důkladnou analýzou dat lze identifikovat mezery na trhu, které vedou k potenciálnímu zisku. Tyto příležitosti by bez komplexní informační analýzy nebyly vůbec patrné.

Takřka každý moderní programovací jazyk nabízí minimálně jeden svůj vlastní webový framework. Zde je pouze několik základních příkladů:

- Python – Django.
- Java – Spring MVC.
- .NET (C#) – ASP.NET.
- PHP – Nette Framework.
- Javascript - Node.JS.
- Ruby – Ruby on Rails.

Všechny tyto moderní webové frameworky nabízí ucelený technologický základ pro vývoj pokročilých webových služeb a aplikací. Produkční společnosti požadují co nejrychlejší vývojový proces, přehlednost aplikace, flexibilitu nasazení, bohaté zázemí podpůrných nástrojů, kvalitní prostředí pro publikaci aplikace, dostatečné zázemí lidských zdrojů, podporu testovacích nástrojů, efektivitu řešení krizových situací a v neposlední řadě bezpečnost vytvořené služby.

Jedním z uvedených frameworků je ASP.NET, který umožňuje vývoj webových aplikací postavených na platformě .NET firmy Microsoft Corporation. Je značně oblíbený u komerčních vývojových společností, protože umožňuje rychlý, normovaný a automatizovaný vývoj webových aplikací. Jeho nejmodernější verze s označením ASP.NET Core je v porovnání s předchozími verzemi značně inovativní. Tato verze frameworku je určitý „restart“ dané technologie a vstup do nové generace webových aplikací vytvořených pro platformu .NET.

Krom samotných frameworků a programovacích jazyků je nutné webovým aplikacím přizpůsobit také samotnou architekturu softwarových řešení. V dřívějších dobách byla

webová aplikace chápána spíše jako kolekce stránek obsahující sémanticky řazený obsah. Tyto aplikace se většinou nezabývaly dynamikou nebo složitější aplikační logikou. Z tohoto důvodu byly starší webové frameworky orientované spíše na stránkování a prezentaci textového obsahu, než na aktivní funkcionalitu. Moderní internetový svět se však rychle transformuje a přizpůsobuje aktuálním trendům, například rostoucímu významu mobilních technologií nebo složitějším požadavkům na správu dat a jejich porozumění.

V důsledku těchto trendů musí webové informační systémy řešit mnoho problémů:

- Významné množství klientů používající mobilní zařízení.
- Dynamická prezentace dat.
- Pohodlné ovládání a spolehlivost na různorodých zařízeních.
- Kontinuální analýza aplikace a chování aktivních uživatelů.
- Požadavky klientů na pravidelné inovace aplikace.
- Vyšší nároky na management rizik a bezpečnost.

Klasické nástroje a architektury webového prostředí na tyto požadavky už nestačí.

Velkým krokem vpřed byl rozkvět dynamických prezentačních webových technologií, které fungují na straně klienta webové aplikace. Značný pokrok proběhl v úrovni internetových prohlížečů, které konečně začaly respektovat moderní trendy, a umožnily efektivní implementaci pokročilé aplikační logiky aplikací pro prezentaci dat. Toto vše se stalo plodným základem evoluce webových aplikací, ale často neexistoval způsob jak tyto moderní přístupy implementovat do konvenčních monolitických architektur.

Objevil se volný prostor pro nové přístupy k řešení problémů a pokrytí poptávky ze strany poskytovatelů, i koncových konzumentů. Staré konvenční architektury, a na nich postavené webové frameworky, s těmito požadavky příliš nepočítají a moderní doba velí posunout se dál.

2 Moderní webové aplikace

Internet se v moderní době stal vysoce žádaným prostředím ve všech odvětvích, která jsou alespoň okrajově napojena na informační technologie. Banky, pojišťovny, obchody, mobilní operátoři, automobilový producenti, pohostinství, dopravní a cestovní společnosti jsou jenom špičkou ledovce zájemců poskytovat své služby klientům skrze webová rozhraní kdekoli a kdykoli to bude jenom možné [1].

Také díky moderním technologickým platformám vznikají nové druhy služeb, které mají tendenci velice rychle expandovat a získávat velké množství uživatelů. S rostoucím počtem nových služeb roste i objem dat, se kterými je nutné pravidelně pracovat. Příkladem mohou být služby sdílené ekonomiky, například Uber, Car4way nebo Airbnb. Samostatným tématem jsou expandující sociální platformy, skrze které lidé sdílí stále větší množství informací. To umožňuje společnostem lepší cílení reklamy, její aktivnější a efektivnější distribuci.

Z hlediska moderní architektury existují dvě hlavní cesty vývoje webové aplikace [2]:

- Architektura monolitické aplikace.
- Architektura principu mikroslužeb.

Obě možnosti vývoje jsou stále široce uplatňovány jak u stávajících, tak nově vznikajících projektů. Rozhodování o použití konkrétní architektury by mělo být součástí úvodní analýzy požadavků společnosti a jejich koncových klientů. Je nutné předvídat stav a požadavky systému v budoucnosti.

Základní otázky, které je vhodné si ještě před samotnou realizací projektu položit:

1. Expandování aplikace:
    a. Jaká bude velikost aplikace a kolik bude mít logických služeb?
    b. Jaký je očekávaný růst aktivních uživatelů aplikace?
    c. Bude se aplikace v budoucnu rozšiřovat o další nové služby?
    d. Jaká bude frekvence úprav aplikace?
    e. Bude aplikace reagovat na moderní technologické trendy?
2. Nároky na vývojový tým:
    a. Jaké budou odborné požadavky na vývojáře?
    b. Jaká bude velikost týmu během aktivního vývoje?
    c. Bude se v průběhu vývoje měnit odborného složení vývojářů?
3. Zpracování a archivace dat:
    a. Kolik dat bude aplikace aktivně zpracovávat?
    b. Lze očekávat růst požadavků na zpracování dat?
    c. Lze očekávat rozdělení dat do více databází?
    d. Lze očekávat migrace dat?
    e. Bude aplikace využívat data z externích zdrojů?
4. Časové nároky:
    a. Jaký bude časový harmonogram vývoje aplikace?
    b. Jaké budou milníky během vývoje aplikace?
    c. Bude aplikace jednorázovým řešením, nebo lze předpokládat kontinuální dlouhodobý vývoj?

Odpovědi na tyto otázky mohou zásadně usnadnit rozhodování o použití vhodné webové architektury. Špatné rozvržení může v budoucnu vést k nutnosti migrace z jedné architektury na druhou. To je finančně, časově i pracovně náročná operace. Předem může být velice těžké odhadnout potenciál expanze dané webové aplikace. Je proto nezbytné nepodceňovat ověření veřejného zájmu o business model dané aplikace. V moderní době může u menších společností pomoci například crowdfunding[1] – výsledek kampaně

[1] Poskytnutí finančních zdrojů pro vývoj aplikace ještě před uvedením na trh uživateli skrze veřejnou platformu.

reflektuje zájem ze strany potenciálních klientů. Větší společnosti mohou pak použít profesionální průzkum trhu.

2.1 Monolitické aplikace

Jedná se o komplexní aplikace zastávající role klienta i správce zpracování dat. Aplikace se navrhují, vyvíjí a kompilují jako jednolité softwarové řešení s jasně definovanými zdroji výkonu a paměti. Často jsou rozšířeny o externí knihovny nebo zdroje dat. Mohou obsahovat jednoduchá API rozhraní (Application Programming Interface) jako bránu pro komunikaci s okolním světem mimo standardní přístup skrze webový prohlížeč. Praktická použitelnost je však omezena [2].

Z hlediska vývoje vyžadují monolitické aplikace nižší časové a finanční nároky pro krátkodobý vývoj, protože vše se v rámci vývojového procesu řeší současně. Struktura aplikace je navržena v rámci komplexního frameworku, který stačí pouze vhodně rozšiřovat o aplikační logiku a datové struktury specifické aplikace. Monolitické aplikace si během vývoje vystačí s menším týmem vývojářů, protože rozmanitost technologií je omezena na daný framework a jeho reálné možnosti. Není tedy nutné vytvářet nebo implementovat složitá unifikovaná řešení pro každou konkrétní aplikaci. Vše lze navrhovat normalizovaně a tím pádem také poměrně rychle.

Výhodou této architektury je také jednodušší a komplexnější implementace zabezpečení, kdy jednotná bezpečnostní politika pokrývá celý systém. Existuje jen relativně malá konečná množina potenciálních hrozeb, která se nerozšiřuje o chyby při vytváření členitého neprověřeného unifikovaného řešení dané webové aplikace. Toto platí za předpokladu, že bezpečnostní politika daného monolitického frameworku je správně navržena a implementována.

Zaškolení nových vývojářů do již existujícího projektu bývá u monolitických aplikací jednodušší, jelikož jsou založeny na obecně známé architektuře. Noví vývojáři se v projektu rychle orientují a nemají problém se začleněním do aktivního vývoje webové aplikace. Je nutné však zajistit větší množství pracovníků s jednou konkrétní specializací, což nemusí být na současném trhu práce snadné.

Otestování a publikace konkrétní verze monolitické aplikace bývá obecně závislé na jejím obsahu. V rámci testování celé komplexní aplikace lze použít normalizovaná testovací řešení. Nevýhodou je potřeba tvorby testů pro celé komplexní řešení a relativně obtížné

separování jeho dílčích částí (knihovny, repositáře). Tyto části jsou pevně svázány s daným monolitickým frameworkem. Zde hrají důležitou roli integrační testy, nebo důkladná simulace objektů uvnitř jednotkových testů. Každá změna ve zdrojovém kódu nebo samotném návrhu aplikace ovlivní systém jako celek. Vzniká tak vysoké riziko skryté chyby, což logicky znamená náročné požadavky na testování před nasazením produkční verze webové aplikace.

K publikaci aplikace lze použít již předpřipravené a otestované služby cloudových platforem. Tyto služby vyžadují pouze minimum konfigurace pro jedno specifické řešení monolitické aplikace. Případné úpravy již existujícího řešení webové aplikace lze relativně rychle nechat projít testovacím procesem a následnou publikací. Tento fakt umožňuje krátkou reakční dobu na opravu chyb. Každý incident ale může relativně zásadně ovlivnit celou aplikaci jako celek. Může dojít k fatálnímu selhání, přestože incident souvisí pouze s omezenou částí aplikace. Stabilita monolitického systému je tak v mnoha ohledech bipolární – buď systém funguje celý, nebo dojde k fatálnímu selhání. Stejně tak případné prolomení bezpečnostní politiky monolitického systému má mnohem zásadnější dopady. Monolitické aplikace často používají jednotnou společnou databázi pro veškerá data. Případný únik informací má fatální rozsah pro všechny uživatele.

Nevýhodou monolitické architektury je vysoká technologická svázanost. Proto jsou monolitické aplikace málo flexibilní co do kreativity konečného řešení. V rámci návrhu se nepočítá, že by se návrh řešení měl v budoucnu zásadním způsobem měnit nebo expandovat. Tento fakt bývá nejzásadnější problém monolitických aplikací, když začnou „přerůstat“ svůj prvotní účel. Od určité velikosti už nezvládají plnit požadovanou funkcionalitu rychlým a efektivním způsobem. Rozšiřováním aplikace vzniká příliš komplikovaný doménový model a rozsáhlé množství komplikovaných relací. Sestavování dotazů pro získání dat je až příliš výpočetně náročné, stejně jako jejich následná filtrace a dynamická prezentace. To u monolitické aplikace vede k prodlužování reakční doby na zpracování požadavků a nespokojenosti klientů při prezentaci dat.

V případě poklesu efektivity monolitické aplikace mnoho společností volí expanzi alokace hardwarových zdrojů pro výkon a paměť. To sice nabízí krátkodobé řešení problému, ale při dlouhodobě rostoucí tendenci zátěže aplikace ze strany pravidelných klientů je pouze otázka času, kdy opět začne monolitická architektura selhávat. Navíc řešení navyšováním prostředků pro chod špatně navržené aplikace je dlouhodobě velice nákladné. Druhou

volbou provizorního řešení problémů s výkonem monolitické aplikace je snaha o hledání dočasných krizových řešení ve zdrojovém kódu. Velice často však dochází ke komplikování vnitřní logiky nebo doménového modelu dané aplikace. O to více nákladný je následný pokus o migraci systému na pokročilejší druh architektury. Vývojářům zabere velké úsilí analýza monolitické aplikace s rozsáhlým počtem neefektivních úprav.

2.2 Architektura mikroslužeb

Jde o alternativní architektonický přístup návrhu webových aplikací, který předpokládá členitější, rozsáhlejší a komplikovanější strukturu konečného systému. Původně se s touto architekturou bylo možné setkat například u bankovních informačních systémů, protože mají velmi komplikovanou vnitřní stavbu s vysokými požadavky na spolehlivost, bezpečnost a stabilitu. Díky rostoucím požadavkům společností a jejich koncových klientů na moderní webové aplikace se tato architektura stává běžným řešením také u komerčních projektů. Využití tohoto architektonického principu umožňují moderní webové frameworky, které dříve k dispozici nebyly. Rovněž se zásadně snížila pracnost implementace tohoto řešení webové architektury, protože moderní sady pomocných nástrojů a knihoven jsou mnohem méně náročné na konfiguraci a umožňují snazší testování aplikace.

Základní myšlenkou architektury na principu mikroslužeb je fragmentace velkého systému na menší samostatné entity. Jednotlivé logické role uvnitř rozsáhlého systému se rozdělí mezi menší webové aplikace. Ty jsou v literatuře a webových zdrojích nazývané jako moduly, mikroslužby případně pouze služby. Tyto entity žijí vlastním životem, jsou plně samostatné a navzájem izolované. Zpracování požadavků uživatele vychází ze vzájemné spolupráce služeb na základě přidělených rolí a odpovědností v systému. Celkové řešení na uživatele působí jako jednolitá webová aplikace, přitom uživatelský požadavek může vyřizovat hned několik malých aplikací současně [2] [3].

Základní principy a požadavky architektury mikroslužeb:

- **Decentralizace** – celý systém je tvořen spoluprací samostatných prvků.
- **Nezávislost a izolace** – každá mikroslužba musí být co nejvíce nezávislá na ostatních službách v systému.
- **Odpovědnost** – každá služba musí být plně odpovědná za logickou funkcionalitu, kterou vykonává.
- **Nahraditelnost a flexibilita** – nová služba musí být schopna nahradit starou, aniž by změna zásadně ovlivnila aplikaci jako celek.
- **Spolupráce a komunikace** – služby musí mít definovaný protokol pro vzájemnou komunikaci a spolupráci.
- **Kontinuální vývoj** – vývoj systému lze realizovat jako dlouhodobý kontinuální proces.

Decentralizace systémů je základním aspektem architektury mikroslužeb. Pokud by systém měl silně centralizovaný prvek, přišel by o mnoho výhod architektury mikroslužeb. Naopak by na sebe převzal odpovědnost za rizika a omezení definovaná u monolitické architektury, například otázky stability a řešení kritických chyb. Příkladem může být služba v roli vstupní brány serveru s uživatelskými daty. Pokud by taková služba byla v návrhu pouze jedna, hrozilo by odříznutí uživatelských dat od klientských modulů, protože po jejím selhání by se požadavky nedostaly k ostatním službám v systému.

Vzájemná nezávislost a izolace služeb je prevencí případného dominového efektu při selhání určité mikroslužby. Tato chyba by nikdy neměla spustit řetězovou reakci ve stabilitě ostatních služeb uvnitř systému. Ten musí vždy zůstat funkční v nejvyšší možné míře. Pokud by například mikroslužba zajišťující logování událostí systému selhala, služba pro práci s daty osobního účtu uživatele musí stále plnit své úkoly. Oblasti systému, kde postižená data nejsou nezbytná, musí zůstat plně funkční.

V rámci architektury mikroslužeb musí panovat nejvyšší možná nezávislost na použitých technologiích, tedy použitých programovacích jazycích, frameworcích a nástrojích. Z venkovního pohledu musí být všechny moduly navzájem abstraktní a přístupné skrze normalizované rozhraní. Většinou jde o vzájemnou komunikaci služeb pomocí protokolu HTTP, ten však není jedinou možností. Volí se strukturovaný formát předávaných dat.

Musí být pro všechny mikroslužby snadno čitelný a zpracovatelný. V dnešní době jsou preferovány hlavně strukturované datové formáty JSON a XML [1] [4].

V praxi se společnosti většinou snaží o menší fragmentaci použitých technologií (programovací jazyky a frameworky), protože je nutné brát ohled na lidské zdroje a odborné profesní souznění členů vývojového týmu. Existují však případy, kdy například pro datovou analýzu nebo prvek umělé inteligence v aplikaci je určitý programovací jazyk lepší volbou, než jiný. Pak je architektura mikroslužeb určitě mnohem lepším řešením než monolitický systém, protože implementace nové technologie je díky vnitřní abstraktní vrstvě velmi snadná.

Princip odpovědnosti služby je implementací pravidla „Single Responsibility", tedy každá mikroslužba musí mít v návrhu architektury určenou právě jednu konkrétní odpovědnost. Tu by nikdy neměla překročit nebo ji předávat dál jiné službě, která není k plnění této funkce určena. V praxi například služba, která má odpovědnost nad logováním událostí v aplikaci, nesmí vůbec řešit formátování uživatelských dat a naopak. Je možné rozložit určitou rozsáhlou a komplexní úlohu do více služeb, avšak vždy musí být naprosto jasné, jaké úlohy konkrétní mikrolužba vykonává.

Nahraditelnost a postradatelnost logicky vyplívá z předchozích zásad. Pokud jsou mikroslužby navzájem abstraktní, nezávislé, izolované a samy za sebe plně odpovědné, mohou být rychle nahrazovány, aniž by to mělo zásadní dopad na systém jako celek.

Otázkou je také rozložení mikroslužeb v aplikaci. Závisí na business modelu a předpokládané zátěži systému. Je potřeba dosáhnout rovnoměrného rozložení zátěže na jednotlivé služby a tím i stability systému. Sledování zátěže je možné na základě vhodné politiky logování událostí systému a analýzy. Pokud se ukáže, že je určitá služba příliš vytížena a způsobuje snížení výkonu systému, je možné tuto službu rozdělit na dvě menší mikroslužby. Na základě logických rolí v systému se mezi tyto dvě nové služby rozdělí úkoly.

Správně navržené webové aplikace na principu mikroslužeb uplatňují princip „Continous Delivery", volně přeloženo kontinuálního vývoje pomocí průběžného dodávání nových služeb do aplikace. Na rozdíl od monolitické aplikace není nezbytně nutné pokrýt celou oblast funkcionality aplikace hned její první verzí, dokonce ani detailně předvídat budoucí

nároky systému. Postačí propracovaný základ architektury na základě důkladné business analýzy požadavků klienta [2].

Architektura mikroslužeb umožňuje flexibilně reagovat na nové moderní technologické trendy, požadavky klientů nebo přizpůsobení konečného řešení nejnovějšímu business modelu společnosti. Ten se logicky mění s tím, jak společnost expanduje. Úpravy probíhají bez nutnosti rozsáhlých změn celého systému a na nákladných finančních investic ze strany společnosti. Díky tomu nemusí společnost při plánování budoucí strategie brát ohled na podobu webové aplikace.

V rámci architektury mikroslužeb je možné jeden modul opravovat a testovat, ale ostatní nezávisle na něm fungují dál. Testovat jednu izolovanou mikroslužbu není problém. Jde o zjednodušenou webovou aplikaci s jasnou logickou rolí. Umožňuje snadné jednotkové i integrační testování. Problém nastává při testování systému jako celku. Identifikovat chybu s nejednoznačnou příčinou může být u rozsáhlého řešení složitý proces. Pravděpodobnost výskytu skryté chyby systému se výrazně sníží použitím důkladné testovací politiky. Pomůže například princip TDD (Test Driven Development). U něj platí, že se požadovaná funkcionalita nejprve vytvoří pomocí izolované testovací metody. Teprve po úspěšném otestování se metoda zanese do logiky produkční aplikace. Spouštět systém jako celek na lokálním počítači si vyžaduje značný výpočetní výkon, hlavně co do nároků na virtualizaci.

Architektura mikroslužeb si vyžaduje vyšší nároky na politiku zabezpečení systému, protože jednotlivé moduly musí uvažovat jako hrozbu nejen data přímo od klienta, ale i od ostatních modulů v systému. Toto je důležité kvůli ochraně před falešnými žádostmi nebo chybami v zabezpečení ostatních modulů aplikace. Na druhou stranu, pokud se bezpečnostní politika aplikace implementuje správně, je systém mnohem odolnější vůči útokům. Případné prolomení bezpečnosti v jedné logické větvi systému neohrozí systém jako celek. Mikroslužby nesmí spoléhat, že „nepříjemnou práci" za ně udělá jiná služba, která stojí v procesu zpracování uživatelského požadavku před nimi. Každá služba musí nezávisle ověřit validaci příchozího požadavku. Je to z toho důvodu, že předchozí služba může selhat nebo může být terčem nepřátelského útoku na systém. V případě, že by služby neprováděly kontrolu požadavků samostatně, mohl by případný útok ochromit současně více služeb a způsobit fatální škody, například únik dat nebo selhání chodu systému.

Z pohledu řízení lidských zdrojů jsou projekty založené na mikroslužbách více variabilní a jednotliví členové týmu mají často odpovědnost právě za svoji separovanou část systému odpovídající jejich odborné specializaci. Často tak spolu na projektu spolupracují vývojáři, kteří se odborně nepřekrývají, ale jsou propojeni právě skrze domluvenou strukturu abstraktního rozhraní v rámci informačního systému. Vývojáři mají přidělené právě ty moduly, které spadají do jejich odborné způsobilosti a profesní orientace.

Problém může vznikat v časových nárocích na projekt, jelikož architektura mikroslužeb často tvoří komplexní a jedinečná řešení. Lze pouze minimálně spoléhat na normované šablony nebo metodiky. V praxi to znamená redundantní pokrytí určitých aplikačních procesů ve více modulech současně.

2.2.1 Návrh architektury mikroslužeb

Prvním krokem při návrhu aplikace je důkladná analýza business modelu a definování základů požadované funkcionality systému. Analýza nastolí směr dalšího vývoje aplikací principu modularity systému mikrolsužeb. Krom business logiky aplikace je nutné předem promyslet vedlejší procesy nezbytné pro bezproblémový chod aplikace – testování, logování, vhodná bezpečností politika atd. [3]

Moduly moderní webové aplikace se z hlediska kontaktu s uživatelem dají rozdělit na dva druhy [5]:

- **Klientské moduly** – zajištují přímý kontakt s aktéry aplikace, tedy uživateli, administrátory, jinými systémy. Slouží k prezentaci dat vhodnou formou a odesílání uživatelských požadavků směrem na server. Příkladem dynamické webové aplikace, mobilní aplikace, desktopové aplikace.
- **Serverové moduly** – služby serveru sloužící ke zpracování přijatých požadavků, výběru, přidání, úpravě nebo smazání dat. Příkladem jsou aplikace typu REST API propojené mezi sebou nebo s databází.

Při návrhu klientských modulů platí pravidlo, že aplikace by měla uchovávat jenom velmi malé a nezbytné množství systémových informací sloužících pro plnění své funkce. Jde například o autorizační tokeny nebo veřejné klíče. U mobilních nebo desktopových klientských aplikací to může být privátní klíč vázaný s daným zařízením. Jejich primárním zdrojem dat je vždy serverová část aplikace. Data jsou získávána pomocí zasílání požadavků na server.

Klientské moduly kladou vysoký důraz na moderní prezentační technologie a aktuální požadavky UI/UX designu[2]. Mají často malou životnost, aby co nejvíce reflektovaly aktuální trendy. Zde se projevuje výhoda principu nahraditelnosti u architektury mikroslužeb. Výměna klientského modulu se dá provést okamžitě bez ovlivnění funkce služeb serveru. Uživatelé nejsou vůbec zasaženi dlouhodobým výpadkem a existuje pouze malé procento rizika selhání systému [2]. Mezi službami a klientským modulem existuje pouze abstraktní vrstva komunikačního protokolu a strukturovaných dat, která se s novou verzí klientské webové aplikace vůbec nemění.

Jelikož klientské webové aplikace jsou realizovány pomocí frameworků založených na javascriptu, a také na uživatelem zvoleném prohlížeči, bývá u nich problém se zabezpečením. Existují zde rozporuplná témata, například javascriptové kryptografické knihovny nebo spolehlivost internetového prohlížeče. Vývojář veškeré faktory nemůže příliš ovlivnit, ale měl by s nimi počítat při navrhování bezpečnostní politiky systému. U mobilních aplikací, které slouží jako klienti serverových služeb, je nutné řešit problematiku zastaralého hardwarového zabezpečení nebo verzí operačního systému.

Vzhledem k výše uvedeným bezpečnostním výzvám je nezbytné přizpůsobit bezpečnostní politiku systému na stranu serveru. V oblasti serverových služeb má vývojář nejvyšší možnou kontrolu nad bezpečnostní politikou aplikace jako celku.

[2] Rozvržení prvků grafického uživatelského rozhraní, aby bylo pro uživatele praktické, příjemné a pro společnost esteticky reprezentativní.

Architekturu mikroslužeb lze podřídit vnitřním návrhovým vzorům [6]:

- **Database per Service** – k databázi serveru má přístup pouze jedna konkrétní služba. Ta slouží jako zdroj databázových dat ostatním mikroslužbám v systému.
- **API Gateway** – klientské moduly přistupují k serveru skrze samostatnou službu, která plní funkci vstupní brány. Tato služba filtruje přijaté požadavky a směruje je na ostatní služby.
- **Client Side Discovery** – klientské a serverové moduly sdílí společní registr adres pro vzájemnou komunikaci. Klientský modul si je vědom adres jednotlivých služeb serveru. Na tyto adresy odesílá klientský modul své požadavky.
- **Server Side Discovery** – klientský modul zná pouze adresu jedné vstupní brány serveru. Ten obsahuje speciální mikroslužbu pro směrování přijatých požadavků od klientů ostatním službám.

Při migraci nebo zásadní úpravě datového úložiště nesmí systém jako celek změnu vůbec pocítit. Zde slouží návrhový vzor „Database per Service“, tedy vybrané datové úložiště obsluhuje pouze jedna vybraná mikrolslužba. Tím dojde k separaci dat od ostatních služeb od konkrétního databázového systému. Rovněž se sníží riziko chybné operace při komunikaci s databází, protože veškeré příkazy jsou obsluhovány jednou aplikací.

Existují dva návrhové vzory komunikace mikroslužeb s okolním světem. Prvním je návrhový vzor „Server Side Discovery“, kdy je jako kořen na straně serveru použita speciální služba v roli „vstupní brány“. Ta řídí veškeré požadavky a odpovědi systému vůči klientským aplikacím. Výhodou je možnost vytvoření určitého řádu pro komunikaci serveru s okolním světem. Z toho vyplývá odolnější vrstva zabezpečení. Z hlediska síťové komunikace také veškeré klientské moduly budou mít jenom jeden bod, kam budou směřovat uživatelské požadavky. To usnadňuje konfiguraci. Nevýhodou je riziko vzniku centralizovaných služeb a špatné rozvržení zátěže aplikace, protože centralizované služby zpravidla čelí vysokému provozu.

Druhou možností řešení komunikace služeb je návrhový vzor „Client Side Discovery“. Dojde k propojení mikroslužeb vzájemně mezi sebou a klientské aplikace přistupují přímo ke konkrétním mikroslužbám na straně serveru. Zvyšuje se tak složitost struktury propojení aplikace, tedy i časové nároky na vývoj. Vzniká problém obtížného komplexního testování systému. Webová aplikace stabilnější v případě incidentu. Také odolá náporu

velkého množství požadavků ze strany klientských aplikací, protože poskytuje vyšší míru decentralizace.

V praxi se používá kombinace obou řešení podle specifických požadavků systému. Například tedy použití dvou vstupních bran a k nim připojeným mikroslužbám. Požadavek vytvořený v klientské aplikaci pak propadává jednotlivými uzly až k mikroslužbě na straně serveru, která je v přímém kontaktu s databází. V návrhu architektury se implementuje více služeb v roli vstupní brány. Každá brána slouží specifické skupině klientských modulů. Pokud jedna vstupní brána selže, ostatní druhy klientů mají stále zajištěný přístup k datům. Příkladem je jedna vstupní brána pro webového klienta a druhá pro mobilní aplikace.

Existují dva odlišné principy vnitřní architektury jednotlivých mikroslužeb [2]:

- **SOA (Service Oriented Architecture)**
  - Architektura orientovaná na služby. V tomto případě služba představuje určitý koncový bod, který vykoná proces v systému. Jde o starší architekturu, která má dobrý logický základ, ale je obtížně implementovatelná. Problém leží v nedostatečné abstrakci systému, problému určit efektivní velikost aplikace nebo složité konfiguraci. Obecně však umožňuje podporu velkého spektra internetových protokolů nebo univerzálnější vrstvu pro větší škálu různorodých zařízení a systémů. Je velmi dobře standardizována.
- **REST (REpresentational State Transfer)**
  - Architektura orientovaná na zdroje. Každý koncový bod aplikace představuje zdroj dat přístupný klientům. U moderních internetových aplikací více preferovaná. Je však silně závislá na HTTP protokolu. Využívá jeho strukturu a atributy. Použitelná pro webové aplikace bez speciálních technologických požadavků, například datové proudy.

Pro stanovení konkrétních logických služeb uvnitř systému je dobré použít výsledek analýzy požadavků klienta – kvalitní business model. Z něj jsou pro následnou dekompozici klíčové aktivity, které bude aplikace poskytovat a domény, které budou v systému obsaženy.

Existují dva návrhové vzory dekompozice systému [6]:

- **Decomposite by Subdomain –** rozvržení vnitřních služeb podle priority domén systému. Tyto domény jsou identifikovány v úvodní business analýze webové aplikace (Domain Analysis).
- **Decompose by business capability** – rozvržení vnitřních služeb podle činností aplikace. Z úvodní analýzy se určí modely užití a pro každý se definuje konkrétní služba.

Oba dva druhy dekompozice se ve výsledku částečně překrývají. To je ale v pořádku, protože cílem obou metodik je vytvoření stabilního rozvržení na základě důkladné úvodní analýzy business modelu aplikace. Je logické, že klíčové požadavky budou zohledněny v každém ze zmíněných přístupů dekompozice. Pokud by se výsledek příliš lišil, mohlo by to nasvědčovat špatně provedené úvodní analýzy požadavků klienta.

Domény se obecně dělí na dva druhy [2] [3]:

- **Privátní –** jsou relevantní pouze v rámci dané mikroslužby, pro ostatní služby v návrhu systému nemají význam.
- **Sdílené –** Modely využívané napříč vícero službami. Je nutné tedy jejich sdílení, například pomocí sdílených knihoven v systému.

Příkladem privátní domény může být hypotetická doména „Transaction" reprezentující určitou finanční transakci uvnitř systému. V rámci hypotetického scénáře s touto doménou pracuje pouze služba pro finanční operace. Tato doména je privátní, protože jiné mikroslužby s ní nepracují. K transakci se ale může vztahovat hypotetická doména „Client" reprezentující klienta systému. Tato doména bude sdílená, protože s klientem může pracovat více mikroslužeb současně. Obecně vzato, modely představující aktéry aplikace jsou často sdílené napříč službami.

Příklady konkrétních mikroslužeb v architektuře pomyslné aplikace:

- Služba pro logování událostí aplikace.
- Služba pro autentizaci a autorizaci.
- Služba pro správu objednávek.
- Služba pro správu faktur.
- Služba pro správu skladu.
- Služba pro zpracování dat správců.
- Služba pro venkovní veřejné API.
- Vstupní brána.

Pro deklaraci doménových modelů nebo registrů adres jednotlivých služeb lze použít sdílené knihovny programovacího jazyka. Použití sdílených knihoven u webové aplikace na principu mikroslužeb může být dvousečná záležitost. V jednom ohledu jsou sdílené knihovny nástrojem, jak se vyhnout redundancím v kódu. Pomáhají udržovat určitou normu a stav doménových modelů napříč veškerými službami. Toto platí pouze, pokud je na základě realizace projektu předem jasné, že všechny mikroslužby serveru budou tvořeny pomocí jediného programovacího jazyka. Pokud by mikroslužby byly vytvořeny v různých programovacích jazycích, pak by sdílené knihovny nebyly napříč mikroslužbami kompatibilní. Vzniká tak závislost na konkrétní technologii a snížení univerzálnosti systému co do technologické variability. Některé programovací jazyky umožňují kompilaci normalizovaných bitových knihoven, které lze navzájem kombinovat. Příkladem jsou DLL knihovny [3].

2.2.2 Migrace z monolitické aplikace na aplikaci mikroslužeb

Jedná se o častý klientský požadavek pro údržbu řešení starších webových aplikací. Problém vzniká formou scénáře chybné úvodní business analýzy. Vzhledem k podhodnocení potenciálu růstu byla aplikace realizována pomocí monolitické architektury. V produkčním prostředí se aplikace osvědčí, začne u ní vzrůstající tendence celkového počtu pravidelných uživatelů. Ze strany společnosti dochází k rozšiřování systému o nové funkcionality. S nimi se zvyšuje i množství ukládaných dat a frekvence zpracovávaných požadavků. Nové funkce aplikace si vyžadují transformace entitní relačních modelů pro nové potřeby klientů a tím komplikování relací nebo porušování normálních forem dat.

Tento trend pak dojde až do kritického bodu, kdy monolitický framework již nedokáže splnit požadavky provozovatele na škálovatelnost a kvalitu poskytovaných služeb. Je tedy zapotřebí aplikaci zásadně upravit. Často jediným smysluplným řešením je právě migrace z monolitické aplikace na architekturu mikroslužeb. Tato operace není vůbec jednoduchá ani levná, jde o pečlivě naplánovaný a kontinuální proces.

Je nezbytné začít s tvorbou úplně nové paralelní aplikace včetně všech standardních fází vývoje, například analýzy požadavků společnosti a klientů, business analýza výsledné aplikace nebo doménové a objektové návrhy konečného řešení. Požadavky se mohou od původní aplikace výrazně změnit. Společnost již například počítá s další expanzí služeb nebo zaváděním nových produktů a složitějších technologií.

V návrhu nové aplikace je nezbytná pečlivost při rozvrhnutí struktury služeb u nové aplikace, určení jejich zodpovědností a vhodného rozdělení datových úložišť. Tato úložiště následně budou sloužit pro rozdělení dat z jedné univerzální databáze, která je součástí původní monolitické aplikace. V rámci vývoje je nutné stanovit vhodné milníky pro postupné dosažení plné funkcionality současné monolitické aplikace. [2].

Kritickou a rizikovou částí celého procesu změny architektury je migrace dat ze staré monolitické aplikace do aplikace nové postavené na architektuře mikroslužeb. Ačkoli oba systémy nabízejí z pohledu koncového uživatele stejné služby, entitně-relační model úložišť je většinou značně odlišný. Původní model je většinou příliš komplikovaný a bývá zakomponován do centralizovaného úložiště dat. Je proto nezbytné tento entitně-relační model rozdělit do menších celků odpovídajícím vícero koncovým databázím pro více samostatných aplikací – mikroslužeb.

Jde o rizikovou operaci, jelikož data jsou jednou z nejcennějších a nejchráněnějších částí celé aplikace. Data představují faktickou historii aplikace - důvěru, kterou uživatelé do nabízené služby svěřili při uložení. Ochranu dat ukládá i legislativa (GDPR 2018) [7], která je v tomto ohledu velice striktní. Nesprávné zacházení s daty jiných fyzických osob může mít za následek likvidační finanční postihy pro odpovědný podnikatelský subjekt.

Vhodným řešením pro přesun již dat ze staré databáze monolitické aplikace je tvorba pomocné aplikace pro mapování starého entitně-relačního modelu do nové aplikace. Data ze staré databáze vytvořený nástroj ukládá do nových úložišť, aby s nimi mohla nová aplikace na principu mikroslužeb efektivně pracovat. Mapovací nástroj musí v rámci své

aplikační logiky dokonale rozumět oběma stranám datových zdrojů. Musí být schopen předvídat a řešit možné problémy při procesu migrace dat, například chybějící hodnoty, duplicity, nebo odlišnosti datových typů. Je důležité přenesená data průběžně analyzovat. Snižuje se tak riziko chyby a nutnost provádět opravy procesu, nebo ho muset dokonce opakovat. Data uživatelů původní aplikace nesmí po procesu migrace nikde chybět nebo být fakticky pozměněna. Bohužel u velkých systémů se může jednat až o miliony záznamů a zajistit naprostou záruku bezchybného přenosu je velice obtížné. Zdroj dat původní monolitické aplikace by tak měl zůstat vždy pro jistotu archivován jako nouzová záloha.

Jakmile nově vyvinutá aplikace na principu mikroslužeb dosáhne bodu umožňující produkční nasazení, zpřístupní se klientům původní monolitické aplikace. Bývá zvykem ze začátku udržovat paralelní provoz obou. Tento paralelní provoz trvá během přechodného období, dokud nová aplikace neimplementuje veškerou funkcionalitu staré aplikace a nemá přesunuta veškerá relevantní data do nových úložišť. Konkrétní doba se liší podle potřeb projektu, ale u velkých systémů se může jednat o řády měsíců, nebo dokonce let. To může být komplikované, protože stará monolitická aplikace je paralelně v provozu - stále ukládá a mění aktuální data uživatelů. Po dosažení vyrovnaného stavu, kdy oba systémy nabízí obdobné služby svým klientům, je možné původní monolitickou aplikaci bezpečně deaktivovat. Uživatelé systému jsou přesměrováni v plném rozsahu na novou aplikaci postavenou na architektuře mikroslužeb. Další vývoj se pak soustředí pouze na aplikaci novou, ideálně nadále pomocí principu průběžné realizace.

3 Moderní podoba .NET platformy

Platforma .NET je rodina frameworků, programovacích jazyků, nástrojů a prostředí sloužící k vývoji aplikací pro web, servery, cloudová prostředí, mobilní zařízení, osobní počítače nebo herní konzole. Je primárně orientována na prostředí operačního systému Windows od společnosti Microsoft Corporation. Společnost zajišťuje vývoj a technickou podporu. Moderní podoba této platformy umožňuje rovněž vývoj pro operační systém Linux nebo mobilní operační systémy Android a iOS.

Značnou výhodou .NET platformy je komplexní rozsah. Umožňuje pomocí malého množství specifických nástrojů a programovacích jazyků pokrýt téměř celý proces realizace rozsáhlého informačního systému od návrhu business logiky, přes realizaci, správu zdrojového kódu až po vytvoření funkční verze aplikace určené pro publikování ve finálním prostředí. Propojenost všech nástrojů umožňuje vývojářům úsporu času při konfiguraci nebo správě projektu. Nevýhodou platformy .NET je vysoká cena komerčních licencí nástrojů a služeb, což může být hlavně pro menší společnosti problém. Obecně skoro všechny nástroje .NET platformy mají svoji bezplatnou verzi, ta ale často nepokrývá nároky rozsáhlého projektu.

.NET platforma zahrnuje velké množství programovacích jazyků. Hlavním je objektově orientovaný programovací jazyk C#. Krom objektového syntaktického základu nabízí výrazné prvky funkcionálního programování. Jazyk je spolu se zdroji programu kompilovaný do podoby IS (Intermediate Language). Tento stav se také nazývá „Managed Assembly“ nebo „Bytecode“ a má většinou podobu .DLL souboru. Následně prostředí CLR (Common Language Runtime) vytvoří strojový kód pro operační systém, který provede požadované instrukce. Podoba syntaxe C# vychází původně z jazyka C/C++.

Tento programovací jazyk obsahuje mnoho pokročilých funkcí [8]:

- **LINQ –** zabudovaný dotazovací jazyk pro extrakci dat z kolekcí a jejich úpravu.
- **Delegáti –** možnost definování metody jako budoucí proměnné uvnitř třídy nebo jiné metody.
- **Lambda výrazy –** funkcionální deklarace anonymních metod.
- **Asynchronní volání** – možnost snadno provádět konkrétní sadu instrukcí ve vedlejším vlákně procesu programu.
- **Rozšiřující metody** – injektování nových metod do již definovaných tříd externích zdrojů aplikace.
- **Tuples –** tvorba dynamických objektů složených z libovolných datových typů.
- **Použití destruktorů –** možnost vynuceného odstranění objektů z paměti.
- **Imutabilní kolekce –** kolekce obsahující prvky s neměnnými hodnotami.

Krom výše uvedených vlastností zahrnuje syntaxe jazyka C# standardní datové typy, operátory, vnitřní konstrukce jazyka, genericitu, reflexi vlastního kódu nebo OOP strukturu. Jedná se o nejčastěji používaný programovací jazyk platformy .NET pro všechny klíčové druhy aplikací.

Dalším velice populárním jazykem platformy .NET je F#. Jedná se o kompilovaný funkcionální programovací jazyk s omezenou podporou objektového návrhu. Aplikace napsaná v programovacím jazyce F# je složena z funkcí a modulů. Syntaxe jazyka umožňuje tvorbu snadných řešení problematiky rozhodovacích stromů, řetězení funkcí pomocí principu „roury“, úsporu rozsahu kódu díky eliminaci nepotřebných znaků (Code Debris). Tento jazyk je primárně imutabilní s volitelnou možností tvorby referencí. V rámci .NET slouží k tvorbě funkcionálních DLL knihoven pro jiné projekty. Tyto knihovny jsou vzájemně kompatibilní s jinými programovacími jazyky uvnitř .NET platformy. Existuje možnost nativního vývoje multiplatformních mobilních aplikací pomocí nástroje Xamarin.

Dalšími programovacími jazyky, které platforma .NET podporuje jsou například:

- Javascript / Typescript
- C++
- Python / IronPython
- VB.NET

Pro vývoj aplikací je krom programovacího jazyka důležité kvalitní vývojové prostředí a pomocné nástroje. Hlavním vývojovým prostředím platformy .NET je Visual Studio IDE. Tento nástroj americké softwarové společnosti Microsoft Corporation vznikl v roce 1997. Současná verze Visual Studio 2017 se nabízí ve třech verzích. První verze se jmenuje Community, má licenci k použití zdarma. Cílí především na samostatné softwarové vývojáře a studenty. Zbylé dvě edice, Professional a Enterprise, mají komerční licenci a jsou určené pro softwarové společnosti.

Vývojové prostředí Visual Studio IDE vývojářům umožňuje:

- Psaní a editaci zdrojového kódu.
- Debugging[3] zdrojového kódu.
- Konfiguraci projektů.
- Tvorbu instancí aplikace.
- Analytiku výkonu spuštěných aplikací.
- Vyhodnocování jednotkových a integračních testů.
- Přímé napojení na externí vývojové nástroje.
- Tvorbu produkčních verzí aplikací a jejich publikaci.

Z hlediska vývojového procesu nabízí komerční verze Visual Studio IDE pokrytí všech potřeb pro realizaci aplikace. Nevýhodou vývojového prostředí Visual Studio IDE je požadavek značného prostoru na systémovém disku počítače. Instalační nástroj sice umožňuje částečnou instalaci na datový disk, přemístí se ale pouze přibližně třetina souborů.

Během tvorby informačního systému potřebují produkční společnosti nástroje pro správu procesu vývoje. Mnoho vývojářů .NET platformy využívá nástroj

[3] Hledání chyb uvnitř aktivní aplikace na základě tvorby kontrolních bodů a sledováním hodnot proměnných ve zdrojovém kódu.

Microsoft Team Foundation Server (TFS) [9]. Jedná se o produkt společnosti Microsoft Corporation pro implementaci agilních metodik vývoje informačních řešení, řízení práce vývojového týmu, komunikaci jeho členů, správu zdrojového kódu, tvorbu stabilních verzí aplikace, testování produkčních verzí systému a následné publikování do produkčního prostředí. TFS podporuje dvě základní metodiky agilního vývoje projektu, Agile a Scrum. Nástroj TFS je zdarma pro vývojové týmy s maximálně pěti členy. Pro větší společnosti je k dispozici za měsíční poplatek. TFS je nativně integrováno do vývojového prostředí Visual Studio IDE.

Jakmile vývojáři připraví otestovanou produkční verzi aplikace, je nutné ji poskytnout koncovým klientům. Platforma .NET nejčastěji používá cloudové virtuální prostředí Microsoft Azure pro sestavení hotových informačních řešení. Nabízí širokou paletu služeb, které je možné vzájemně kombinovat. Microsoft Azure nabízí publikování webových aplikací, databází, datových skladů, virtuálních sítí a další podpůrné služby chodu informačního systému. Popřípadě lze na platformě Microsoft Azure konfigurovat unifikované virtuální počítače.

V roce 2016 oficiálně představila společnost Microsoft Corporation strukturu moderní podoby platformy .NET rozdělenou do tří hlavních pilířů vývoje podle cílové platformy [10]:

- **.NET Framework –** Skupina frameworků pro vývoj aplikací určených pro operační systémy Microsoft Windows a Microsoft Windows Server.
- **.NET Core –** Skupina frameworků umožňující multiplatformní vývoj bez nutného ohledu na cílový operační systém.
- **Xamarin –** Skupina frameworků pro vývoj multiplatformních mobilních aplikací.

Platforma .NET je velice oblíbená pro vývoj komerčního softwaru, například v oblastech bankovnictví, energetického průmyslu, aerolinek, ecommerce, finanční analytiky nebo podnikových systémů. Naopak není příliš preferována v oblasti multimediálních služeb nebo sociálních sítí. Tyto oblasti informatiky se technologicky opírají o technologie na bázi programovacího jazyka Java, Ruby nebo Python.

3.1 Webové frameworky ASP.NET

ASP.NET je skupina frameworků spadající do vývojové platformy .NET zaměřená na vývoj webových aplikací a služeb. Je hojně využívána komerčními společnostmi pro vývoj moderních online aplikací postavených na jakékoli webové architektuře.

Hlavní frameworky spadající pod ASP.NET:

- **WebForms** – Starší a dnes již velmi málo používaný monolitický framework postavený na myšlence dynamických internetových formulářů. Samotná architektura vychází z předpokladu vnímání webové aplikace jako kolekce sémanticky strukturovaných stránek. Samotný framework nijak zvlášť neimplementuje dynamickou aplikační logiku. Umožňuje snadnou tvorbu webových formulářů díky speciálním formulářovým prvkům nad rámec klasických prezentačních webových technologií. Jednotlivé stránky vytvořené v tomto frameworku mají podobu souborů s koncovkou ASPX. Internetový prohlížeč k nim přistupuje přímo. Aplikace má jasně dané směrování na základě adresářové struktury.
- **Web API** – Starší framework sloužící pro tvorbu jednoduchých REST služeb. Může fungovat jako alternativa SOA webových služeb realizovaných v platformě .NET většinou pomocí frameworku WCF. REST API aplikace vytvořené pomocí Web API fungují primárně přes HTTP protokol. Aplikace vrací výhradně strukturovaná data. Web API podporuje konfiguraci směrování požadavků a jejich průchod kolekcí vstupních metod (HTTP Message Handlers).
- **MVC** – Monolitický framework postavený na Model-View-Controller návrhovém vzoru. Používá nástroje pro tvorbu pohledů (Views) - Razor, JQuery, Ajax. Výsledkem zpracovaného požadavku uživatele by měl být primárně vygenerovaný pohled - kompletní stránka ve formátu HTML určená pro prohlížeč uživatele. Framework je serverově orientovaný, generování výsledku se tedy provádí na straně serveru před odesláním výsledku zpět v podobě HTTP odpovědí. Díky tomu není vhodný pro složitější dynamickou prezentaci dat.
- **Core** – Nejnovější framework spojující MVC a WebAPI do jedné univerzální podoby „webové aplikace“ s jednotnou konfigurací a společnými knihovnami tříd. Umožňuje multiplatformní vývoj, je více orientován na architekturu mikroslužeb.

Zůstává plná podpora monolitických MVC aplikací včetně všech nástrojů prezentační vrstvy.

Důležitým prvkem klasických aplikací ASP.NET je Open Web Interface (OWIN). Jde o nástroj pro vytvoření standardizované vrstvy abstrakce mezi webovým serverem a webovou aplikací. OWIN se v ASP.NET MVC používá k definování vstupní společné kolekce metod pro zpracování všech HTTP požadavků (middlewares). Zároveň umožňuje u starších aplikací zpracovávat modernější postupy autentizace a autorizace, například protokol OAuth 2.0. Tento protokol je důležitým standardem při využití třetích stran právě pro autentizační a autorizační procesy. Příkladem je například přihlášení skrze sociální sítě. OWIN lze ještě rozšířit o nástroje pro použití datových proudů, Input/Output operací nebo websocketů, podle RFC 6455.

Frameworky v ASP.NET je možné společně kombinovat do jedné webové aplikace. Velmi častou kombinací bývá kupříkladu spojení ASP.NET MVC společně s ASP.NET Web API. Výsledná aplikace nabízí uživatelský prostor pro prezentaci obsahu (MVC) a datovou bránu (API) pro čtení nebo poskytování veřejných dat třetím stranám. Jelikož se však jedná o odlišné frameworky, vyžaduje si takový projekt dvojí konfiguraci, například pro nastavení směrování, vstupní procesy zpracování uživatelského požadavku nebo autentizačních a autorizačních procesů. Takové spojené projekty jsou díky komplikovanému nastavení často nepřehledné a zvyšuje se riziko chyby v aplikaci. Ačkoli je takový systém složen ze dvou různých frameworků, architektura je stále monolitická. Jak uživatelské požadavky pro získání grafického výstupu, tak požadavky na strukturovaná data z API jsou spravována jednou instancí aplikace se společnou sadou zdrojů pro výkon a paměť.

Prostředí pro publikování hotových aplikací v ASP.NET jsou v podstatě veškeré hostingové a cloudové služby, které podporují operační systém Microsoft Windows Server. Jde tedy jak o standardní levnější hostingy, tak o komplexnější cloudová prostředí. Framework ASP.NET Core nabízí reálnou možnost nasazení na jakémkoli webovém serveru, jelikož podporuje prostředí na bázi operačního systému Linux.

4 ASP.NET Core

Tento framework byl společností Microsoft Corporation původně představen pod označením ASP.NET 5, tedy s pořadovým číslem navazujícím na starší verzi ASP.NET 4 [11]. Nejednalo o konečnou verzi frameworku určenou pro komerční vývoj. Byla uvolněna k testování a optimalizaci odbornou veřejností. Od starších verzí ASP.NET byly znatelné zásadní rozdíly ve struktuře realizovaných projektů. Lišila se konfigurace, adresářová struktura a podpora externích technologií, které patří mimo platformu .NET. ASP.NET 5 bylo později přejmenováno na ASP.NET Core. Tento název a označení bylo konečné až do oficiálního vydání komerční verze [11] [12].

Původním cílem ASP.NET Core bylo oproštění frameworku od závislosti vytvořených aplikací na platformě Microsoft Windows Server a umožnění publikování aplikace pro jiná produkční prostředí. Druhým cílem bylo technické sjednocení starších frameworků ASP.NET do univerzálního frameworku pro serverově orientované aplikace s názvem „Web Application“. Veškeré konstrukce webových aplikací (MVC i REST API) mají jednotnou konfiguraci, jádro a knihovny tříd. U ASP.NET Core odpadávají například potíže s odlišným směrováním při kombinaci různých frameworků, problematická implementace OWIN, autentizační a autorizační procesy nebo implementace sessions. Přibyla nativní podpora nástrojů pro vývoj aplikací dynamické prezentační vrstvy (Angular nebo ReactJS). ASP.NET Core podporuje abstrakční vrstvu OWIN, ale pouze na vyžádání. Je určena pro webový server, který tuto vrstvu abstrakce vyžaduje. Nastavení OWIN se provádí v konfiguračním souboru.

Jádro aplikace ASP.NET Core tvoří aplikační webový server Kestrel [13]. Webová aplikace stojí na procesu zpracování HTTP požadavku a vytvoření HTTP odpovědi odeslané zpět klientovi. Kestrel je funkcionálně sestaven pomocí knihovny „libuv“. Jde o knihovnu pro asynchronní input/output operace navrženou mimo jiné také pro podporu Node.JS. Takto knihovna umožňuje [14]:

- Asynchronní použití protokolů TCP a UDP.
- Asynchronní použití DNS rozpoznávání.
- Asynchronní použití zpracování souborů a s nimi spojených událostí.
- IPC se sdílením socketů.
- Použití Unixových doménových socketů a pojmenovaných „pipelines“.
- Řízení dceřiných procesů.
- Obsluhu signálů.
- Časování s vysokým rozlišením.
- Řízení chodu a prostorů vláken.
- ANSI kontrolované přes TTY.

Klíčovým objektem ASP.NET Core je instance třídy HttpContext. Tento objekt představuje kontext HTTP požadavku. Po celou dobu zpracování uživatelského požadavku na webový server nese informace o příchozím HTTP požadavku, identifikaci žadatele, nabízených interních službách a následně samotné HTTP odpovědi zpět ze serveru. Během zpracování HTTP požadavku klienta si příchozí informace nejprve převezme jádro aplikace Kerstrel. Tato instance serveru zpracované parametry přetvoří do kontextu požadavku (HttpContext). Kerstrel odešle kontext požadavku instanci specifického zdrojového kódu webové aplikace. Tato část ASP.NET Core aplikace je v moci vývojářů. Provede požadované operace a následně doplní informace o konečné HTTP odpovědi. Odpověď je přidána do kontextu požadavku. Ten se vrátí zpět instanci webového serveru Kerstrel, který odešle HTTP opověď zpět klientovi.

V praxi je možné proces doplnit o proxy server, například IIS., Nginx nebo Apache. Tento model se používá v případě, kdy vícero aplikací sdílí jednu IP adresu a číslo portu na témže serveru. Takovou situaci Kerstrel přímo nepodporuje. Vložení proxy serveru navíc umožňuje zvýšit úroveň síťové bezpečnosti aplikace, nastavit hranici vytížení, prostoru a

zdrojů pro instanci webové aplikace, nebo zvýší možnou kompatibilitu u specifických zařízeních.

Odklon ASP.NET Core od platformy tradiční platformy .NET byl u první verze příliš velký. Ta vyšla v květnu roku 2016 pod označením ASP.NET Core 1.0. Objevily se u ní potíže s kompatibilitou v rámci zbytku platformy. Nevyřešenou otázkou první verze byla implementace jednotkových testů. Provést konfiguraci testovacího prostředí bylo velice náročné. Klasické testovací nástroje (např. MSTEST) nebyly kompatibilní, protože nešlo navzájem odkazovat na reference mezi knihovnami projektů. Projekty ASP.NET Core 1.0 nebyly kompatibilní s klasickými projekty typu CSPROJ. Akutním řešením byla implementace nezávislého nástroje pro jednotkové testy XUnit. Ten testování umožňoval, ale celý proces konfigurace testovacího prostředí byl značně komplikovaný, protože vyžadoval použití pouze specifických verzí ASP.NET Core a testovacího nástroje.

Verze ASP.NET Core 1.1, která přišla na podzim roku 2016, tento problém vyřešila. Byla kompatibilní s CSPROJ projekty a standardními knihovnami platformy .NET. [15].

Ve verzi ASP.NET Core 2.0, vydané v roce 2017, byla zavedena nativní podpora populárních prezentačních frameworků ReactJS a Angular v rámci kombinovaných monolitických aplikací. Byla upravena konfigurace zabezpečení webové aplikace, autentizaci a autorizaci, kdy se pro tyto procesy začaly preferovat vnitřní služby webové aplikace.

Pro rok 2018 je společností Microsoft Corporation připravována verze frameworku ASP.NET Core 2.1, užitečné novinky, například možnost tvorby odpovědí na HTTP požadavky pomocí generických objektů, propracovanější atributy a filtry pro vstupní požadavky klientů nebo výchozí nastavení webové aplikace, aby mohla komunikovat pouze v šifrovaném spojení. Šifrovanou vrstvu zajišťuje technologie SSL, samotná komunikace probíhá pomocí protokolu HTTPS, který je pro šifrované připojení přímo navržen.

4.1 Architektura aplikace

ASP.NET Core stojí na architektonickém vzoru Model-View-Controller. Ten je v praxi rozšířen na Model-View-ViewModel-Controller kvůli extrakci relevantních informací z hrubých modelů a lepší optimalizace procesů zpracování požadavku klienta [4] [15].

Jednotlivé složky obecného MVVC vzoru:

- **Model:** Strukturální datová entita, reprezentuje abstraktní objekt skutečného světa, se kterým aplikace pracuje. Model tvoří libovolná třída programovacího jazyka. Obsahuje především atributy a relace relevantní k doménovému modelu aplikace. Tato konstrukce modelu může sloužit vybranému ORM nástroji pro vytvoření entitně-relačnímu modelu v databázi. Model je možné použít k validaci dat pohledu (View) nebo k tvorbě vhodných formulářových prvků.
- **View:** Reprezentuje zdroj pro vykreslení strukturovaných dat uživateli. Má většinou podobu generovaného HTML kódu s referencemi na kaskádové styly a javascriptové funkce. Pro tvorbu pohledu se v ASP.NET Core používá nástroj Razor, který generuje konečný HTML kód z dynamického zadání odpovídajícímu kódu v programovacím jazyce. Podporuje například cykly, podmíněné zobrazení nebo generátory specifických HTML elementů. Pohled by neměl provádět složité úpravy dat (formátování, filtrace, řazení), pokud to není nezbytné.
- **View-Model:** Výběr relevantních dat z modelů získaných v repositáři nebo databázovém kontextu. Každý pohledový model je určen pro jeden specifický pohled. Finálním stavem je převedení komplexních objektů na primitiva, eliminuje se tak riziko zacyklení překladače dat z objektového do strukturovaného textového formátu (JSON, XML, CSV apod.). Slouží buď jako datový podklad pro pohled, nebo tvoří předlohu pro sterilizaci datového modelu poskytovaného webovou REST API službou.
- **Controller:** Řídící element aplikace. Je potomkem třídy Microsoft.AspNetCore.Mvc.Controller. K sestavení instance se používá HTTP kontext daného požadavku. Lze do něj injektovat interní služby webové aplikace, má oprávnění pracovat s identitou uživatele a také vytváří HTTP odpověď pro daný požadavek. S použitím repositářů obstarává data z úložiště a převádí je do požadovaného formátu. Veřejné metody controlleru se nazývají akce (Actions). Název controlleru a každé z akcí je podstatný při tvorbě směrování pro zpracování

HTTP požadavku. Controllerům a akcím se mohou přiřadit speciální vlastnosti skrze atributy filtrů (vysvětleno dále). Akce controlleru podporují jak synchronní tak asynchronní zpracování požadavků.

ASP.NET Core je serverově orientovaný framework, veškerá funkcionalita a proces zpracování HTTP požadavků probíhá tedy na straně serveru. Výjimkou může být zabudovaná dynamická funkcionalita webového rozhraní v browseru vytvořená pomocí javascriptu uvnitř generovaných pohledů. V případě serverových REST aplikací je pohled nahrazen strukturovanými daty, která jsou poté zpracována klientskými aplikacemi.

4.2 Princip komunikace a práce s protokolem

Aplikace vytvořené pomocí ASP.NET Core využívají pro síťovou komunikaci HTTP protokol. Výměna dat probíhá vždy mezi dvěma body - aplikací v roli klienta (prohlížeč) a aplikací v roli poskytovatele služeb (server). Komunikačními entitami jsou HTTP požadavky ze strany klienta a po jejich vyhodnocení HTTP odpověď ze strany serveru. [13].

HTTP požadavek klienta na server je obecně označován jako HTTP Request Message. Validní HTTP Request je složen za tří základních částí [16]:

- **Request Line** – zahrnuje adresu, metodu a verzi HTTP protokolu.
- **Request Header** – seznam popisných hodnot požadavku (host, autentizace, jazyk, formát, vlastní hlavičky).
- **Request Body** – Obsah požadavku, například data z webového formuláře.

Každý požadavek potřebuje správně zvolenou adresu, kam je směřován a konkrétní metodu získání dat (Request Method). Tato metoda definuje způsob, jak s daty požadavku manipulovat. Metody se liší podle typu a určení požadavku klientské aplikace:

- **GET** – Žádost o získání dat ze serveru.
- **POST** – Vložení nových dat na server.
- **PUT** – Úprava zdroje dat serveru.
- **DELETE** – Požadavek na smazání dat ze serveru.
- **HEAD** – Žádost o stručnou odpověď v podobě hlavičky HTTP protokolu.
- **OPTIONS** – Žádost o výpis metod, které server podporuje.
- **PATCH** – Úprava datové entity uložené na serveru.

Se všemi těmito metodami umí ASP.NET Core efektivně pracovat v průběhu vyřizování požadavku. Ne všechny HTTP metody jsou v praxi praktickým řešením. Například metoda DELETE funguje podobně jako metoda GET - parametry nejsou předávány v těle požadavku, ale v URL adrese. Je tedy velice transparentní a snadněji zneužitelná. V praxi je nahrazována metodou POST, která citlivá data uschová uvnitř těla HTTP požadavku.

Adresa URL uvnitř požadavku obsahuje doménu a cestu k cílové metodě aplikace, která na principu REST obsluhuje zdroje. Cílem odeslaného požadavku je dosáhnout průchodu aplikací ASP.NET Core a získat odpověď obsahující strukturovaná data obsažená v těle

HTTP odpovědi. HTTP odpověď serveru je zpětnou reakcí na přijatý požadavek klienta. Na každý požadavek, který dorazí na server, musí ze strany webové aplikace následovat určitá odpověď. V opačném případě se předpokládá chyba spojení mezi serverem a klientem.

HTTP Respononse Message se skládá ze tří hlavních částí:

- **Status Line** – Obsahuje verzi HTTP protokolu, návratový kód (Status Code) a vysvětlující frázi ke kódu.
- **Response Headers** – obdobné hodnoty jako u požadavku (viz. Výše).
- **Response Message Body** – data navrácená serverem v podobě strukturovaného textu (HTML, JSON, XML, CSV).

Některé požadavky klientských aplikací nepotřebují žádná data v těle odpovědi - například pokud je cílem požadavku pouze potvrzení validity spojení. Stačí jim informace uvedená v hlavičce HTTP odpovědi. V takovém případě se používá metoda volání HEAD, která klientovi v odpovědi poskytne specifický návratový HTTP kód s vyplněnou hlavičkou. Služby vytvářené v ASP.NET Core velice často používají návratové kódy k pokrytí všech předpokládaných stavů komunikace.

Nejčastěji používané kódy HTTP odpovědi:

- OK (200) – Proces zpracování požadavku proběhl v pořádku.
- Created (201) – Úspěšná odpověď na požadavek metodou POST, došlo k vložení nových dat skrze službu.
- BadRequest (400) – Chyba při zpracování požadavku. Důvodem je chyba požadavku, například obsahuje nevalidní data.
- Unauthorized (401) – Pokud se požadavek snaží získat data, ke kterým nemá klient povolený přístup.
- NotFound (404) – Data požadovaná klientem nebyla nalezena.
- InternalServerError (500) - Chyba při zpracování požadavku. Důvodem je chyba na straně serveru, například selhání databáze.

O vložení správného návratového kódu rozhoduje buď cílová akce controlleru, nebo metoda pro filtraci HTTP požadavků. V obou případech aplikace generuje odpověď serveru na obdržený požadavek.

4.2.1 Middlewares a filtry

Middleware je speciální třída, jejíž kontrolou musí projít každý příchozí požadavek na server. Kolekce middlewares tvoří příchozí „rouru“ (Pipeline) společnou pro všechny HTTP požadavky vystupující do webové aplikace ASP.NET Core. K požadavkům se jednotlivé middlewares dostávají skrze injektovaný objekt „HttpContext“, který byl při přijetí požadavku vytvořen jádrem aplikace. Tento objekt si postupně middlewares předávají v definovaném pořadí. Během průchodu mohou být příchozí požadavky pozměněny, nebo být zamítnuty [17].

Třída middleware obsahuje [4]:

- Delegáta[4] pro přesun HTTP požadavku do dalšího modulu.
- Metodu „Invoke“, která provádí operace nad daným požadavkem.
- Objekt „HttpContext“, který obsahuje informace o požadavku klienta.

Middlewares se starají primárně o autentizaci a autorizaci klientů webové aplikace, nastavení správců výjimek (Exception Handlers) nebo nastavení směrování požadavků v aplikaci. Po deklaraci jejich tříd se do webové aplikace přidávají v konfiguračním souboru „Startup.cs“ uvnitř metody „Configure“.

Je velice důležité dbát na pořadí volání konkrétních middleware, jelikož jde o princip „roury“. Úpravy kontextu požadavku se kumulují v nastaveném pořadí. Pokud dojde k chybě v pořadí volání specifických middlewares, může zpracování požadavku selhat. Ačkoli mají middlewares definovanou vnitřní strukturu metod, nejsou potomkem žádné rodičovské třídy a neimplementují ani žádné rozhraní. Podporují jak synchronní tak asynchronní zpracování požadavků.

Jakmile projde požadavek všemi middlewares nakonfigurovanými vývojáři, dostane se jeho obsah až k závěrečné middleware určené pro směrování požadavků uvnitř aplikace. Tato middleware přepošle kontext požadavku na konkrétní definovaný controller. Tam je filtrován a odkazován ke konkrétní cílové akci controlleru.

Filtry se používají se pro úpravu funkcionality akcí controlleru. Je možné filtr natavit pro jednu konkrétní akci, nebo rovnou plošně pro celý controller. V takovém případě se filtr aplikuje pro všechny akce controlleru u všech požadavků.

[4] Proměnná v jazyce C#, která pro dosazení očekává implementaci metody se specifickými parametry.

Typy filtrů v ASP.NET Core jsou [18]:

1. **Autentizační filtry** – ověřují identitu požadavku a uživatele.
2. **Zdrojové filtry** – provádí se optimalizace výkonu a zdrojů aplikace, například caching.
3. **Filtry akcí** – provádí se před a po provedení akce v controlleru.
4. **Filtry výjimek** – kontrolují neočekávané výjimky vzniklé v akci.
5. **Filtry výsledků** – provádí aplikační logiku úspěšné akce v controlleru.

Vývojáři definované filtry se tvoří pomocí implementace rozhraní filtru v nové třídě. Implementace nového filtru spočívá v definování konkrétní aplikační logiky při zachycení události aplikačního procesu (Event). Tato událost je spouštěna voláním filtru uvnitř aplikace, například provedením akce controlleru.

Filtry se do aplikace přidávají vložením speciálního pojmenovaného atributu obsahující jméno filtru. Atribut filtru se přidává nad metodu akce, nebo nad název třídy controlleru. Je možné aplikovat více filtrů současně.

4.2.2 Application Services

Jednou z hlavních odlišností oproti předchozím verzím ASP.NET je podpora interních služeb aplikace (Application Services). ASP.NET Core pro práci s interními službami používá Dependency Injection Container, který je postaven na principu IoC (Inversion of Control) - moduly webové aplikace vyšší úrovně nesmí být silně závislé na modulech nižší úrovně. Oba druhy modulů jsou závislé na vrstvě abstrakce uvnitř aplikace [19].

Interní služby se do instance aplikace přidávají v konfiguračním souboru Startup.cs, konkrétně v metodě „ConfigureServices“. Samotná aplikační logika služeb je definována v samostatném souboru třídy spolu s rozhraním abstraktní vrstvy. Aplikační logika interní služby je naprosto libovolná a záleží pouze na vývojáři. Mezi interní služby patří často datové repositáře, databázový kontext, překladače dat, služba datové ochrany nebo instance analytických nástrojů.

Služby v ASP.NET Core se dělí na tři typy [4] [15] [19]:

- **Singleton –** Instance se vytvoří při prvním volání služby a zůstane uložena po celou dobu chodu aplikace. Slouží jako náhrada staršího „ApplicationState“. Jde o globální proměnnou společnou pro všechny požadavky.
- **Scoped** – Instance služby se vytvoří pro každý HTTP požadavek, který o ni požádá. Instance zůstane stejná po celou dobu vyřizování požadavku. Příkladem použití je například interní služba datového repositáře nebo třídy databázového kontextu pro daný controller.
- **Transient –** Pro každé volání služby se vytváří nová instance. Vhodné například pro injektované kryptografické operace řízené náhodnými hodnotami.

Interní služby webové aplikace jsou oproti starším verzím ASP.NET velkým posunem vpřed, jelikož umožňují snadnou implementaci pomocí injektování klíčových objektů kdekoli v aplikaci. To funguje na principu vložení abstraktní reference do konkrétního konstruktoru nebo parametru metody. Referenci ve zdrojovém kódu automaticky nahradí instance z kolekce interních služeb. Tato kolekce je obsažena uvnitř kontextu požadavku (HttpContext).

Díky podpoře principu Dependency Injection existuje silná vrstva abstrakce proti definovaným rozhraním. Lze tak snadno vytvářet potřebné jednotkové testy a třídy imitující jiné složité objekty určené k jednotkovému testování (Mock). Následně je možné simulovat reálné datové objekty aplikace a jejich chování.

Častým využitím interních služeb aplikace je implementace návrhového vzoru „Repository Pattern“. Služba repositáře definuje prostřední abstraktní vrstvu mezi controllerem a logickou reprezentací databáze (databázový kontext). Repositář má definované metody pro vykonávání konkrétních operací s daty, které jsou dostupné controlleru. Repositář následně realizuje změny datových hodnot uvnitř databáze pomocí jejího kontextu nebo použitím databázového klienta.

4.3 Konfigurace aplikace

Přepracované způsoby konfigurace webové aplikace jsou hlavním zdrojem inovace ASP.NET Core. Oproti starším verzím ASP.NET se konfigurace výrazně zjednodušila. Zahrnuje menší množství konfiguračních souborů, rozsah deklarací a formát samotných konfiguračních souborů. Sjednocení dvou samostatných frameworků Web API a MVC do jedné univerzální webové aplikace vedlo k jednotné konfiguraci libovolné webové aplikace.

Hlavním místem pro konfiguraci ASP.NET Core aplikace je soubor „Startup.cs“. Nastavení aplikace je realizováno výhradně deklarací vhodných middlewares a aplikačních služeb. Řada z nich je předpřipravena již v rámci výchozích knihoven frameworku. V klíčových metodách konfiguračních tříd se nastaví reference na konkrétní hodnoty skrze anonymní třídu, nebo se dosadí konkrétní hodnoty.

Příklady částí aplikace konfigurovaných pomocí interních služeb [4]:

- Autentizace a autorizace[5].
- Nastavení CORS[6].
- Nastavení Sessions.
- Nastavení připojení k databázi a kontextu databáze.
- Použití ochrany dat.
- Použití MVC modelu.

Příklady částí aplikace konfigurovaných pomocí middlewares:

- Nastavení směrování v MVC modelu.
- Nastavení autentizace a autorizace.
- Použití statických souborů.
- Použití chybové stránky.
- Nastavení parametrů CORS

[5] Do verze ASP.NET Core 1.1 nastavení probíhalo v middleware, dále už pomocí interních služeb.
[6] Výpis podporovaných domén pro přijetí HTTP požadavku.

Konfigurační soubor „Startup.cs“ je tvořen stejnojmennou třídou, která obsahuje tři metody:

- **Konstruktor –** Slouží k přidání vedlejších statických konfiguračních souborů do aplikace. Tyto soubory obsahují většinou informace o prostředí nebo podobu řetězce pro připojení k databázi (Connection String).
- **ConfigureServices –** Pomocí této metody jsou do webové aplikace zakomponovány všechny požadované interní služby. Ty přidá instance kolekce interních služeb (IServiceCollection). Webová aplikace ASP.NET Core pak může injektovat interní služby, kdekoli je potřeba.
- **Configure –** Metoda pro přidání a konfiguraci middleware a úvodního zpracování HTTP požadavku. Instance třídy implementující rozhraní „IAppBuilder“ pomocí metody „Use“ přidá do aplikace postupně všechny potřebné middleware moduly. Je potřeba dbát na pořadí. Přidávání modulů do „roury“ může být podmíněné aktuálním stavem aplikace.

V rámci konfigurace se pro zpřehlednění zdrojového kódu velice často používají „Extension Methods“. Jedná se o vlastnost programovacího jazyka C# umožňující přidání vlastní a pojmenované metody do již definované třídy. V tomto případě se do objektu „ApplicationBuilder“ v konfiguračním souboru přidá vlastní pojmenovaná metoda s danou logikou napsanou bokem v jiném souboru. Ke každé interní službě se definuje abstraktní vrstva pomocí rozhraní.

Vedlejší aplikační soubory obsahující statická data a hodnoty byly oproti starším verzím ASP.NET výrazně zredukovány. Většina z nich přešla na datový formát JSON oproti dřívějším XML souborům. Standardně se používají k definování požadovaných vývojářských balíčků pro danou aplikaci (skrze nástroje NuGet a Bower). Případně tyto soubory obsahují řetězce pro připojení k databázi. Úpravou těchto souborů je možné automaticky doinstalovat do aplikace požadované knihovny externích rozšíření.

4.3.1 Směrování požadavků v aplikaci

Jak již bylo uvedeno v popisu principu komunikace webové aplikace ASP.NET Core, důležitou informací uvnitř každého přijatého požadavku je adresa cílové metody uvnitř controlleru (akce). V případě ASP.NET Core nemá internetový prohlížeč přímý přístup do adresářové struktury webové aplikace. Odpověď serveru je dynamicky generována instancí webové aplikace jako výsledek zpracování HTTP požadavku vnitřními metodami [4] [1].

Aby ASP.NET Core aplikace věděla, na kterou akci controlleru má daný požadavek odeslat, je nutné v rámci konfigurace webové aplikace definovat principy směrování (routing). Z přijaté URL adresy ASP.NET Core dynamicky vytvoří cestu k určité akci ve vybraném controlleru. Aplikace se vždy pokusí identifikovat relevantní parametry ze struktury URL adresy přijatého požadavku.

Směrování uvnitř webové aplikace je založeno na definování šablony URL adresy společné pro všechny příchozí HTTP požadavky. Jednotlivé části šablony jsou poté identifikovány a substituovány. URL adresa definovaná uvnitř HTTP požadavku je na základě speciálních znaků separována na jednotlivé prvky – domény, adresu a doprovodné parametry. V rámci analýzy URL adresy záleží na pořadí prvků.

Konfigurace směrování se provádí buď globálně pro celou webovou aplikaci při definování parametru middleware s názvem MVC uvnitř konfiguračního souboru, nebo lokálně uvnitř pojmenovaného atributu controlleru nebo konkrétní akce. Každý controller nebo akce, může mít vlastní předepsanou cestu a požadované parametry uvnitř URL adresy. Pokud systém nenalezne žádnou metodu, která by požadované URL adrese vyhovovala, vrátí HTTP odpověď se statusem 404 (Not Found).

Uvnitř šablony URL adresy je možné použít:

- Pevné statické části požadované adresy.
- Dynamické části následně substituované podle názvu specifického controlleru nebo akce.
- Volitelné parametry, například identifikátor pro zpracování konkrétního elementu v databázi.

Díky spojení frameworků MVC a Web API do jediné webové aplikace je definice směrování prováděna společně na jednom místě. Dříve bylo nutné pro každý použitý framework definovat samostatný konfigurační soubor.

4.4 Práce s daty a jejich skladování

Jelikož je ASP.NET Core primárně serverově orientovaný framework, dostává se většinou do přímého kontaktu se zdroji dat – lokální databází nebo externím poskytovatelem [1] [4]. Framework ASP.NET Core není limitovaný konkrétním typem databázového systému. Je schopen vytvářet přímá ADO.NET spojení s libovolnou entitně-relační databází. V rámci tohoto spojení umožňuje ASP.NET Core volat uložené procedury a pohledy definované databázovým serverem. Druhou možností je požití nástroje pro ORM (Object Relational Mapping). Na platformě .NET se nejčastěji používá nástroj Entity Framework. Pomocí externích rozšíření ASP.NET Core je možné do projektu zakomponovat také NoSQL technologie, jako je například MongoDB.

Existuje otázka, jestli je vhodnější vyvíjet samostatný databázový projekt s nezávisle definovaným entitně-relačním modelem a procedurami, nebo použít nástroj ORM. Na jednu stranu dává samostatný databázový projekt vyšší kontrolu nad samotnou databází, jejím návrhem, procedurami, pohledy nebo spouštěči. Nicméně je nutné počítat s velkým množstvím duplicitní práce během vývojového procesu. Modely a CRUD (Create-Read-Update-Delete) metody ve zdrojovém kódu se velice často překrývají s entitně-relačním návrhem databáze a definovanými procedurami pro práci s daty.

Pokud aplikace ASP.NET Core potřebuje získávat data z externích zdrojů, využívá se standardně instance třídy HttpClient. S pomocí instance klienta je nezbytné sestavit kompletní HTTP požadavek pro externí server, který data poskytuje. Tento požadavek musí obsahovat konkrétní HTTP metodu, autentizačních a autorizačních informace a samozřejmě správnou URL adresu koncového bodu zdroje dat. Následně je nutné

zpracovat získanou odpověď z externího serveru – zanalyzovat návratový kód, hodnoty v hlavičce odpovědi a strukturovaný obsah v těle odpovědi.

4.4.1 Entity Framework

Nástroj typu ORM od společnosti Microsoft Corporation určený pro prostředí ADO.NET. Nejde o nedělitelnou součást webového ASP.NET Core, ale v praktickém vývoji je využíván velice často [1]. Na základě modelů, relací a procedur, definovaných pomocí objektového programovacího jazyka, nástroj Entity Framework generuje odpovídající entitně-relační model v databázi webové aplikace. Umožňuje tvorbu dotazů a jejich aplikaci uvnitř databáze [4] [15].

ASP.NET Core používá verzi ORM nástroje Entity Framework Core. Při konfiguraci projektu a knihoven je zapotřebí opatrnost. Při použití jiné verze by měl projekt problém s kompatibilitou pro koncové platformy, například Linux servery.

Programovým základem nástroje Entity Framework je třída kontextu databáze. Webová aplikace ji definuje jako potomka rodičovské třídy „DbContext". Ten v aplikaci reprezentuje logickou entitu celé databáze. Kontext umožňuje přístup k datům, plnění databáze testovacími daty, konfiguraci entitně-relačního modelu, tvorbu tabulek, jejich parametrů, klíčů, indexů nebo výchozích hodnot. Pro připojení ke konkrétní databázi používá připojovací řetězec (Connection String). Ten je definován ve statickém konfiguračním souboru aplikace ASP.NET Core a vkládá se do objektu databázového kontextu prostřednictvím parametru v jeho konstruktoru.

Každý databázový kontext obsahuje generické kolekce typu DbSet. Ty reprezentují tabulky s daty. Nástroj Entity Framework podporuje dotazovací jazyk LINQ. Ten umožňuje tvorbu dotazů pro výběr dat z kolekcí programovacího jazyka C#. Entity Framework překládá dotazy deklarované v LINQ na klasické databázové SQL dotazy. Po provedení SQL operace v databázovém systému převádí výsledky volání na kolekce programovacího jazyka C#. Entity Framework podporuje jak synchronní, tak asynchronní volání metod. To s sebou nese výhody při práci s rozsáhlými tabulkami nebo komplexními dotazy.

Relace mezi entitami podporované v nástroji Entity Framework [20]:

- **One-to-One –** Každá instance objektu obsahuje jednu instanci druhého objektu.
- **One-to-Many** – Každá instance objektu obsahuje neomezené množství instancí druhého objektu.
- **Many-to-Many** – Mnoho instancí objektu obsahuje mnoho instancí druhých objektů. Tato relace již není od verze Entity Framework Core nativně podporována. Podle oficiální dokumentace společnosti Microsoft Corporation je nutné nahradit tuto relaci dvěma relacemi One-to-Many s využitím asociační třídy.

Modely databázových entit jsou definovány buď na principu AoP (Aspect Oriented Progamming) přímo uvnitř deklarace modelů v ASP.NET Core aplikaci, nebo pomocí FluentAPI – voláním posloupnosti metod pro tvorbu entitně-relačního modelu uvnitř databázového kontextu. Eventuálně je možné kombinovat obojí, ale není to praktické. Vzniká tak nepřehledná deklarace entitně-relačního modelu a zvyšuje se riziko chyby při jeho generování v reálné databázi.

Metoda pomocí atributů uvnitř modelů je vhodnější pro monolitické aplikace využívající pohledy. Umožňuje pomocí definování atributů nad parametry třídy definovat jak vlastnosti modelu, tak požadavky na validaci formuláře. U rozsáhlých modelů však vzniká nepřehledný zdrojový kód. Metoda deklarace relací za použití FluentAPI řeší nastavení relací pomocí instance třídy „ModelBuilder“, která se stará o sestavení entitně-relačního modelu v nástroji Entity Framework. Umožňuje přímo definovat vzájemné vztahy, klíče, indexy nebo názvy tabulek v databázi. Navíc, deklarace fungují oboustranně mezi modely v dané relaci. Není tedy nutné, na rozdíl od atributů, v modelech definovat relaci v každém jednom modelu.

Mapování modelů tříd na finální tabulky databáze funguje na principu migrací. Při každé změně entitně-relačního modelu v aplikaci je nutné připravit migraci. Tu tvoří kód programovacího jazyka pro provedení dané změny oproti aktuálnímu stavu. Následně se spustí realizace migrace skrze překlad instrukcí programovacího jazyka do SQL příkazu a odeslání tohoto požadavku databázovému systému.

Jednotlivé migrace jsou tvořeny a archivovány ordinálně. Je tedy možné stav databáze transformovat do předchozí verze entitně-relačního modelu.

Migrace jsou uloženy:

- V souborech uvnitř projektu webové aplikace na straně aplikace. Mají podobu kódu programovacího jazyka.
- V tabulce uvnitř spravované databáze. Tabulka má název „_EFMigrationsHistory“. Obsahuje historii migrací, která obnoví stav databázového entitně-relačního modelu, pokud by došlo ke ztrátě záznamů ve zdrojovém kódu webové aplikace.

Příkazy pro zpracování migrací se zadávají buď v příkazovém řádku operačního systému Windows, nebo uvnitř prostředí Visual Studio IDE v příkazové řádce konzole „Package Manager“.

Kontrola změn funguje oboustranně. Pokud Entity Framework zjistí, že byl upraven entitně-relační model ve zdrojovém kódu projektu, bude vyžadovat novou migraci do databáze. Stejně tak pokud dojde k úpravě modelu na straně databáze, například smazání sloupce pomocí SQL příkazu, bude nástroj Entity Framework vyžadovat návrat modelu do stavu poslední migrace.

Existují dvě strategie návrhu entitně-relačního modelu v nástroji Entity Framework:

- **Code First** – Počítá s novou prázdnou databází a projektem. Aplikace pak současně se svým vývojem tvoří a upravuje model databáze.
- **Entity First** – Počítá s již existující databází bez Entity Frameworku a novou aplikací na její správu. Nejprve je nutné implementovat model existující databáze do aplikace s Entity Framework, teprve poté provádět vývoj aplikace.

První strategie je vzhledem ke svobodě vývoje často privilegovaná. Nesvazuje vývojáře již zavedenými relacemi a objekty. Vše podléhá logice nové a průběžně vyvíjené aplikace. Druhá je podstatně složitější a komplikovanější. Existují vestavěné nástroje pro implementaci existujícího entitně-relačního modelu do projektu, stejně jako doporučené postupy a strategie. V praxi se ale jedná o nepříliš vhodný postup, protože znemožňuje vytvoření modelů podle potřeb nové webové aplikace. Vývojáři pak mají značně „svázané ruce“ při návrhu zbytku systému.

5 Zabezpečení ASP.NET Core REST aplikací

Na webové aplikace se oprávněně kladou extrémně vysoké bezpečnostní nároky. Zpracovávají největší množství citlivých dat ze všech druhů informačních systémů. Jednoznačně existuje vyšší míra rizika odcizení nebo znehodnocení dat oproti aplikacím, jenž pracují lokálním zařízení nebo síti. Problémem bezpečnosti moderních informačních systémů je hledání kompromisu mezi uživatelským komfortem a efektivní mírou zabezpečení [21]. Tyto dva faktory bohužel v praxi vzájemně kolidují. Vysoká míra zabezpečení si vyžaduje snahu uživatelů dodržovat přísné bezpečnostní protokoly. Moderním kompromisem je použití biometrické informace uživatele. Tento druh identifikace sice nemá tak vysokou míru entropie[7], jako náhodně generované heslo, ale je pro uživatele velice komfortní. Míra bezpečnosti je u biometriky dostatečně velká pro většinu běžných uživatelských aktivit. Webové aplikace však stále nemají příliš propracované systémy pro využití biometrického kódu.

Před rozhodnutím o konkrétním způsobu zabezpečení je nutné nejprve provést předběžnou analýzu, která jednoznačně stanoví:

- Před kým je nutno webovou aplikaci a její data chránit.
- Jaká vysoká bude citlivost dat zpracovaných dat.
- Komu v celém procesu vývoje bude důvěřováno.

V rámci počítačové bezpečnosti existuje škála potenciálních útočníků. Lze vytvořit jejich základní rozdělení podle předpokládané odbornosti, technologického zázemí, finančních zdrojů nebo legislativní podpory [22]:

- Prostý uživatel, který se vědomě pokouší skrze uživatelské rozhraní napadnout systém.
- Kvalifikovaný odborník, s pokročilým vybavením, znalostmi a zdroji.
- Policejní orgány nebo informační služby s téměř neomezenými prostředky a legislativní podporou.

Data jsou tím nejcennějším, co každá webová aplikace obsahuje. Definování citlivosti dat vychází z platné legislativy. V České Republice se dříve jednalo o zákon č. 101/2000 Sb. o ochraně osobních údajů [23]. Od května roku 2018 se pravidla zpracování citlivých údajů

[7] Rozsah hodnot, ze kterých lze generovat náhodnou veličinu.

řídí evropským nařízením GDPR [7]. V rámci úvodní analýzy vývojového procesu je nezbytné posoudit rizika v případě úniku dat. Rizika je nutné definovat jak pro společnost provozující webovou službu, tak pro uživatele aplikace. Při podcenění bezpečnostní politiky internetové služby hrozí společnostem na základě evropského nařízení GRPD až likvidační finanční postihy. Uživatelům, kteří se stanou obětí úniku dat, hrozí odcizení elektronické identity, ztráta kontroly nad využívanými službami a v neposlední řadě odcizení finančních prostředků skrze falešné transakce.

Hlavní chyby uživatelů při využívání internetových služeb:

- Použití veřejně známého hesla.
- Nastavení hesla s nízkou mírou entropie.
- Uložení nešifrované podoby hesla v počítači.
- Zaznamenání hesla na viditelném místě – například lísteček vedle počítače.
- Používání jednoho hesla pro velké množství aplikací.
- Ignorování upozornění poskytovatele služeb na případ narušení bezpečnosti.
- Použití neaktuálního softwarového vybavení.

U uživatelů je nezbytností předpokládat selhání lidského faktoru. Proto je důležité při návrhu webové aplikace naplno využít dostupné možnosti zabezpečení, které moderní technologie nabízí:

- Vynucení šifrovaného spojení.
- Aktivace více faktorové identifikace.
- Ověření uloženého emailu a telefonního čísla uživatele.
- Bohaté využití kryptografických operací při zpracování údajů (viz. dále).
- Respektování ověřených bezpečnostních protokolů a kryptografických algoritmů.
- Ověření identity odesílatele požadavků na server.

Uvnitř návrhu webové aplikace je velice nebezpečné definovat pouze jedinou roli „super-správce" systému. Naopak je potřeba vytvořit kolekci uživatelských rolí, které budou mít ve webové aplikaci různá oprávnění. ASP.NET Core má propracovaný systém přidělování vlastností a rolí klientů v systému. Každému přijatému požadavku jsou přidělena oprávnění již během úvodního procesu zpracování.

Společnost navrhující webovou aplikace musí rozhodnout také o důvěře vůči poskytovatelům služeb třetích stran. Tyto služby jsou pro chod webové aplikace nezbytné. Jedná se například o hostingové služby, použité softwarové řešení třetí strany, síťová úložiště dat, autentizační a autorizační autority nebo databázové systémy. Použít službu třetí strany pro přenesení odpovědnosti za bezpečnostní procesy může být výhodné, pokud vývojáři nemají dostatek zkušeností s implementací bezpečnostní politiky moderní webové aplikace. Na druhou stranu prolomením bezpečnosti identifikační autority třetí strany, například účtu sociální sítě, vede k ovládnutí všech uživatelských účtů, které jsou na tuto bezpečností autoritu napojeny.

Webové platformy, včetně ASP.NET Core, čelí obecně známým typům útoků [4]:

- XSS (Coss-Site-Scripting).
- SQL Injection.
- Session Hijacking.
- CSFR (Cross-Site Request Forgery).

Těmto útokům lze předcházet pomocí mechanismů, které má ASP.NET pro monolitické aplikace již nativně implementováno. Pomoci mohou například validační tokeny pro formulářové požadavky (Anti-Forgery Tokens). Ty ověřují, že data formuláře pochází z původní aplikace. Použitím nástroje Entity Framework, případně voláním již připravených procedur, je možné minimalizovat riziko injektování SQL příkazu do databáze. Validace formulářových dat pomocí nastavení atributů modelů umožňuje předcházet zpracování závadných hodnot ze strany uživatele. Ochranu proti XSS útokům díky injektování škodlivého javascriptového kódu uvnitř formulářových dat má ASP.NET Core zavedeno nativně.

Architektura mikroslužeb má svá vlastní úskalí vyplívající ze své struktury a principů. Problém je například v klientských modulech, které jsou velice často reprezentovány webovou aplikací postavenou na javascriptovém kódu a zabezpečení uživatelova prohlížeče. Pokud prohlížeč obsahuje kritickou chybu nebo uživatel používá jeho zastaralou verzi, nemůže vývojář nic dělat. Při návrhu zabezpečení webových aplikací je proto nutné akceptovat fakt, že část bezpečnosti bude plně závislá na úrovni uživatelova vybavení, jeho soukromém nastavení a gramotnosti v oblasti informačních technologií. Kód aplikace určené pro internetový prohlížeč je plně transparentní, tedy snadněji

napadnutelný [24] [25]. Je možné na straně klienta manipulovat s hodnotami aplikace, kódem nebo s hodnotami cookies. To může vést k narušení bezpečnosti celého systému.

5.1 Zabezpečení komunikace REST modulů

Klientské moduly fungují jako grafické rozhraní mezi uživatelem a zbytkem systému. Jejich hlavním úkolem je tvorba HTTP požadavků a vyžádání si odpovědi na straně serveru. Je tedy celkem snadné klientskou aplikaci i s veškerým implementovaným zabezpečením obejít. Postačí, když útočník sestaví svůj vlastní HTTP požadavek a odešle ho na veřejnou URL adresu serverové služby, která má za úkol přijaté požadavky zpracovat. Může tak obejít ochranu proti XSS nebo jinou validaci hodnot formuláře implementovanou v klientské aplikaci. Serverová REST aplikace na straně serveru pak obdrží škodlivá data nebo kód [26].

V rámci bezpečného fungování webové aplikace na principu mikroslužeb je proto nutné mít kontrolu nad komunikací mezi službami. Je potřeba vědět, kdo daný HTTP požadavek odesílá a ujistit se, že nepochází z nedůvěryhodného zdroje. To však není vůbec snadný úkol. V podstatě celý HTTP protokol je možné podvrhnout. Z pohledu bezpečnosti aplikace není možné obsahu přijatého požadavku bezmezně důvěřovat. Pro zkušenějšího experta není problém s hlavičkami, cookies a tělem HTTP požadavku pracovat v některém z pokročilých programovacích jazyků. Přijaté požadavky pro REST aplikace je proto nezbytné filtrovat a podrobit kontrole.

Vhodné řešení existuje na úrovni síťové komunikace služeb. Například prostředí Microsoft Azure nabízí možnost vytváření vlastní virtuální sítě VPN mezi jednotlivými moduly aplikace. Je možné provádět filtraci na úrovni síťové komunikace a požadavky z jiných zdrojů než od klientských modulů automaticky zahazovat. Toto řešení však vyžaduje vyšší nároky na poskytovatele služeb provozu systému, což většinou i znamená mnohem vyšší finanční náklady. Je nezbytná důvěra v poskytovatele služeb, jeho úroveň zabezpečení. Alternativou pro zabezpečení produkčního prostředí je správa vlastního fyzického serveru. Jde o nákladnou variantu - speciálně pro menší společnosti a vývojové týmy.

Krom produkčního prostředí je vhodné systém chránit i na úrovni samotné webové aplikace. V systému se zavede vnitřní protokol na identifikaci jednotlivých modulů.

V praxi jde o podobnou obranu, jako v případě CSRF (Cross-Site Request Forgery) útoků[8]. Služby musí k požadavkům přidávat identifikovatelný a validní token v souladu s vnitřní bezpečnostní politikou aplikace. Tokeny jsou identifikovatelné pouze na základě privátního klíče. Bezpečnostní politika systému se tak izoluje od venkovního světa. Validace probíhá většinou uvnitř middleware ASP.NET Core aplikace, kde se pro každý požadavek ověří validita tokenu. V případě negativní validace je možné celý požadavek odmítnout.

Částečnou ochranou REST aplikace je implementace nastavení CORS (Cross-Origin Requests). Toto zabezpečení funguje na úrovni moderních prohlížečů. V případě, že chce javascriptová aplikace běžící v internetovém prohlížeči zaslat požadavek na konkrétní serverovou doménu, prohlížeč nejprve zkontroluje, jestli požadavek směřuje na stejnou doménu jako je sama klientská aplikace. Pokud ne, vyšle prohlížeč nejprve na server dotaz pomocí vlastního HTTP požadavku s hlavičkou „Origin“. Tento dotaz obsahuje hodnotu domény dané klientské aplikace. Cílem tohoto dotazu je zjistit, jestli server akceptuje požadavky z domény klienta. Server mu odpoví pomocí HTTP odpovědi obsahující hlavičku „Access-Control-Allow-Origin“ s hodnotami všech domén, které podporuje. Pokud server nepotvrdí prohlížeči akceptování domény klientské aplikace, prohlížeč klientův HTTP požadavek zablokuje. Tato ochrana však funguje pouze na úrovni prohlížeče. Pokud útočník vytvoří požadavek v některém z nižších programovacích jazyků (C#, Java, Python, C++…) pracujících na úrovni operačního systému, server je bezbranný. Přesto je vhodné na straně serveru CORS implementovat jako bezpečnostní standard.

[8] Odesílání falešných hodnot požadavku z jiného zdroje, než je sama aplikace.

5.2 ASP.NET Core - DataProtector

Jde o předdefinovanou interní službu ASP.NET Core vycházející ze staršího DPAPI (Data Protection API). Tuto službu ocení zejména vývojáři, kteří nechtějí trávit velké množství času při implementaci šifrovacích algoritmů a bezpečnostních protokolů uvnitř systému. Interní službu DataProtector vývojář přidá do webové aplikace ASP.NET Core v konfiguračním souboru pomocí volání metody „AddDataProtection“. DataProtector je implementací bezpečnostní politiky izolace interních dat od venkovního světa a symetrického šifrování. Služba provádí kryptografické operace pomocí funkcí volání „Protect“ a „Unprotect“ s definovaným bezpečnostním klíčem [27] [28].

Operace, kterou DataProtector provádí, je kombinací symetrické blokové šifry AES256 v režimu CBC a podepisovacího algoritmu s podporou hashovaní HMAC-SHA512. Vývojář má širokou škálu možných konfigurací této interní služby. Uvnitř konfiguračního souboru může nastavit preferované šifrovací algoritmy, včetně svého vlastního. Dále specifické podepisovací certifikáty, životnost klíčů, oblast systému pro uložení klíčů, bezpečnostní protokoly apod.

Služba DataProtector má vnitřní stromovou strukturu s kořenem v interní službě DataProtectorProvider. Od této služby se vytváří jednotlivé větve nástrojů DataProtector. Každá větev (uzel stromu) má vlastní identifikační klíč zvolený vývojářem aplikace. A každá služba DataProtector může vytvořit dceřiné služby s vlastními klíči.

Na jeden identifikační klíč interní služby DataProtector připadá jedna možná zašifrovaná hodnota. DataProtector hodnoty zašifruje, pak je v rámci aplikace ASP.NET Core a zadaného identifikačního klíče podepíše. Dešifrovat hodnoty může pouze instance služby DataProtector konkrétní webové aplikace s danou stromovou strukturou volání. Samotné zašifrované hodnoty aplikace neukládá. Uloženy jsou pouze identifikátory klíčů použité pro provedené kryptografické operace [29].

Služba skrze instanci aplikace pracuje na úrovni operačního systému serveru. Využívá souborový systém, interní serverové hodnoty nebo registry systému. Tady velice záleží na konkrétním produkčním prostředí aplikace ASP.NET Core.

5.3 Uložení citlivých informací v aplikaci

Jedním z největších bezpečnostních problémů moderních webových aplikací je řešení způsobu uložením citlivých dat a informací v systému. ASP.NET Core nabízí několik možných míst, kde lze data trvale uchovávat.

Citlivá data je možné rozdělit na dvě skupiny:

- Data klientů.
- Data webové aplikace.

Příkladem citlivých dat, která pochází od uživatelů aplikace, jsou uživatelská hesla, loginy, čísla platebních karet, rodná čísla nebo adresy. Tato data jsou chráněna zákonem a musí být zpracována bezpečnostním systémem aplikace, kryptografickými operacemi a protokoly. REST aplikace mají mimo CRUD operací také na starost transformaci dat při zpracování, například hashovaní hesel (PBKDF2 algoritmus) nebo zašifrování čísla kreditní karty (AES nebo RSA).

Interní citlivá data aplikace jsou většinou stavebními kameny bezpečnostní politiky systému. Příkladem jsou přihlašovací údaje k databázím, serverům třetích stran, kryptografické klíče nebo hodnoty „salt“[9] a „IV“[10] pro kryptografické operace.

ASP.NET Core nabízí několik možností uložení dat [15] [4]:

- **Konfigurační soubor (appsettings.json):** Běžně obsahuje hodnoty pro připojení k databázi (ConnectionString) nebo přístupu k SMTP serveru. Je možné do něj uložit libovolné hodnoty. Při nasazení aplikace lze hodnoty uložené v souboru nahradit hodnotami definovanými přímo uvnitř produkčního prostředí. Nehodí se na kryptografická nebo citlivá data [30].
- **Databáze:** Externí systémy pro dlouhodobé uložení dat uživatelů. Databáze není vhodné nadužívat jako „bezpečný trezor“ pro extrémně citlivá data. Bezpečnost celé webové aplikace následně stojí na zabezpečení připojení databáze. Velmi citlivá data je naprosto nezbytné před uložením do databáze šifrovat pomocí vhodného algoritmu.

[9] Náhodně generovaná kryptografická hodnota sloužící jako ochrana hodnot u jednosměrných šifrovacích algoritmů.
[10] Iniciační vektor. Jedná se o náhodně generovaný blok dat u symetrických blokových šifer. Vkládá se do procesu šifrování pro zvýšení nečitelnosti výsledných zašifrovaných hodnot.

- **Azure Key Valut:** Jde o službu cloudového prostředí, kde je možné ukládat interní systémová citlivá data používaná pro kryptografické operace. Tuto službu využívají aplikace ASP.NET Core publikované v produkčním prostředí Microsoft Azure. Citlivá data lze injektovat do webové aplikace dle potřeby.
- **Operační paměť serveru:** Lze použít šifrovanou globální proměnnou pro celou webovou aplikaci. Hodnota se nejprve vygeneruje kryptografickým pseudonáhodným generátorem. Následně je zašifrována, například pomocí interní služby DataProtection. V posledním kroku zašifrovanou hodnotu systém uloží do instance interní služby typu Singleton. V aplikaci pak může být citlivá hodnota kdekoli injektována skrze dešifrovací proces interní služby.

5.4 Autentizační a autorizační procesy

Pro současné webové aplikace je nezbytnost implementace systému pro bezpečné ověření identity svých klientů. Architektura mikroslužeb do tohoto procesu vnesla nové výzvy, jelikož o proces ověření identity příchozího požadavku na server se může starat několik služeb současně.

Proces identifikace uživatele je rozdělen do dvou částí [26] [31]:

- **Autentizace** – Ověření, zdali identita v systému vůbec existuje. Zkoumá se, má-li uživatel k jakémukoli chráněnému obsahu v aplikaci povolený přístup.
- **Autorizace** – Ověření specifických práv autentizovaného uživatele. Framework ASP.NET Core primárně podporuje „Role Based Autorization", tedy přiřazování jednotlivých rolí v aplikaci konkrétním klientům (uživatel, administrátor, editor obsahu apod.).

Autentizační a autorizační procesy probíhají primárně v middleware, která ověřuje identitu každého požadavku. Po ověření identity uživatele middleware přiřazuje potvrzení o autentizaci a případnou roli v aplikaci. Tyto informace se vkládají do instance kontextu HTTP požadavku. Procesy kontroly identity jsou realizovány také v autorizačních filtrech pro jednotlivé controllery a jejich akce. Zkoumá se, jestli má uživatel s danou rolí povolený přístup k určité akci. Vývojáři ASP.NET Core aplikací mají k dispozici celou řadu implementovaných middleware pro ověření identity požadavků.

Tyto metody jsou závislé na způsobu autentizace:

- Authentication Cookie.
- Bearer Token.
- JWT (JSON Web Token).
- Autentizace pomocí služeb třetích stan (Facebook, Twitter, Google).

Popřípadě vývojář může vytvořit vlastní middleware, která daný proces autentizace a autorizace provede manuálně. V takovém případě je nutné implementaci sestavení objektu „ClaimsPrincipal“ v úvodní fázi zpracování uživatelského požadavku.

Postup sestavení „ClaimsPrincipal“ v ASP.NET Core:

1. Systém provede autentizaci a autorizaci uživatele s pomocí příchozích dat a databáze – dohledání uživatele v uložených datech.
2. Vytvoří kolekci objektů typu „Claim“. Tento objekt lze interpretovat jako určitou roli či vlastnost autorizovaného uživatele v systému.
3. Pomocí kolekce těchto objektů vytvoří instanci objektu „ClaimIdentity“. Ten představuje určitou identitu uživatele v systému s definovanými vlastnostmi.
4. Pomocí objektu „ClaimIdentity“ vytvoří instanci objektu „ClaimsPrincipal“. Tento objekt reprezentuje uživatelskou identitu HTTP požadavku. Aplikace dosadí „ClaimsPrincipal“ do uživatelské role v instanci „HttpContext“. Tím je proces identifikace požadavku kompletní. Ostatní elementy webové aplikace mohou s identitou požadavku pracovat, například pro vyhledání uživatele v repositáři.

Není bezpečné implementovat ověřování identity uživatele až do autentizačních filtrů. HTTP požadavek „propadne“ příliš hluboko v procesu svého zpracování. Ostatní middleware, které mohou již předpokládat ověřenou identitu, by pak nemohly vykonávat svoji práci. Filtry lze použít pro nastavení přístupu k určité akci controlleru, pokud identita uživatele splní specifickou kombinaci znaků.

V moderních webových aplikacích existují dva hlavní druhy autentizace:

- Pomocí autentizační cookie.
- Pomocí HTTP tokenu.

První příklad běžný pro většinu monolitických aplikací, kde se jedna instance aplikace stará o veškerou komunikaci a správu uložených dat. Jakmile uživatel projde ověřovacím procesem, informaci o přihlášení uživatele webová aplikace uloží do unikátní session. Ta je tvořena kombinací hodnot o uživateli, které jsou uložené v operační paměti serveru s nastavenou životností. Dále se vytvoří identifikátor patřící k dané session a server ho odešle zpět klientovi. Internetový prohlížeč tuto hodnotu uloží jako autentizační cookie. Její hodnotu klientská aplikace automaticky odesílá s každým následujícím požadavkem. Na základě platnosti identifikátoru, nebo platnosti záznamu v sessions, se aplikace rozhoduje, jestli má uživatel povolený přístup k informacím [4] [31].

Při procesu autentizace a autorizace pomocí HTTP tokenů platí princip, že REST aplikace serveru si nemusí v souvislosti s přihlášením uživatelů nic pamatovat. REST se orientuje na zdroje, které jsou k dispozici každému, kdo při požadavku prokáže svůj nárok je získat. Uživatel se tedy musí identifikovat s každým odeslaným požadavkem. Server pokaždé vyhodnotí jeho identitu, aniž by o přihlášení uchovával informaci. Informace o identitě odesílatele se vkládá do hlavičky HTTP protokolu u každého požadavku, konkrétně do „Authentication Header“. Tato hlavička obsahuje standardizovaný název schématu pro autentizaci a hodnotu tokenu. Jeho podoba závisí na zvoleném schématu autentizace a autorizace [26].

Nejjednodušším schématem pro HTTP autentizaci je „Basic Authentication“. Principem je vkládání zřetězených přihlašovacích údajů ke každému požadavku uvnitř vyhrazeného pole hlavičky HTTP protokolu. Existují varianty, kdy je daná hodnota upravena hashovaní funkcí. Tato metoda je velice citlivá na útok „Man in the middle“, protože uživatelská data jsou odeslána s každým HTTP požadavkem a mají statickou podobu. Použití šifrovaného spojení pomocí SSL je tedy nezbytností.

Lepší variantou je HTTP autentizace na základě standardu OAuth [32]. Klient odešle svá data pouze jednou za účelem ověření identity, server zhodnotí jejich validitu a v případě úspěchu vygeneruje a vrátí klientovi token. Ten v rámci své časové platnosti slouží jako

identifikátor klienta webové aplikace. Token uživatel vkládá ke každému dalšímu požadavku.

Standardizované druhy tokenů [26]:

- SWT (Simple Web Token).
- SAML (Security Assertion Markup Language Tokens).
- JWT (JSON Web Token).

Tokeny jsou většinou složeny z dvou části. První část tvoří veřejné informace o přihlášení, například jméno uživatele, název jeho role, datum přihlášení nebo datum expirace tokenu. Tyto hodnoty mají strukturovanou textovou formu, u JWT například formát JSON. Veřejný obsah tokenu se zřetězí a zakóduje (forma Base64). Druhou částí tokenu tvoří kryptograficky podepsaná první zakódovaná část. Podpis se provádí pomocí vhodného kryptografického algoritmu (HMAC256 nebo RSA) a privátního klíče serveru. Ověření tokenu přijatého s požadavkem na server probíhá principem znovu sestavení kryptograficky podepsané části z přijatých veřejných zakódovaných hodnot. Proběhne jejich podepsání kryptografickým klíčem serveru a následným porovnání s obdrženou podepsanou částí tokenu v požadavku klienta. Tajný klíč serveru je opěrným bodem celé bezpečností politiky autentizace pomocí podepsaných tokenů. Případné prozrazení klíče by znamenalo snadný průnik do aplikace pomocí podvržení falešného tokenu.

Pokud vývojář z určitého důvodu nechce, aby obsah tokenu byl transparentní, je možné tokeny nepodepisovat, ale šifrovat. Používají se algoritmy asymetrického šifrování, například RSA. Klient i server znají své veřejné klíče. Obsah zprávy šifrují pomocí veřejného klíče příjemce, ten ho poté dešifruje svým privátním klíčem.

Algoritmy asymetrického šifrování lze použít také k podpisům tokenů. Odesílatel svým privátním klíčem obsah zašifruje a příjemce ho dešifruje veřejným klíčem odesílatele. Tím se ověří, že obsah pochází skutečně od daného odesílatele.

Procesy ověřování identity nejsou pouze operacemi mezi klientskými a serverovými moduly webových aplikací. V rámci architektury mikroslužeb, kde REST aplikace tvoří jednotlivé služby serveru, je potřeba ověřovat identitu všech požadavků, které si služby odesílají navzájem. Jak bylo uvedeno, REST aplikace nevede žádné záznamy o aktuálně přihlášených klientech. Ověření identity mezi službami slouží k eliminaci rizika přijetí falešného požadavku, který by pocházel mimo vnitřní systém propojených služeb.

5.5 Práce s Cookies

Cookies představují hodnoty serveru uložené na straně klienta a odesílané spolu s HTTP požadavky zpět na server. Tím se liší od sessions nebo globálních proměnných uvnitř webové aplikace, které jsou uloženy na straně serveru. Na základě této specifikace cookies lze použít pro identifikaci známého uživatele nebo pro specifického nastavení aplikace dle konkrétní identity.

Cookies se určitě nehodí k ukládání citlivých informací. Informace v cookies se automaticky odesílají se všemi požadavky a jsou na straně klienta snadno čitelné. Lze je použít pro uložení základní veřejné autentizační informace, tokenu nebo identifikátoru pro danou session. V tomto případě je nezbytné nastavit status cookie jako „http-only“. Server odesílající cookie tímto označením sděluje internetovému prohlížeči, že nikdo krom serveru nesmí hodnotu cookie měnit. To například brání editaci hodnoty skrze kód napsaný v javascriptu a spouštěný v prohlížeči klienta. Jde o jeden ze způsobů ochrany proti XSS útokům [33]. Dále se pro cookies používá parametr „secure“. Tento parametr si vynucuje k odeslání hodnot cookie skrze šifrovaného spojení. ASP.NET Core umí těmito vlastnostmi pracovat v rámci nastavení cookies v controlleru nebo kontextu HTTP požadavku.

5.6 ASP.NET Identity

Jedná se o komplexní, flexibilní a rozšiřitelné API v rámci .NET platformy. Slouží k jednoduchému definování standardním autentizačním a autorizačním procesům. Strukturálně má silnou závislost na ORM nástroji Entity Framework. Obsahuje konkrétní modely pro vytvoření potřebného entitně-relačního modelu uživatelů, rolí nebo oprávnění. Využívá upravený databázový kontext (IdentityContext). Základní entity a relace jsou definovány již v rodičovské třídě kontextu. Jelikož Entity Framework nepodporuje použití rozhraní, dosahuje se abstrakce databázového kontextu skrze

systém genericity objektů. Vývojář má možnost rozšiřovat například parametry uživatelova účtu.

ASP.NET Identity podporuje různé druhy autentizace [31]:

- Autentizační cookies.
- Bearer Token.
- Basic Authentication.
- JSW Token.
- Autentizaci přes třetí stranu.

Implementace do projektu je možná hned při tvorbě základní šablony projektu, kdy je ASP.NET Identity automaticky implementováno v plném rozsahu. ASP.NET Identity lze implementovat také manuálně v pozdější fázi vývoje projektu.

Konfigurace ASP.NET Identity se provádí v konfiguračním souboru webové aplikace pomocí nastavení použitím definovaných interních služeb. ASP.NET Identity implementuje do projektu základní bezpečnostní operace:

- Registraci uživatele.
- Přihlášení a odhlášení.
- Změnu hesla.
- Přehlášení přes třetí stranu.
- Ověření role (autorizaci).
- Více-faktorová autentizace.
- Ověření emailu.

Pro konkrétní operace s profilem uživatele se používá instance třídy UserManager. Webová aplikace využívající ASP.NET Identity používá zmíněný speciální databázový kontext jak pro údaje o autentizačních a autorizačních procesech, tak jako klasický kontext databáze pro běžná data. Jedná se pouze o upravený potomek třídy klasického databázového kontextu. V rámci optimalizace a bezpečnosti je však vhodné ponechat databázový kontext ASP.NET Identity pouze pro citlivá data a pro zbytek dat webové aplikace vytvořit kontext nový. Systém pak používá více než jednu databázi.

6 Spojení ASP.NET Core s architekturou mikrolužeb

V rámci plnění cílů této práce byla popsána architektura aplikací mikroslužeb a také framework pro tvorbu webových aplikací ASP.NET Core. Nyní je cílem na základě zpracovaných informací zhodnotit možnosti použití ASP.NET Core pro tvorbu systémů postavených právě na architektuře mikroslužeb.

Relevantní otázky k této problematice:

- Je možné v ASP.NET Core vytvořit webovou aplikaci, která plní požadavky na mikroslužbu?
- Jaké vlastnosti tato služba může v daném frameworku mít?
- Existují vhodné nástroje k propojení služeb do kooperujícího systému?
- Jaké jsou vývojové služby, nástroje a prostředí podporující ASP.NET Core, které lze pro návrh systému mikroslužeb použít?
- Jaké nevýhody má ASP.NET Core při použití v architektuře mikroslužeb?

Alternativu pro architekturu mikroslužeb ve světě .NET jsou webové služby vytvořené pomocí frameworku WCF (Windows Communication Foundation). Ten se orientuje na princip SOA. Slouží k vývoji hostovaných serverových služeb odesílajících data formou zpráv mezi koncovými body. Data mezi definovanými koncovými body jsou přenášena ve formátu XML nebo jako datový proud. WCF podporuje velké množství internetových protokolů, včetně HTTP nebo TCP. Vývoj v tomto frameworku charakterizuje podrobná konfigurace abstraktních kontraktů služeb, definování klientů a struktury dat. Na druhou stranu vývoj rozsáhlého systému obnáší velké množství práce a nepřehlednost. Díky nutnosti detailní konfigurace zabere vývojový proces velké množství času a je náchylný k chybám [34].

V případě ASP.NET Core je použit princip nezávislé webové aplikace poskytující zdroje dat. Každá taková aplikace na principu REST je jednotkou architektury mikroslužeb. Má svojí specifickou roli, odpovědnost a je plně samostatná. Logická role REST aplikace v návrhu systému je dána úvodní business analýzou požadavků společnosti. Množina samostatných REST aplikací vytvořených v ASP.NET Core, které dohromady spolupracují, tvoří systém architektury mikroslužeb.

Podmínkou k nezávislosti služeb je vytvoření univerzálního komunikačního rozhraní mezi jednotlivými službami – společný komunikační protokol a struktura dat. ASP.NET Core má v sobě zabudovaný komplexní proces analýzy HTTP protokolu. Přijaté HTTP požadavky jsou automaticky převedeny na instanci kontextu požadavku „HttpContext". Je velmi snadné zpracovávat příchozí požadavky od jiných služeb v plném rozsahu (HTTP hlavičky, cookies nebo data v těle požadavku). Po provedení příslušné operace v koncovém bodě REST aplikace je jednoduchým způsobem deklarována příslušná odpověď na přijatý požadavek. Výsledek zpracování je automaticky přidán do kontextu požadavku a vrácen zpět klientovi formou HTTP odpovědi. Datové modely jsou v ASP.NET Core automaticky překládány do strukturované podoby a připojovány k odpovědi na požadavek. Výchozím formátem strukturovaných dat je JSON. Je možné nastavit překlad do formátu XML nebo vlastního formátu s použitím vlastního překladače dat. Komunikace s klienty a mezi službami systému je tak u ASP.NET Core velmi snadná a praktická.

Díky abstraktní vrstvě protokolu mezi službami je celý systém snadno škálovatelný a editovatelný bez ohledu na technologie. Přidání nebo nahrazení konkrétních služeb není složitou operací, která by představovala riziko pro systém jako celek. Nikde neexistuje závislost ASP.NET Core mikroslužby na okolních technologiích systému. Služby mohou kooperovat a přitom být vytvořeny v úplně jiném programovacím jazyce nebo být založeny na odlišném technologickém prostředí. V tomto ohledu ASP.NET Core plní požadavek na schopnost nezávislosti technologií. ASP.NET Core se zbavuje striktní závislosti na platformě .NET a umožňuje možnost vývoje také pro platformu Linux. Je to možné díky technologicky nezávislému jádru a moderním nástrojům pro tvorbu webových kontejnerů. Odpadá tedy závislost frameworku na produkčním prostředí pro publikování hotového systému.

V ASP.NET Core lze tvořit jak datové, tak dispečerské služby. Datové služby většinou reprezentuje konkrétní REST aplikace s přímým spojením k datovému úložišti, často entitně-relační databázi. Může se jednat o přímé propojení pomocí ADO.NET nebo skrze ORM prostředníka. V ASP.NET Core jde většinou o ORM nástroj Entity Framework. Vzhledem k vlastnostem Entity Framework je vhodné, aby jednu databázi obsluhovala pouze jedna mikroslužba – tedy implementace návrhového vzoru „Database per Service". Ta může fungovat jako zdroj dat pro ostatní služby v systému. Pokud by více

mikroslužeb přímo obsluhovalo jedno datové úložiště, mohlo by dojít ke konfliktům při tvorbě a úpravách entitně-relačního modelu, speciálně při použití nástroje Entity Framework a jeho politikou ověřování entitně-relačního modelu. Dispečerská služba je taková REST aplikace v systému, která nemá přímý přístup k databázi, ani přímo nekomunikuje s klientem. Od datových služeb sbírá informace vyžádané přijatým požadavkem a odesílá je zpět službám, které mají za úkol tyto informace předat klientovi. Stejně tak může dispečerská mikroslužba přijatá data od klienta rozdělit vícero datovým službám, které se starají o jejich uložení v databázi. Dispečerské služby většinou v ASP.NET Core používají instanci klienta HTTP protokolu, který se v abstraktní vrstvě tváří jako zdroj dat. Tyto služby odesílají požadavky mikroslužbě napojené na databázi systému. Pro službu, která komunikuje s klientem, se dispečerská služba jeví jako abstraktní datový zdroj. Reálné úložiště dat je tak izolováno od většiny systému, ačkoli k datům mají služby přístup skrze abstraktní jeho vrstvy.

Velkým tématem architektury mikroslužeb je zabezpečení. V tomto ohledu je nevýhodou ASP.NET Core vazba frameworku na předdefinovaná řešení, například ASP.NET Identity. Tato řešení jsou často silně propojena se specifickými technologiemi, například Entity Framework nebo jednu specifickou konfiguraci webové aplikace. Architektura mikroslužeb si vyžaduje flexibilitu. Při použití ASP.NET Core je proto nutné hledat unifikovaná řešení náročná na testovací procesy.

ASP.NET Core má bohaté možnosti pomocných nástrojů pro usnadnění procesu vývoje komplexních webových systémů. Pro vývoj aplikací se používá Visual Studio IDE, které díky bohatým možnostem práce s projekty umožňuje snadný vývoj komplexních a složitých systému postavených na architektuře mikroslužeb. V rámci jednoho vývojového řešení umožňuje zpracování více paralelních REST aplikací. Nástroj Visual Studio IDE podporuje paralelní konfiguraci, kompilaci, spouštění a testování. Má nativní podporu virtualizačního nástroje Docker, který bude blíže popsán dále. Visual Studio IDE nabízí synchronizaci s nástrojem MS TFS pro správu projektů a zdrojového kódu. Ten umožňuje organizaci práce na jednotlivých sprintech[11], úpravu verzí zdrojového kódu u všech paralelních aplikací, testování a publikaci hotového systému.

[11] Časový úsek, kdy společnost pracuje na vývoji nové funkcionality systému.

Pro publikování hotového systému vytvořeného v ASP.NET Core slouží především platforma Microsoft Azure. Jde o velice rozsáhlou technologickou platformu na principu cloud computingu. Obsahuje širokou nabídku služeb pro chod a správu hotového systému:

- Distribuce běžících aplikací.
- Datová úložiště a jejich správa.
- Prostor pro klientské moduly.
- Active Directory pro správu uživatelských účtů aplikace.
- Key Valut pro bezpečné uchování citlivých dat systému (šifrovací klíče apod.).
- Možnost vytvoření virtuální sítě (VPN) pro síťovou ochranu jednotlivých služeb v systému.
- Přímou podporu kontejnerových virtualizačních služeb, například Docker.

Microsoft Azure také nabízí službu virtuálního počítače pro unifikovaná řešení komplexních systémů. Jelikož ASP.NET Core není závislý pouze na tradičním prostředí platformy .NET, ale je kompatibilní s prostředím operačního systému Linux, je možné služby vytvořené pomocí ASP.NET Core distribuovat na obdobných clusterech mimo platformu .NET, například na Amazon Services. Je možné konstatovat, že ASP.NET Core splňuje požadavky pro adaptivní nasazení hotového systému na principu architektury mikroslužeb bez ohledu na technologie prostředí.

6.1 Virzualizační nástroj Docker

Docker je open-source projekt sloužící primárně k možnosti nasazení více kooperujících aplikací jako samostatného a kompatibilního systému. Principem je tvorba „kontejnerů“, což v praxi zjednodušeně znamená vytvoření velmi silné abstraktní aplikační vrstvy kolem výsledné aplikace a umožnění její spolupráce s ostatními aplikacemi umístěných v jiných kontejnerech v systému [1] [5].

Kontejner je definován jako běžící instance aplikace, samotný systém je pak reprezentovaný skrze virtuální kontejnerové obrazy (container image). Tyto obrazy jsou řazeny do Docker registru. Nástroj je univerzální pro všechna přední prostřední a operační systémy.

Tento nástroj je velice vhodný pro tvorbu aplikací na principu mikroslužeb, protože je schopen vytvořit virtuální obraz celého systému včetně případných závislostí. Docker je schopen reagovat na události v systému, není třeba řešit pouze konkrétní běžící aplikace. Samotný virtuální obraz systému je díky vysoké abstrakci kompatibilní se jakýmkoli prostředím, které Docker podporuje.

Pro podporu v operačním systému Windows je potřeba podpora Hyper-V virtualizace. V praxi to znamená využití profesionální verze operačního systému Windows a procesoru s podporou SLAT (Second Level Address Translation). Druhou možností je použít softwarový virtualizační nástroj pro vytvoření běžící instance operačního systému Linux a v této instanci spustit nástroj Docker ve verzi pro operační systém Linux.

Prostředí pro vývoj ASP.NET Core aplikací (Visual Studio IDE) nabízí nativní kompatibilitu nástroje Docker. Podporu Dockeru je možné nastavit přímo při tvorbě nového projektu, nebo i zpětně manuálním nastavením. Prostředí pro publikaci aplikací Microsoft Azure umožňuje nasazení aplikace postavené na kontejnerech vytvořených v nástroji Docker.

7 Praktická část práce – ukázková aplikace

Praktická část práce má za cíl na jednoduché webové aplikaci demonstrovat vývoj pomocí ASP.NET Core s použitím architektury na principu mikroslužeb. Demonstrovaná webová aplikace se inspiruje principem „záložek“ od společnosti Facebook a ukládáním oblíbených internetových stránek pomocí webových prohlížečů. Umožňuje uživateli ukládat své oblíbené internetové články. Dále umožňuje jednoduché sdílení uložených článků s ostatními uživateli systému, kteří mají zájem o podobný obsah.

Cílem návrhu aplikace je vytvořit funkční skupinu mikroslužeb s prezentační vrstvou. Tento systém musí tvořit jeden komplexní celek pro koncového uživatele. Jednotlivé služby serveru musí být na sobě vzájemně nezávislé, aby se prováděnými změnami během vývoje navzájem ovlivňovali pouze v nutné míře. Tak vznikne modulární a škálovatelný návrh architektury.

Aplikace se skládá ze čtyř modulů:

- Dynamická aplikace pro zobrazení prezentační vrstvy.
- Služba správy uložených článků.
- Služba pro logování událostí.
- Služba analyzátoru webového obsahu.

Serverové moduly jsou založeny na frameworku ASP.NET Core 2.0. Využívají knihovny umožňující multiplatformní nasazení aplikace. Jako datové úložiště slouží databázový systém MSSQL. Pro objektově relační mapování a komunikaci s databází byl použit nástroj Entity Framework Core 2.0. Prezentační vrstva používá dynamický webový framework pro tvorbu SPA[12] (Single Page Aplication) založený na programovacím jazyce Javascript.

[12] Webová aplikace je tvořena pouze jedinou HTML stránkou. Její obsah se dynamicky vykresluje na základě aktivity uživatele.

7.1 Serverový cluster

Serverová část systému je složena z REST aplikací vytvořených pomocí ASP.NET Core. Pro vzájemnou komunikaci využívají HTTP protokol. Bylo dosaženo vysoké míry abstrakce uvnitř systému a nízkých časových nároku na zpracování příchozích požadavků. Vnitřní architektura služeb je postavena na návrhovém vzoru MVC, ačkoli místo pohledů jsou výstupem serverových služeb strukturovaná data pro klientský modul nebo okolní služby.

Entitně-relační model je navržen na základě strategie mapování „Code First“ skrze FluentAPI. V třídách modelů jsou pomocí atributů definovány pouze primární klíče výsledných databázových tabulek. Z hlediska architektury služeb je použit návrhový vzor „Database per Service“. Žádná serverová služba nepracuje s více než jednou databází.

Aplikace serveru se skládá z následujících domén:

- **AppUser** – Definuje uživatele aplikace.
- **UserSecret** – Definuje citlivou část uživatelova profilu, zejména data pro autentizační a autorizační procesy.
- **Website** – Reprezentuje záznam o jedné konkrétní webové stránce uložené v systému. Obsahuje základní informace, například titulek, hlavní nadpis, adresu hlavního obrázku nebo hlavní text stránky.
- **WebPortal** – Portál sdružující konkrétní uložené stránky pod určitou internetovou doménou. Obsahuje název a doménu portálu, jeho favicon a relační propojení s přidruženými stránkami.
- **SavedWebsite** – Představuje webovou stránku obsaženou v databázi systému a uloženou některým uživatelem pro další použití v aplikaci.
- **Channel** – Každý uživatel má při registraci automaticky vytvořen svůj kanál, kde se zobrazují publikované stánky ostatním uživatelům.
- **Post** – Základní jednotka funkcionality sdílení uložených stránek mezi uživateli systému.
- **Subscription** – Jde o propojení mezi uživatelem a odebíraným kanálem se sdílenými příspěvky. Propojují se konkrétní uživatelé a kanály jiných uživatelů.

Jelikož samotná serverová aplikace nevrací jako odpověď na požadavek celou vygenerovanou HTML stránku, ale pouhá strukturovaná data, převádí se datové modely

na strukturovaný text ve formátu JSON. Objektový model webové aplikace je vždy převeden na primitiva a jednoduché kolekce. S těmi integrovaný překladač umí efektivně pracovat a prezentační vrstva systému nemá problém takto strukturované data zpracovat.

Vytvořená aplikace obsahuje testovací datový generátor. Ten naplní databázi testovacími daty, pokud je prázdná. Tento stav nastane například při odeslání nové migrace entitě-relačního modelu. Generátor je podmíněně volán pomocí middleware. Podmínkou je konfigurace aplikace do testovacího režimu. Starší verze Entity Framework obsahovaly svoji verzi plnícího generátoru, který databázi obnovoval do testovacího stavu na přání vývojáře. V Entity Framework Core je nutné generátor implementovat manuálně.

Pro komunikaci mezi controllery aplikace a reprezentací databáze je aplikován vzor „Repository Pattern“. Pro každou logickou oblast aplikace (autentizační a autorizační procesy, práce s uživatelskými daty, práce s uživatelským profilem apod.) existuje vlastní vnitřní služba repositáře. Tyto repositáře jsou typu interní služby ASP.NET Core typu Transient. Jsou pomocí DI Containeru injektovány do jednotlivých controllerů. Interní služba Transient byla zvolena podle principu Thread-Safety[13], protože pro jeden přijatý požadavek od klienta se vytváří jedna instance interní služby. Nehrozí tak přehlcení vnitřní služby při zvýšené zátěži webové aplikace. Principem služeb repositářů je získat data z databáze pro určitou logickou operaci v systému a následně je transformovat do podoby, se kterou může pracovat controller. Ten sám o sobě obsahuje jenom minimum aplikační logiky a stará o vytváření odpovědí na základě výsledků z repositáře.

[13] Koncept používaný u aplikací, které využívají více vláken procesu. Každé vlákno přistupuje k prostředkům sdílených s ostatními vlákny tak, aby neovlivňovalo činnost ostatních vláken.

7.1.1 Autentizace a autorizace

Pro autentizační a autorizační procesy byl v aplikace vytvořen vlastní jednoduchý systém na principu HTTP tokenů. Tělo tokenů obsahuje následující prvky:

- Datum vytvoření.
- Datum zániku tokenu.
- Identifikátor uživatele.
- Doménu klientské aplikace.

Jednotlivé prvky jsou zřetězeny a odděleny oddělovacími znaky, následně zakódovány do formátu Base64. Tato část tvoří první polovinu tokenu. Druhá polovina je tvořena první polovinou, která je kryptograficky podepsána a jednosměrně zašifrována pomocí HMAC-SHA512 s využitím privátního klíče serveru. Obě půlky se nakonec spojí pomocí oddělovacího znaku do jednoho řetězce a slouží jako identifikační token pro klientskou aplikaci. Ta ji přidává ke všem požadavkům vyžadujícím uživatelsky ověřený přístup k serveru. Díky náhodným prvkům uvnitř tokenu a vlastnostem algoritmu HMAC-SHA512 by měl být každý token jedinečný, není nutné řešit problém s krádeží tokenu mimo vymezený rámec jeho platnosti.

Z uvedeného postupu plyne, že klíčem zabezpečení je privátní klíč serveru. Ten aplikace kryptograficky generuje pomocí pseudonáhodného generátoru kryptografické knihovny programovacího jazyka C#. Klíč je generován při sestavování aplikace. Lze ho na vyžádání změnit. Díky použitému generátoru je hodnota klíče statisticky nepředvídatelná a zcela náhodná. Hodnota klíče je uložena v interní službě typu Singleton. Před uložením do paměti je zašifrována pomocí ochranné aplikační služby „DataProtector“. Při injektování hodnoty do konkrétního místa aplikace se hodnota pomocí „DataProtector“ nejprve dešifruje a následně injektuje interní službou Singleton. Hodnota klíče není nikde uložena v nechráněné podobě. Nemělo by dojít k ohrožení bezpečnosti aplikace ani skrze analýzu operační paměti serveru. Pokud by hodnota klíče byla odhalena třetí stranou, mohlo by dojít ke kompromitaci aplikace pomocí falešných tokenů.

Ověření autorizovaného požadavku probíhá uvnitř deklarované middleware, která přistupuje k instanci HTTP požadavku. Z hlavičky protokolu určené pro autorizaci načte hodnotu tokenu klientské aplikace. Nejprve middleware ověří platnost tokenu tím, že se pokusí znovu podepsat první polovinu za pomocí serverového klíče. Pokud se podepsaná

část shoduje s druhou půlkou tokenu, znamená to, že token byl vytvořen službou serveru. Následně middleware dekóduje jeho první polovinu do podoby strukturovaných dat. Z parametrů tokenu zkontroluje, jestli není za hranou platnosti. Pokud je vše v pořádku, spojí se middleware s databází a vyhledá podle identifikátoru z tokenu konkrétního uživatele. S pomocí dat o uživateli middleware sestaví objekt „ClaimPrincipal“. Tento objekt reprezentuje v ASP.NET Core instanci identity uživatele webové aplikace. Uživateli jsou v systému přiděleny konkrétní vlastnosti (role v systému, oprávnění atd.). Objekt „ClaimPrincipal“ je pomocí middleware přidán do kontextu HTTP požadavku. Tím je proces autentizace a autorizace uživatele dokončen.

7.1.2 Analýza webových stránek

Jak již bylo uvedeno, smyslem aplikace je ukládání, analýza a následná filtrace oblíbených webových článků uživatele. Musí se nutně analyzovat příchozí odkazy a z obsahu stránky získat relevantní informace. Ty se následně použijí například pro filtraci nebo výslednou prezentaci v klientské části. Aplikace stahuje zdrojový HTML kód a analyzuje:

- Titulek stránky.
- Hlavní nadpis.
- Hlavní odstavec.
- Hlavní obrázek.
- Kódování stránky.
- Popis stránky.
- Portál, na kterém je webová stránka umístěna.
- URL adresa stránky.

Tyto informace jsou nezbytné k plnění funkcionality aplikace. Webové stánky nejsou striktně normované a jedna od druhé se značně na HTML výstupu liší.

Hlavními problémy, které se při analýze vyskytují, jsou například:

- Absence některého z očekávaných elementů.
- Stará verze HTML.
- Netradiční používání tagů a atributů.
- Ověřování adres zdrojů.
- Nepřístupný web.

Pro samotnou analýzu je použita knihovna třetí strany „HtmlAgilityPack“, která převádí řetězec HTML kódu do formátu kolekce elementů s možností LINQ dotazování. Rovněž umožňuje měnit kódování obsahu zdroje dat. Tato knihovna zrychluje filtračních procesy a snižuje výkonnostní nároky na server. Právě nároky na server jsou jedním z aspektů, které je potřeba brát v úvahu při návrhu konečného řešení. Dalšími jsou náklady na úložiště, aktualitu a relevanci dat.

Aby se snížily nároky na úložiště dat, jsou entitně-relační vztahy nastaveny tak, aby nebyla v databázi žádná konkrétní stránka uložena vícekrát. Pokud si dva uživatelé uloží stejný článek, informace o něm jsou uloženy pouze jednou. Mezi články a uživateli se vytvoří relace informující o uložení. Stejně tak u webových portálů existuje strategie k minimalizaci záznamů. Portálem se rozumí mateřský web, který obsahuje více článků. Při ukládání nového článku webová aplikace zkoumá, jestli portál tohoto článku již v databázi existuje. Pokud ano, pouze se vytvoří nové propojení a nic dalšího se neukládá. Pokud se uživatel rozhodne uloženou stránku smazat, nesmažou se uložené údaje o stránce, ale pouze spojení mezi uživatelem a uloženou stránkou v databázi. To vede k možné situaci, že uložená stránka nemusí být propojená s žádným uživatelem. Tato stránka je později v rámci údržby databázových dat smazána. Stejně se při údržbě nakládá s uloženým portálem, který již není propojen s žádnou stránkou, kterou by obsahoval. Údržba je automatická v rámci speciálního repositáře webové aplikace.

V rámci ukládání informací o adresách obrázků probíhá kontrola, jestli v dané adrese není injektován javascriptový kód nebo odkaz nesměřuje na soubor typu .JS. Na straně klienta se obsah zdroje načítá do prohlížeče a mohl by potenciálně injektovat škodlivý kód.

7.2 Klientský webový modul

Klientský modul tvoří samostatná webová aplikace naprogramovaná v jazyce Javascript. Pro vykreslení výsledné stránky je použit frameworku ReactJS. Aplikace je tvořena dílčími komponentami uživatelského rozhraní. Ty se v reakci na aktivitu klienta sestaví do podoby aktuální webové stránky prezentační vrstvy. Komponenty aplikace mají stromovou strukturu. Kořenem tohoto stromu je komponenta „App" reprezentující celkovou aplikaci.

Pro zajištění požadované funkcionality byla do aplikace přidána další externí rozšíření:

- **React-Router V4 –** Knihovna, která nastavuje směrování a navigaci skrze adresu URL. Umožňuje pracovat s historií stránek, vkládat dynamické odkazy na jiné komponenty, nebo přiřazovat URL adresám konkrétní komponenty aplikace.
- **Axios** – Nástroj k asynchronnímu provádění HTTP požadavků a zpracování získané odpovědi ze strany serveru.
- **React-Bootstrap –** Grafický framework pro responzivní design webových stránek. Webová stránka je rozdělena na sekce a dynamicky reaguje na změnu rozlišení okna prohlížeče. Framework obsahuje základní komponenty pro uživatelské standardní rozhraní webové aplikace.

Aplikace funguje nezávisle na serveru s daty. Je tedy plně nezávislá na zvolené architektuře serverového modulu. Jako univerzální rozhraní slouží HTTP protokol obsahující strukturovaná data ve formátu JSON. Javascript umí s tímto formátem pracovat bez nutnosti dalšího formátování.

7.3 Možnosti dalšího rozvoje vytvořené aplikace

Systém dynamické webové aplikace určené pro prohlížeč a serverového systému postaveného na principu mikroslužeb se podařilo vyvinout do testovací verze s implementovanou funkcionalitou popsanou v cílech praktické části práce. Vytvořený systém za cíl především demonstrovat možnosti použití nezávislých aplikací pro tvorbu rozsáhlých informačních systémů.

Dalším rozvojem aplikace by mohlo dojít k vytvoření plnohodnotné aplikace určené ke komerční publikaci ve webovém prostředí. Je zapotřebí přidat další klientské služby, například mobilní aplikaci. Dále je nutné vylepšit a zoptimalizovat metody analýzy HTML obsahu článků, s použitím funkcionálního programovacího jazyka pro definování lepších a přehlednějších analytických funkcí.

V internetovém prostředí existuje množství podobných aplikací. Ty jsou definovány kategorií „Read Later“, tedy možností si určitý obsah internetového světa uložit k přečtení na později. Nemají však většinou možnosti sdílení mezi uživateli a potenciál vytvoření obsahově-sociální sítě. Je tedy možné zvážit další vývoj aplikace v tomto směru a umožnit spojení relevantního obsahu vztahujícího se k určitému tématu s vlastnostmi sociálních sítí. V případě dalšího rozvoje projektu by bylo možné aplikaci orientovat na propojení sítě uživatelů s relevantním obsahem internetových článků. Aplikace by kladla důraz na aktualitu a vhodné zaměření obsahu článků vůči uživatelům sítě. K tomuto výsledku by bylo možné dospět využitím základních principů sociálních sítí. Bylo by možné články filtrovat díky hodnocení samotných uživatelů, extrahovat z nich relevantní obsah pro konkrétní skupiny uživatelů a doporučit vhodné články novým členům aplikace. Pro dosažení tohoto cílového stavu je však zapotřebí rozsáhlý vývoj systému.

8 Závěry a doporučení

V rámci práce došlo ke zpracování získaných informací o vlastnostech moderních architektur webových aplikací. Lze reálně volit mezi architekturou monolitickou a architekturou založenou na mikroslužbách. V práci byly prozkoumány a popsány výhody i nevýhody obou přístupů k vývoji webových aplikací.

Monolitické webové aplikace fungují na principu zahrnutí všech poskytovaných služeb, veškerého řízení a obsluhy systému do jedné komplexní aplikace. Jsou co do vstupních nákladů (finanční, časové, lidské zdroje) méně náročné. Z hlediska dlouhodobého růstu systému však monolitické aplikace mohou představovat vážný problém. Mají pouze omezenou hranici své efektivity ve vztahu ke své velikosti. Velikostí je myšlen objem zpracovaných dat a množství pravidelných požadavků od uživatelů. Monolitická architektura nabízí pouze omezenou škálu použitelných technologií. Proto jsou tyto webové aplikace velice citlivé na incidenty, katastrofy a rozsáhlé editace. Nemají vysoké nároky na finální prostředí pro publikaci. Umožňují komplexní a méně složité zabezpečení systému. Lze v nich využívat otestované externí knihovny, nástroje a metodiky určené přímo pro konkrétní monolitickou architekturu.

Architektura mikroslužeb je naopak princip komplexní spolupráce vícero malých webových aplikací. Stojí na principech decentralizace systému, abstrakce a efektivní vzájemné spolupráce. Tato architektura je náročná na vstupní náklady a úvodní analýzu požadavků. Z hlediska dlouhodobého rozvoje jsou systémy založené na architektuře mikroslužeb velmi snadno škálovatelné a editovatelné. Minimalizuje se tak riziko nutnosti velkých komplexních oprav celého systému. Hledání chyb webové aplikace je u této architektury velmi snadné na úrovni jednotlivých služeb. Naopak testování systému jako celku může být náročné. V případě katastrofy jsou tyto aplikace odolným řešením umožňující minimalizaci škod. Architektura umožňuje oddělenou správu, údržbu a vývoj systému. Dává prostor pro spolupráci a paralelní zapojení vícero technologií v rámci jednoho informačního systému. Pro nasazení webové aplikace je potřeba počítat s komplexním prostředí a rozsáhlými nároky na konfiguraci systému.

Dále se práce věnovala modenímu webovému frameworku ASP.NET Core od společnosti Microsoft Corporation. Na základě uvedených zdrojů se ukázalo, že splňuje požadavky pro vývoj moderních webových aplikací. Je serverově orientovaný, umožňuje návrhový vzor

MVC pro přehlednou vnitřní stavbu webové aplikace. Tento framework technologicky sjednocuje starší webové frameworky do jedné univerzální platformy „webové aplikace". ASP.NET Core umožňuje flexibilní vývoj jak komplexních monolitických aplikací, tak samostatných REST služeb určených pro architekturu mikroslužeb. Podporuje pokročilé technologie pro práci s protokolem HTTP, tvorbu prezentační vrstvy i nativní podporu nástrojů pro správu uložených dat. ASP.NET Core umožňuje multiplatformní vývoj pro různá produkční prostředí. Nabízí pokročilé nástroje pro zabezpečení webových aplikací před známými druhy útoků. V tomto ohledu je možné implementovat klasická komplexní řešení, nebo individuálním návrhy podle potřeb systému. ASP.NET Core nabízí velice snadnou konfiguraci interních služeb dané webové aplikace. Konfigurace je jednotná nehledě na druh aplikace. ASP.NET Core nativně implementuje nástroje pro tvorbu webových kontejnerů, což usnadňuje práci a vývoj webových aplikací postavených na členitých architekturách. Nativní podpora mnoha dalších pomocných nástrojů umožňuje urychlení a zpřehlednění vývojového procesu a údržby systému.

Součástí práce bylo shrnutí principů zabezpečení moderních webových aplikací. Tyto principy platí jak pro monolitické aplikace, tak pro architekturu mikroslužeb. Byly popsány a definovány aktuální problémy webové bezpečnosti pramenící z konfliktu mezi uživatelským komfortem a efektivní mírou zabezpečení. Došlo k vysvětlení principu dvou hlavních způsobů ověření identity uživatele za pomocí autentizační cookie, nebo pomocí tokenu v hlavičce HTTP protokolu. Byla popsána speciální úskalí architektury mikroslužeb pro zabezpečení komunikace mezi jednotlivými službami. Je možné konstatovat, že moderní webové aplikace se neobejdou bez vynucení šifrovaného spojení, standardizovaných bezpečnostních protokolů, aktivního využití kryptografických algoritmů a důkladného prověření příchozích dat na server.

V rámci praktické části práce se podařilo vytvořit prototyp webové aplikace vytvořené pomocí frameworku ASP.NET Core 2.0 a dynamické prezentační vrstvy. Je založena na jednoduché reprezentativní architektuře mirkoslužeb. Obsahuje kolekci na sobě nezávislých kooperujících REST aplikací a prezentační vrstvu dynamické SPA. Současná verze vytvořené aplikace implementuje principy definované v teoretické části práce. Pro veřejné publikování je nutný další kontinuální vývoj a testování, podle principů popsaných v práci pro architekturu mikroslužeb.

Frameworky typu ASP.NET Core spolu s moderními komplexními architekturami jsou vhodnými stavebními kameny moderních webových aplikací. Tyto technologie jsou páteří dnešního virtuálního světa. Ani ta nejchytřejší aplikace postavená na libovolné technologii by neměla hlubší význam, kdyby neexistovaly flexibilní způsoby, jak obsloužit uživatele na globální úrovni. Stejně tak je již paradigma, že webová aplikace je pouze kolekcí sémanticky strukturovaných informací, přežitkem. Moderní internetové informační systémy mají mnohem vyšší nároky na aplikační logiku, schopnost analytiky a dynamický uživatelský komfort. Uplatňuje se u nich kontinuální vývoj založený na dlouhodobé obchodní vizi moderních společností. Vždy je možné v rámci daného systému expandovat a rozšířit poskytované služby do nových sfér. Moderní webové platformy umožňují spolupráci různorodých technologií. Díky tomu mohou poskytovatelé služeb nabídnout svým klientům podporu širokého spektra zařízení a zajistit vyšší uživatelský komfort.

Svět webových technologií je zdrojem moderní zábavy, ale také cestou k vysněné seberealizaci. Nezbývá než věřit, že komerční společnosti společně s politickými institucemi budou vnímat potřebu společnosti po rozmanitém inteligentním virtuálním světě. V takovém prostředí budou moci lidé efektivně vykonávat své profese a svobodně přistupovat k odborným znalostem a vzdělání.

9 Citované zdroje a literatura

1. **Smith, Steve.** *Architecting Modern Web Applications with ASP.NET Core and Azure.* Washington : Microsoft Corporation, 2017.

2. **Newman, Sam.** *Building Microservices.* Sebastopol : O'Reilly Media, Inc, 2015. ISBN: 978-1-491-95035-7.

3. **Bonér, Jonas.** *Reactive Microservices.* Sebastopol : O'Reilly Media, 2016. ISBN: 978-1-491-95779-0.

4. **Freeman, Adam.** *Pro ASP.NET Core MVC 2.* London : APress, 2017. ISBN-13 (electronic): 978-1-4842-3150-0.

5. **Torre, Cesar.** .NET Microservices: Architecture for Containerized .NET Applications. [Online] 2017. [Citace: 3. 22 2018.] https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/.

6. **Richardson, Chris.** Pattern: Microservice Architecture. *Microservice Architecture.* [Online] 2017. [Citace: 4. 8 2018.] http://microservices.io/patterns/microservices.html.

7. **Intersoft Consulting.** GENERAL DATA PROTECTION REGULATION (GDPR). *GDPR Info.* [Online] 2018. [Citace: 2018. 3 3.] https://gdpr-info.eu/.

8. **Wagner, Bill.** Introduction to the C# Language and the .NET Framework. *Microsoft Docs.* [Online] 2018. [Citace: 3. 4 2018.] https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.

9. **Levinson, By Jeff.** What Is TFS? *Visual Studio Magazine.* [Online] 2009. [Citace: 2018. 4 16.] https://visualstudiomagazine.com/articles/2009/06/30/what-is-tfs.aspx.

10. **Landwerth, Immo.** Introducing .NET Standard. *.NET Blog.* [Online] 2016. [Citace: 1. 11 2018.] https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/.

11. **Fritz, Jeffrey T.** ASP.NET 5 is dead – Introducing ASP.NET Core 1.0 and .NET Core 1.0. *MSDN.* [Online] Microsoft Corporation, 2016. [Citace: 3. 3 2018.] https://blogs.msdn.microsoft.com/webdev/2016/01/19/asp-net-5-is-dead-introducing-asp-net-core-1-0-and-net-core-1-0/.

12. **Nagel, Christian.** *C# 6 and .NET Core 1.0.* Indianopolis, Indiana : John Wilezy & Sins Inc., 2016. ISBN: 978-1-119-09660-3.

13. **Dykstra, Tom.** Introduction to Kestrel web server implementation in ASP.NET Core. *Microsoft Docs.* [Online] Micorsoft Corporation, 2018. [Citace: 25. 02 2018.] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?tabs=aspnetcore2x.

14. **Libuv.orq.** Cross-platform asynchronous I/O: http://libuv.org/. *Github/libuv.* [Online] 2018. [Citace: 25. 02 2018.] https://github.com/libuv/libuv.

15. **Ciliberti, John.** *ASP.NET Core Recipes.* New York : Apress Media, LLC, 2017. ISBN 978-1-4842-0427-6.

16. **Fielding, R.** Hypertext Transfer Protocol. *RCF 2616.* [Online] 01 1999. [Citace: 12. 07 2017.] https://tools.ietf.org/html/rfc2616.

17. **Rick Anderson, Steve Smith.** ASP.NET Core Middleware Fundamentals. *ASP.NET Docs.* [Online] 2017. [Citace: 13. 07 2017.] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware.

18. **Dykstra, Tom.** Filters. *Microsoft Docs.* [Online] 2016. [Citace: 3. 3 2018.] https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters.

19. **Smith, Steve.** Dependency injection in ASP.NET Core. *Microsoft Docs.* [Online] 2016. [Citace: 3. 3 2018.] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

20. **Miller, Rowan.** Relationships. *Microsoft docs.* [Online] 2018. [Citace: 6. 4 2018.] https://docs.microsoft.com/en-us/ef/core/modeling/relationships.

21. **Mickens, James.** MIT Open Courseware. [Online] 2015. [Citace: 07. 07 2017.] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-858-computer-systems-security-fall-2014/video-lectures/lecture-9-securing-web-applications/.

22. **Zelenka, Josef.** *Ochrana dat: kryptologie.* Hradec Králové : Gadeamus, 2003. ISBN 80-7041-737-4.

23. **Úřad pro ochranu osobních údajů.** Zákon č. 101/2000 Sb., o ochraně osobních údajů a o změně některých zákonů, ve znění účinném od 1. července 2017. *UOOU.CZ.* [Online] 2017. https://www.uoou.cz/zakon-c-101-2000-sb-o-ochrane-osobnich-udaju-a-o-zmene-nekterych-zakonu-ve-zneni-ucinnem-od-6-rijna-2016/ds-3109/p1=3109.

24. **ARCIERI, TONY.** What's wrong with in-browser cryptography? [Online] 2013. [Citace: 03. 07 2017.] https://tonyarcieri.com/whats-wrong-with-webcrypto.

25. **Ionescu, Paul.** The 10 Most Common Application Attacks in Action. *SecurityInteligence.* [Online] IBM, 2015. [Citace: 03. 07 2017.] https://securityintelligence.com/the-10-most-common-application-attacks-in-action/.

26. **Lakshmiraghavan, Badrinarayanan.** *Pro ASP.NET Web API Security.* místo neznámé : Apress, 2013. ISBN 978-1-4302-5783-7.

27. **Anderson, Rick.** Configure ASP.NET Core Data Protection. *Microsoft Docs.* [Online] Microsoft Corporation. [Citace: 1. 4 2018.] https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/overview?tabs=aspnetcore2x.

28. **Rick Anderson**. Consumer APIs overview for ASP.NET Core. *Microsoft docs.* [Online] Microsoft Corporation. [Citace: 1. 4 2018.] https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/consumer-apis/overview.

29. **Scott, Addie.** Purpose hierarchy and multi-tenancy in ASP.NET Core. *Microsoft Docs.* [Online] Microsoft Corporation, 2018. [Citace: 28. 02 2018.] https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/consumer-apis/purpose-strings-multitenancy.

30. **Rozenblit, Jonathan.** Storing Your Secrets in Azure Websites App Settings. *Microsoft Developer.* [Online] 2015, 20. 3 2015. [Citace: 28. 2 2018.] https://blogs.msdn.microsoft.com/cdndevs/2015/03/20/storing-your-secrets-in-azure-websites-app-settings/.

31. **Galloway, Jon.** Introduction to Identity on ASP.NET Core. *Microsoft Docs.* [Online] Microsoft Corporation, 2018. [Citace: 28. 2 2018.] https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?tabs=visual-studio%2Caspnetcore2x.

32. **Hardt, D.** The OAuth 2.0 Authorization Framework. [Online] 10 2012. https://tools.ietf.org/html/rfc6749. RFC 6749.

33. **Mozilla and individual contributors.** HTTP cookies. *MDN web docs.* [Online] 2018. [Citace: 28. 2 2018.] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

34. **Venzel, Maira.** What Is Windows Communication Foundation. *Microsoft Docs.* [Online] 30. 3 2017. [Citace: 5. 12 2017.] https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf.

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Kombinovaná
Obor/komb.: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Bareš Lukáš	Rybova 1907, Hradec Králové - Nový Hradec Králové	I1600831

TÉMA ČESKY:

Vývoj webových aplikací v ASP.NET Core

TÉMA ANGLICKY:

Development of ASP.NET Core application

VEDOUCÍ PRÁCE:

Ing. Jiří Štěpánek - KIT

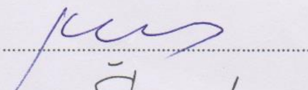
ZÁSADY PRO VYPRACOVÁNÍ:

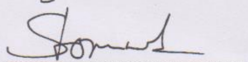
Cíl práce: Shrnout multiplatformní webový framework ASP.NET Core, porovnat jeho přednosti a nedokonalosti oproti předchozím generacím, zhodnotit jeho použitelnost v návrhu moderní webové aplikace na principu mikroslužeb, demonstrovat vlastnosti a použitelnost na praktické aplikaci.

Základní osnova:
1.Úvod
2.Principy vývoje internetových aplikací v ASP.NET Core.
3.Možnosti využití ASP.NET Core v architektuře mikrolužeb.
4.Demonstrace v praktické aplikaci.
5.Shrnutí a závěr.

SEZNAM DOPORUČENÉ LITERATURY:

Nagel, Christian. C# 6 and .NET Core 1.0. Indianopolis, Indiana : John Wilezy & Sins Inc., 2016. ISBN: 978-1-119-09660-3.
Hypertext Transfer Protocol -- HTTP/1.1. RCF 2616. [Online] 01 1999. [Citace: 12. 07 2017.] https://tools.ietf.org/html/rfc2616. rcf2616.
Newman, Sam. Building Microservices. Sebastopol : O'Reilly Media, Inc, 2015. 978-1-491-95035-7.
Ciliberti, John. ASP.NET Core Recipes. New York : Apress Media, LLC, 2017. ISBN 978-1-4842-0427-6.
Zelenka, Josef. Ochrana dat: kryptologie. Hradec Králové : Gadeamus, 2003. ISBN 80-7041-737-4.

Podpis studenta:  .... **Datum:** 15.10.2017

Podpis vedoucího práce:  .... **Datum:** 15.10.2017

(c) IS/STAG . Portál - Podklad kvalifikační práce . I1600831 . 15.10.2017 16:43