FAKULTA INFORMATIKY A MANAGEMENTU
UNIVERZITA HRADEC KRÁLOVÉ

Zavedení Microservice architektury v Legacy systému

Bakalářská práce

Obor:	Aplikovaná informatika
Ročník:	3.
Vedoucí práce:	Ing. Pavel Kříž Ph.D.
Datum zpracování:	27. 6. 2021
Autor:	Roman Šedivý

Prohlášení:

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

……..……..

V Hradci Králové dne 14.8.2022 Roman Šedivý

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Pavlu Křížovi Ph.D. za jeho cenné rady a zároveň děkuji Ing. Lubomíru Andrlovi za poskytnuté konzultace ohledně systému Damas.

Anotace:

Tato bakalářská práce se zabývá architekturou mikroslužeb a jejími rozdíly oproti monolitické architektuře. Architektura mikroslužeb využívá velké množství volně provázaných služeb, které spolu komunikují pomocí definovaného API. Jako názorný příklad nám bude sloužit systém zvaný Damas, který se nyní transformuje z monolitické architektury do architektury mikroslužeb. Cílem teoretické části práce je detailně předvést koncept architektury mikroslužeb a zhodnotit její výhody, nevýhody a rozdíly s monolitickou architekturou. Cílem praktické části práce je ukázat transformaci monolitické architektury na architekturu mikroslužeb.

Klíčová slova:

mikroslužba, monolit, architektura, systém, API, REST, synchronní, asynchronní, migrace

Annotation:

This bachelor thesis deals with microservice architecture and its differences from monolithic architecture. Microservice architecture uses a large number of loosely coupled services that communicate with each other using a defined API. As an illustrative example, we will use the system called Damas, which is now being transformed from a monolithic architecture to the microservice architecture. The aim of the theoretical part of the thesis is to show in detail the concept of microservice architecture and evaluate its advantages, disadvantages and differences from monolithic architecture. The aim of the practical part of the work is to show transformation of the monolithic architecture into microservice architecture.

Key words:

microservice, monolith, architecture, system, API, REST, synchronous, asynchronous, migration

Obsah

1. Úvod

V dnešním rychle se měnícím světě jsou požadovány čím dál tím vyšší nároky na vývoj softwaru. Je potřeba reagovat na různé trendy a popřípadě software upravit, aby těmto trendům odpovídal.

Mnoho velkých firem má v dnešní době svoji aplikaci vytvořenou podle standardů architektury mikroslužeb. Tento trend se začal objevovat v předchozím desetiletí, kdy velké firmy, jako je Netflix, Amazon, Google, Coca cola a další, začaly tuto architekturu používat pro své produkty. Tato změna mimo jiné umožňuje obratně reagovat na dnešní rychle se měnící potřeby trhu.

Architektura mikroslužeb je založená na mnoha službách, které spolu vzájemně komunikují pro dosažení společného cíle. Každá tato služba je samostatná, je tedy potřeba, aby byla co nejméně závislá na fungování ostatních služeb. Má svůj vlastní životní cyklus a většinou i speciální tým, který danou službu spravuje a integruje do velké sítě těchto separátně vyvíjených služeb.

Hlavním cílem této práce je nalézt limity architektury mikroslužeb v kontextu jejího zavádění do monolitického systému. K tomu nám bude sloužit systém Damas MMS (Market Management System), vyvíjený firmou Unicorn a. s. Tento systém slouží jako nástroj pro podporu různých obchodních procesů v energetice a také jako prostředí, kde je možné tyto procesy v reálném čase modelovat a nasazovat. Druhým cílem je zhodnotit přínos zavedení mikroslužeb oproti stávajícímu monolitickému řešení.

Tato práce nemá za cíl zavrhnout použití monolitické architektury. Naopak v mnoha případech může být modulární monolitická aplikace vhodnější než aplikace postavená na mikroslužbách.

V teoretické části práce jsou popsány jednotlivé architektury a porovnány jejich výhody a nevýhody. V praktické části je popsán systém Damas a migrace tohoto systému na architekturu mikroslužeb.

1.1 Vývoj architektury softwaru

Typ architektury je to, co umožňuje systému se vyvíjet a poskytovat určitou úroveň služeb během jeho životního cyklu. V softwarovém inženýrství se architektura zabývá tím, jakým způsobem co nejlépe vyřešit spojení systémových funkcionalit a požadavků na kvalitu, které systém musí splnit. Za posledních pár desítek let byla architektura softwaru detailně studována a jako výsledek přišli softwaroví inženýři s různými nápady, jak vytvářet systémy, které poskytují širokou škálu funkcionalit a uspokojí velké spektrum požadavků. (3)

Problémy spojené s vývojem rozsáhlých programů byly poprvé zaznamenány okolo 60. let 20. století. Kvůli těmto problémům vznikl velký zájem o zkoumání designu softwaru a jeho důsledky na proces vývoje. V této době nebyl design považován jako součást implementace, tudíž pro něj byla potřeba speciální sada notací a nástrojů. V 80. letech byl design softwaru plně zaintegrován do vývoje. V roce 1992 přišli Perry a Wolf s termínem architektura softwaru. Jejich definice byla rozdílná od designu softwaru, a přilákala tak spoustu lidí, kteří začali zkoumat praktické využití těchto poznatků v průmyslové a akademické sféře. Tento vzestup zájmu s sebou také přinesl tvorbu různých architektonických stylů, ze kterých bylo potřeba identifikovat ty užitečné, zhodnotit je a porovnat. (3)

1.1.1 Objektově orientovaný design

Objektově orientovaný design vznikl v 80. letech 20. století a poté v 90. letech přispěl do oblasti softwarové architektury. Společně s rozvojem objektově orientovaného designu začaly vznikat i různé návrhové vzory. Ty mají za cíl definovat způsoby, pomocí kterých se dá vytvořit udržitelný a dobře strukturovaný software ve velkém měřítku. Typickým příkladem návrhového vzoru je v objektově orientovaném programování Model-View-Controller (MVC). Tento návrhový vzor vznikl v roce 1979 s cílem oddělit logickou vrstvu od prezentační a je často používán pro tvorbu robustných aplikací. (3)

1.1.2 Monolitická architektura

Monolitická architektura se vyznačuje nutností nasadit všechny komponenty systému společně. Mezi typy monolitických systémů patří monolitické systémy s jedním procesem a distribuované monolitické systémy. (7)

Monolitické systémy s jedním procesem jsou prozatím nejrozšířenější variantou monolitických systémů. Díky své jednoduché topologii mají mnoho výhod. Jednoduše se nasazují, monitorují, testují a obecně mají jednodušší postupy při činnostech spojených s jejich spravováním. Pro větší robustnost a potřeby škálování může být použito více instancí stejného systému, ale každá instance vždy obsahuje všechny části systému. Tato architektura se zdá být obecně vhodnou pro méně rozsáhlé aplikace či začínající společnosti. (7)

Poddruhem těchto systémů jsou modulární monolity. Tento typ klade velký důraz na určení hranic mezi jednotlivými částmi, které jsou následně rozděleny do separátních modulů. Ve většině případů je modulární monolit velmi dobrá volba. Správné určení hranic mezi moduly umožňuje velkému množství programátorů pracovat na stejné aplikaci. Tato architektura se také nemusí potýkat s nevýhodami aplikací postavených na mikroslužbách. Nejznámější firmou, která jako svou architekturu používá modulární monolit, je Shopify. (7)

Kód aplikace každou novou funkcí roste. Po čase, když se kód rozroste do velkých rozměrů, je složité dodržovat hranice mezi jednotlivými částmi aplikace. Navzdory snaze o čistotu zdrojového kódu, pomocí rozdělení jednotlivých částí do modulů, se bariéry mezi jednotlivými moduly nepřesně dodržují a kód související s jednou funkcionalitou se začne vyskytovat v různých modulech projektu. Velký objem zdrojového kódu u monolitických systémů prodlužuje délku potřebnou pro pochopení fungování procesů a také zesložiťuje revizi kódu. To vede ke zpomalení procesu vývoje a prodlužuje čas mezi vydáváním nových verzí a přidáváním nových funkcí. V monolitickém systému se snažíme pro udržení čistoty kódu zavést abstrakci. (1)

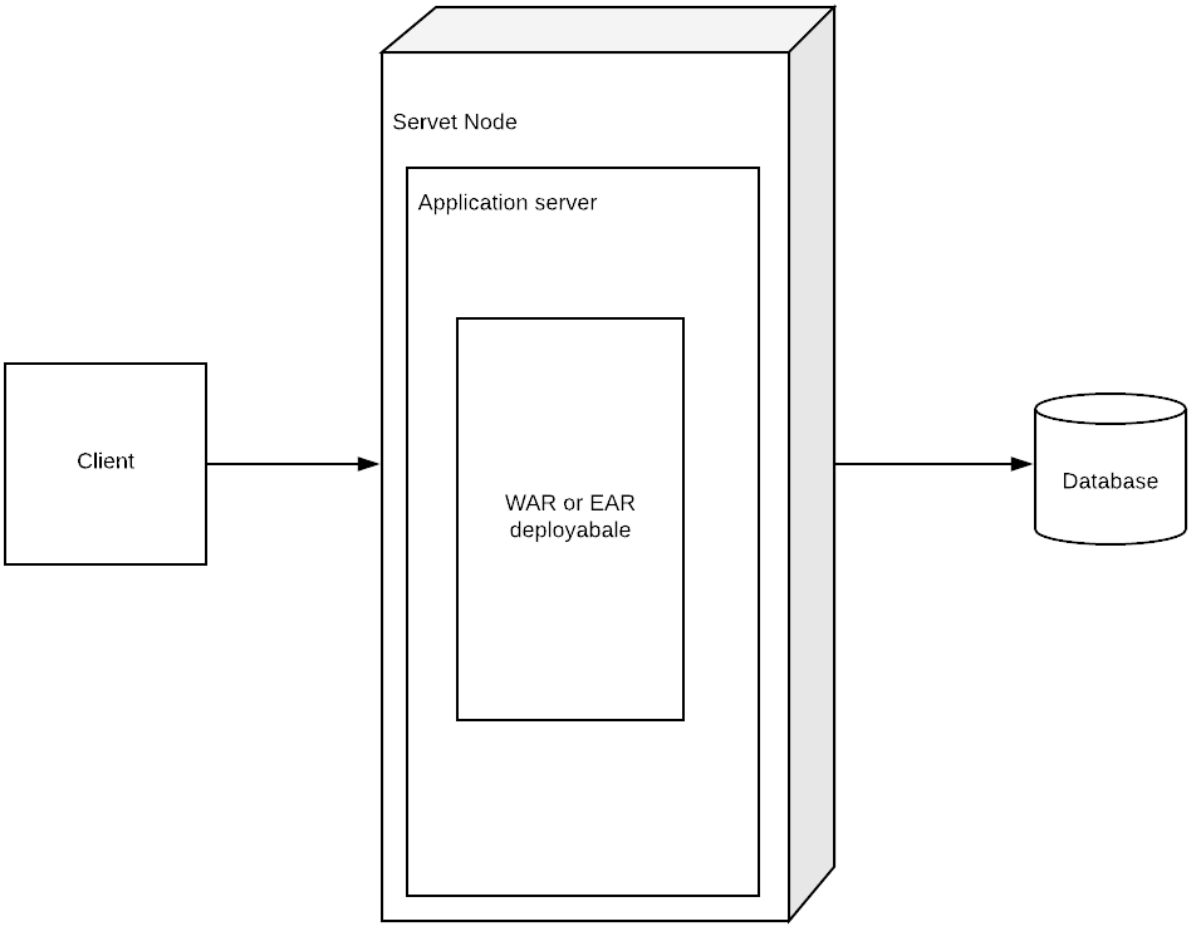
Dalším druhem monolitických systémů je distribuovaná monolitická aplikace. Tato aplikace se skládá z více služeb, které musí být pro správný chod systému nasazeny společně. Distribuované monolitické systémy se potýkají s nevýhodami distribuovaných a zároveň monolitických systémů. Takové systémy většinou vznikají v prostředí, kde není kladen důraz na koncepty, jako je soudržnost nebo skrývání informací. Kvůli nedodržování těchto konceptů

začnou být jednotlivé služby mezi sebou silně provázané a celý systém se začne chovat jako monolit. (7)

Monolitická aplikace má i mnoho omezení. Největším omezením v dnešní době se zdá být složitost přechodu na novější technologie. Například přechod z Javy 6 na Javu 8 znamená, že celá aplikace musí být důkladně otestována a každý malý problém oddálí možnost přechodu na novější verzi. (5)

Vzhledem k tomu, že všechny části monolitické aplikace existují jako jeden proces, není možné škálovat pouze části, které jsou pod velkou zátěží. Je možné škálovat buď celou instanci, nebo navýšit výpočetní zdroje serveru. V obou případech dojde ke škálování částí, které pod zátěží nejsou, což není výhodné. (5)

S monolitickými aplikacemi je spojená i nízká flexibilita změn. Při potřebě přidání nové funkce musí dojít ke koordinaci všech členů, kteří se na vývoji aplikace podílejí. Taková aplikace je také odkázána na původní technologii, ve které byla vytvořena. (5)



Obrázek 1. – Monolitická aplikace

Na obrázku výše můžeme vidět typickou monolitickou aplikaci psanou v jazyce Java. Můžeme si všimnou jednoduché struktury celého systému. Na aplikačním serveru je nahraný balíček obsahující všechny potřebné komponenty pro fungování aplikace a server komunikuje pouze z databází, kam jsou ukládána data.

1.1.3 Architektura orientovaná na služby

Princip oddělení zodpovědností (anglicky separation of concerns) vedl k vytvoření designu založeném na komponentách, díky kterému bylo možné mít větší kontrolu nad implementací, designem a celkovým rozvojem systému. Architektura orientovaná na služby vznikla spojením komponentového přístupu a objektově orientovaného designu. V této architektuře je software rozdělený na jednotlivé volně provázané komponenty zvané služby, které spolu komunikují pomocí zpráv. (3)

Hlavním cílem této architektury je poskytnout prostředek k vytvoření velkých distribuovaných systémů, které jsou škálovatelné a flexibilní. Kvůli dlouhé absenci stručné definice pro tuto architekturu vytvořili Footen a Faust následující: „Architektura orientovaná na služby je architektura nezávislých služeb komunikujících prostřednictvím publikovaných rozhraní přes společnou vrstvu middlewaru.“ (4)

Josuttis (9) navrhoval, že jako middleware by měl být použit Enterprise Service Bus (ESB). ESB odděluje jednotlivé služby, čímž zlepšuje schopnost systémů dosáhnout vzájemné součinnosti a zmenšuje počet komunikačních kanálů. Místo toho, aby jednotlivé služby komunikovaly mezi sebou, posílají své požadavky pouze do ESB. ESB řídí posílání zpráv a podporuje pro to mnoho vzorů, jako je například asynchronní požadavek / odpověď nebo publikování a odběr.

Architektura orientovaná na služby se rozrostla na začátku 21. století společně s vzestupem internetu a celosvětovou sítí (World Wide Web), která se začala stávat dostupnou pro širokou veřejnost. Nově vyvíjené webové technologie a síťové protokoly usnadňovaly tvorbu aplikací, založených na volně provázaných komunikujících službách. Hlavní hnací silou byly protokoly HTTP a SOAP. Velké technologické firmy, jako IBM, Oracle nebo Hewlett Packard, se také začaly orientovat na tuto architekturu a vytvořily okolo ní celý ekosystém. Vytvořily nejen nástroje, technologie a návrhové vzory, ale také tuto architekturu rozšířily do obchodních sfér. Tento krok vedl k velké kritice od lidí, kteří považovali architekturu orientovanou na služby jen jako pokus firem, poskytujících služby v oboru informačních technologií, zbohatnout. (9)

2. Architektura mikroslužeb

Architektura mikroslužeb je architektonický styl, který strukturuje aplikaci do souboru služeb, které spolu komunikují pomocí definovaného API. Funkce těchto služeb jsou určovány jednotlivými byznysovými doménami aplikace. Každá služba je zodpovědná pouze za jednu vymezenou funkcionalitu celého programu, čímž se snažíme vytvořit vysoce soudržné a volně provázané služby. Všechny tyto služby by mělo být možné jednotlivě plně automaticky nasadit. (6)

Tato architektura se podle Fowlera (1) vyvinula z architektury orientované na služby a vznikla jako přístup, kterým se vypořádat s nedostatky velkých monolitických aplikací. Je to

přístup, který se zaměřuje na opětovné použití softwaru. Tato architektura ovšem nemluví o tom, jak rozdělit něco velkého na menší části, nebo o tom, kdy už je program příliš velký a neřeší praktické problémy. Architekturu mikroslužeb tedy můžeme chápat spíše jako specifický přístup k servisně orientované architektuře.

2.1 Charakteristiky mikroservis

2.1.1 Jeden účel a malá velikost

Základní charakteristikou mikroslužeb je soustředit se na jednu věc a dělat ji dobře. To znamená mít dobře vymezené hranice. Při vytváření jednotlivých mikroslužeb se vždy dekomponuje jeden velký systém do malých subsystémů. V dnešní době existuje mnoho způsobů, kterými se dá velký systém rozdělit na menší. Známý způsob, který se pro rozdělení systému do částí používá, je doménově orientovaný design. Tento způsob se řídí tím, že na každý systém může být nahlíženo jako na takzvaný model reality. V tomto designu je celkový model systému tvořený několika menšími. Jednotlivé modely reprezentují části systému, zaměřené na jednotlivé části byznysu, které systém pokrývá. Právě ohraničením těchto modelů, tedy částí systému, které jsou používány ve stejném kontextu, můžeme získat jednotlivé malé mikroslužby s jedním účelem. (2)

2.1.2 Autonomie

V architektuře mikroslužeb jsou jednotlivé mikroslužby brány jako odlišné entity, které je možné nasazovat nezávisle na sobě. Jejich nasazení by mělo být prováděno na různých strojích. Přestože takováto izolace přidá určitou zátěž navíc, výsledná jednoduchost umožňuje o celém systému snadněji uvažovat. Také je třeba přemýšlet o tom, které věci by měla jednotlivá služba nabízet a které schovat. Jestliže jednotlivé mikroslužby sdílí příliš mnoho informací, může mezi nimi dojít k vysoké provázanosti. Taková věc snižuje autonomii a při potřebě změny je nutná koordinace více mikroslužeb. (1)

2.1.3 Zapouzdření

Každá mikroslužba by měla exkluzivně vlastnit svá data a mít prostory pro jejich ukládání. Všechny zápisy do těchto prostor musí být prováděny pouze pomocí vystaveného API a žádná mikroslužba by neměla mít možnost zapisovat do těchto prostor přímo. Cílem architektury mikroslužeb je redukovat vzájemné závislosti jednotlivých částí systému. Jestliže mikroslužba exkluzivně vlastní svá data, snižuje se její provázanost. Vysoká provázanost

nastane tehdy, když více mikroslužeb buď čte, nebo zapisuje stejná data, čímž se poruší základní vlastnost jejich architektury. (8)

2.1.4 Menší týmy

Jednu obrovskou monolitickou aplikaci může programovat i 100 vývojářů. Problém s tímto modelem však spočívá v tom, že jednotliví vývojáři nemusí cítit žádnou odpovědnost za výslednou kvalitu aplikace. V ekonomice se tomuto termínu říká Společná tragédie, což znamená, že jednotlivci dělají jen to, co je pro ně nejvýhodnější. Tento stejný problém nastává i u monolitických systémů, kdy mnoho zaměstnanců dělá pouze to, co považují za důležité. Právě to způsobí velmi komplikovanou monolitickou aplikaci, která používá zastaralé technologie. Nakonec se s komplexitou a technickým dluhem musí potýkat nejen člověk, který je za to zodpovědný, ale i ostatní členové. Mikroslužby fungují z velké části díky pocitu zodpovědnosti. Malé týmy o 2 až 15 lidech vyvíjí, nasazují a starají se o jednu mikroslužbu po celý její životní cyklus. V malém týmu má každý jeho člen pocit zodpovědnosti za mikroslužbu a snaží se, aby byla úspěšná. Jestliže mikroslužbu běží po celý čas bez výpadků, tým může být odměněn. Oproti tomu, když mikroslužba úspěšná není, je jednoduché přiřadit vinu. (8)

2.2 Výhody použití mikroslužeb

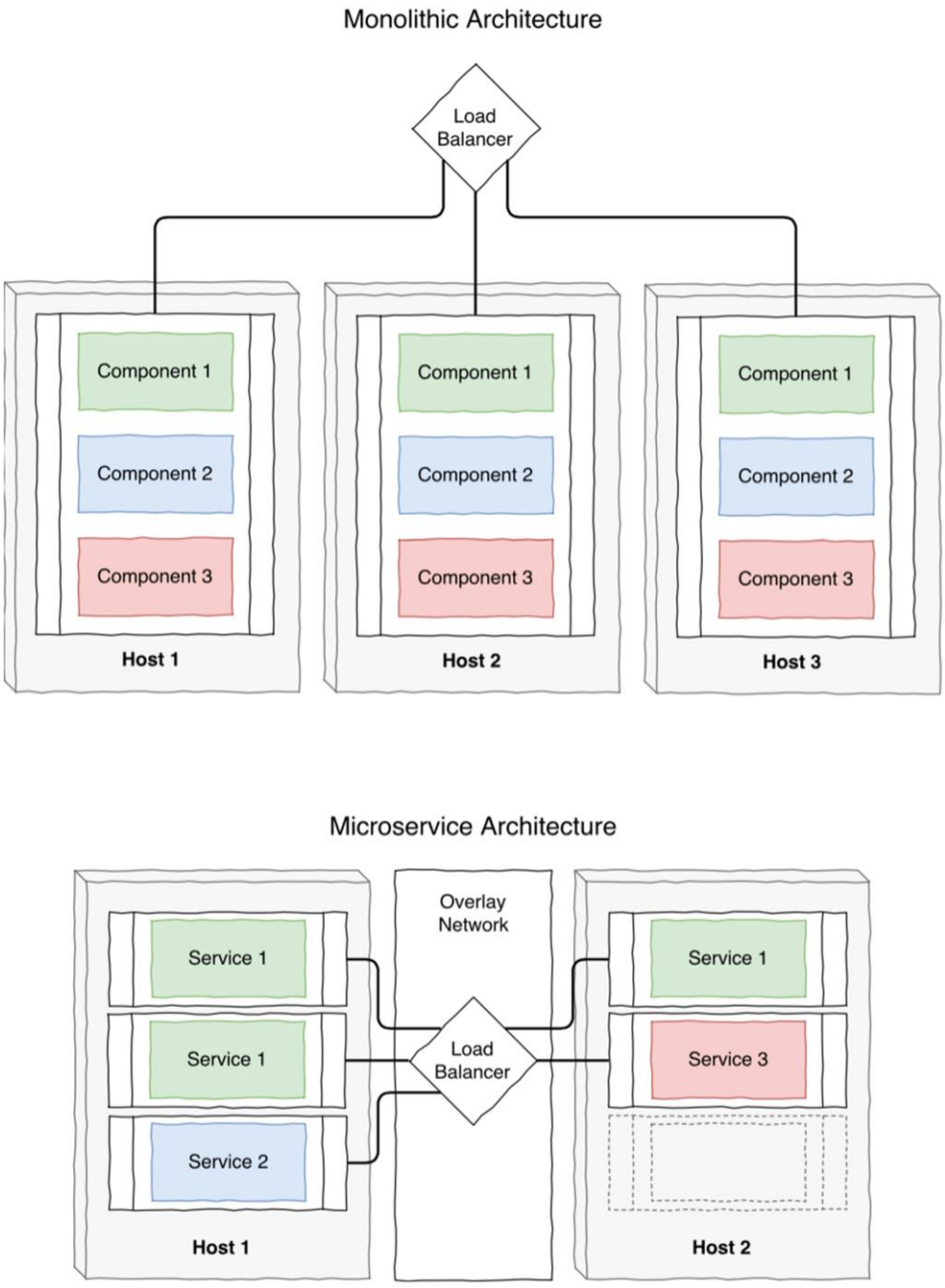
Výhod používání mikroslužeb je mnoho. Velkou řadu těchto benefitů můžeme přiřadit k jakémukoliv distribuovanému systému. Architektura mikroslužeb se snaží tyto výhody ještě prohloubit primárně kvůli tomu, jak daleko bere koncept distribuovaných systémů a servisně orientované architektury. (1)

2.2.1 Škálovatelnost

Škálování softwaru je způsob, pomocí kterého se dá vypořádat s vysokou zátěží systému. Existují dvě varianty škálování. První varianta se nazývá horizontální škálování. To znamená, že přidáváme nové instance aktuálně nasazeného systému nebo služby. Druhá varianta je vertikální škálování, což znamená, že zvýšíme výkon serveru, na kterém je daná aplikace nasazena. U systémů skládajících se z mikroslužeb můžeme škálovat pouze části, které jsou pod velkou zátěží. Tento způsob škálování je výhodný, protože jednotlivé instance mikroslužeb můžeme přidávat pouze tehdy, když je daná část pod vysokou zátěží. (12)

Na obrázku číslo 2 je ukázán rozdíl v horizontálním škálování mezi monolitickým systémem a systémem založeným na mikroslužbách. V tomto případě je funkce komponenty v monolitickém systému shodná s funkcí mikroslužby v systému založeném na mikroslužbách. Jestliže dojde ke zvýšení zátěže komponenty číslo 1, musí dojít k zapnutí více instancí celého monolitického systému, a to i přesto, že ostatní komponenty pod vyšší zátěží nejsou. Při stejné zátěži na mikroslužbu 1 je možné vytvořit nové instance pouze této služby.

K rozdělování zátěže mezi více běžících instancí služby se používá load balancing. Load balancing je process distribuce zátěže na jednotlivé instance mikroslužeb v distribuovaného systému s cílem zefektivnit využití dostupných výpočetních zdrojů, zlepšit odezvu a také se vyhnout situacím, kdy je jedna z instancí pod velkou zátěží a ostatní jsou nečinné nebo velmi málo vytížené.



Obrázek 2. – Ukázka horizontálního škálování

Známý webový obchod Gilt přešel na architekturu mikroslužeb přímo kvůli možnosti horizontálního škálování. V roce 2007 byl vytvořen jako monolitická aplikace. Od roku 2009 se systém Giltu nebyl schopný vyrovnat s vysokou zátěží způsobenou mnoha zákazníky. To se změnilo rozdělením a osamostatněním základních částí systému, díky čemuž mohly být jednotlivé části škálovány, a Gilt se tak vyrovnal s vysokou zátěží. Dnes tento obchod vlastní více než 450 mikroslužeb. (1)

2.2.2 Technologická rozmanitost

Komunikace jednotlivých služeb pomocí formátů nezávislých na platformě zaručuje možnost výběru vhodného programovacího jazyka, popřípadě databáze. Například, když je pro určitou část programu klíčová rychlost, je možné zvolit technologii, pomocí které tohoto cíle dosáhneme. S použitím mikroslužeb je také jednodušší začít používat nové technologie a porozumět, jakým způsobem mohou být prospěšné celému systému. Největší bariérou při přechodu na nové technologie je riziko, že mohou systém více poškodit než mu prospět. U monolitické aplikace je dopad nesmírně vysoký, protože použití nové technologie znamená dopad na celou aplikaci. U mikroslužby, která není kritická pro chod systému, můžeme otestovat použití nové technologie s vědomím, že dokážeme limitovat potenciální negativní dopady. (1)

2.2.3 Více nasazených verzí

U architektury mikroslužeb je možné na jednom prostředí nasadit více verzí stejné mikroslužby. Všechny verze mohou zpracovávat požadavky a posílat odpovědi. Je tedy možné mít na jednom prostředí nasazenou například verzi 1.2, 1.3 a 2.0. Klienti si mohou při tvorbě požadavku zvolit verzi, na kterou se dotazují. Verze je většinou specifikována přes URL. Po čase, kdy již nejsou klienti, kteří by starou verzi používali, může být odstraněna. Schopnost podpory více verzí je jednou ze silných stránek mikroslužeb. (8)

2.2.4 Robustnost

Jedna vlastnost při vývoji mikroslužeb je design proti pádu. Jakákoliv služba může být nedostupná kvůli neočekávanému výpadku sítě nebo nějaké chybě v infrastruktuře. Spíše než snaha o ošetření všech možných pádů aplikace, k čemuž v určitý moment stejně dojde, je lepší nechat komponentu spadnout a udělat to rychle. Při tom je důležité detekovat vypadlé služby v reálném čase a automaticky je zase zapnout. Tento způsob řízení pádu jednotlivých mikroslužeb vede k lepší stabilitě, rychlejšímu zotavení z chyb a garantovaná vysoká dostupnost pro celý systém zaručí, že si výsledný uživatel ani nevšimne času, po který byla mikroslužba vypnutá. (1)

Požadavky zákazníků na dostupnost softwaru, stejně tak jako nutnost používání aplikací, se v dnešní době zvýšila. Rozdělením monolitického systému na více individuálních, samostatně nasaditelných služeb, zvýšíme robustnost aplikace. Díky použití mikroslužeb získá aplikace schopnost implementovat robustnější architekturu, protože jednotlivé funkce

jsou od sebe odtrženy a negativní účinek na část systému nezpůsobí pád celého systému. Také je tím umožněno více se soustředit na kritické části systému, které potřebují být robustné. (7)

2.2.5 Jednoduché nasazení

Pro malý zásah do části velkého monolitického systému, který obsahuje miliony řádků kódu, je potřeba vydat a nasadit novou verzi celé aplikace. Ve většině prostředí se kvůli drobné opravě nevytvoří opravná verze, ale čeká se, až se jednotlivé chyby nakupí a nová verze se vytvoří se všemi opravami najednou. U mikroslužeb je možné udělat změnu pouze u jedné služby a tu nezávisle na ostatních nasadit. Takto může být kód jednotlivých mikroslužeb rychleji aktualizován a nové funkce se dostanou k zákazníkům rychleji. (1)

2.3 Nevýhody použití mikroservis

Přestože architektura mikroslužeb nabízí spoustu výhod, monolitické systémy jsou stále hojně používány. Existují firmy, jako je Shopify, která při růstu změnila pouze strukturu jejich systému, anebo firma Segment, která několik let po přechodu na mikroslužby přešla zpět na monolitickou architekturu. Je zřejmé, že pro velké technologické společnosti, jako je například Google nebo Netflix, dává použití mikroslužeb smysl už jen kvůli možnosti škálování, ale pro menší aplikace s několika tisíci uživateli by monolitický systém mohl být dostačující. Tato bakalářská práce se tedy nesoustředí na znevážení monolitické architektury, ale pouze poukazuje na případy, kdy se zdá být architektura mikroslužeb výhodnějším řešením. V této sekci jsou popsány některé nevýhody architektury mikroslužeb.

2.3.1 Náročnější vnější složitost

Na mikroslužby je občas nahlíženo tak, že mění vnitřní složitost za vnější. Monolitické aplikace komunikují pouze v rámci jednoho procesu a nemusí se komunikací vůbec zabývat. Možnost vytvářet více menších aplikací jako odlišné mikroslužby dovoluje týmům vyvíjet a nasazovat nové funkcionality velmi rychle, avšak interakce mezi jednotlivými mikroslužbami musí být ošetřeny například synchronním či asynchronním kopírováním dat. Dále je také složitější ladění komunikace. Chyba může například nastat při komunikaci 13. verze jedné komponenty s 19. verzí druhé. Každá mikroslužba může mít více spuštěných verzí ve stejném prostředí a je nemožné testovat všechny interakce. (8)

2.3.2 Organizační vyspělost

Struktura organizace určuje, jakým způsobem je software tvořen. Centralizované organizace strukturované okolo jednotlivých vrstev se soustředí na vytvoření týmů založených na jejich technických znalostech, jako je například operační nebo databázový tým. V takovém případě každá menší změna v aplikaci vyžaduje komunikaci napříč jednotlivými týmy. Architektura mikroslužeb vyžaduje změnu struktury takovéto firmy. Je zapotřebí, aby jednotlivé týmy začaly strukturovat podle jednotlivých mikroslužeb, namísto vrstev. Měly by tedy vzniknout týmy zodpovídající za jednotlivé mikroslužby, které budou obsahovat lidi s různými technickými znalostmi. Tato základní změna v myšlení vede ve výsledku k tvorbě lepšího softwaru. Z firemního pohledu je potřeba, aby se na informační technologie přestalo nahlížet jako na náklad, který je zapotřebí minimalizovat. (8)

Kromě struktury je také zapotřebí, aby organizace měla silné povědomí o tom, co je agilní vývoj softwaru, DevOps a cloud. Týmy vlastnící jednotlivé mikroslužby využívají agilních metodik pro řízení vývoje jednotlivých mikroslužeb. DevOps je postup, který spojuje vývojovou a operační část týmu a slouží k podpoře celého životního cyklu aplikace. Tento postup je velmi důležitý, protože každý tým je zodpovědný za programování a také nasazení své mikroslužby. Cloud je pro mikroslužby velmi dobrou volbou, a proto je na něj nasazována většina. U cloudu je podstatné to, že je distribuovaný svou povahou, tedy znalost distribuovaných výpočtů (angl. Distributed computing) zjednoduší přechod na architekturu mikroslužeb, protože je zde distribuované všechno. (8)

2.3.3 Duplikace

Každý tým by měl mít možnost vybrat si pro novou mikroslužbu vlastní programovací jazyk nebo databázi, kterou bude mikroslužba používat. Velmi často se stává, že manažer aplikace je orientovaný na nízké náklady a například omezí použití programovacího jazyka pouze na Javu. Takové omezení může být velmi svazující, protože každý programovací jazyk má své nedostatky. Další forma duplikace je v instancích samotných. Je výhodnější mít jednu obří databázi, kterou používají všechny mikroslužby, což je však v rozporu s jejich architekturou, neboť to mezi nimi vytváří vysokou provázanost. Jestliže se v tomto případě tým zodpovědný za databázi rozhodne pro její vypnutí a aplikování nějaké úpravy, tak se všechny ostatní týmy musí podřídit. (8)

2.3.4 Konzistentnost

Velký problém, který se u mikroslužeb vyskytuje, je to, že všechna data nejsou silně konzistentní. Produkt může například existovat ve 20 různých mikroslužbách, kde každá z těchto mikroslužeb má kopii produktu z jiného časového období. Firmy jsou zvyklé na používání jedné databáze, což zajistí silně konzistentní data – jednu kopii, která je neustále aktuální. V systému založeném na architektuře mikroslužeb vždy existuje jedna komponenta, která má nejaktuálnější data. Například produktová mikroslužba vlastní všechna produktová data, ale katalog produktů a mikroslužba zodpovídající za vyhledávání může mít kopii uloženou v mezipaměti. Je tedy dobré uchovávat dokumentaci, kde bude popsáno, která data vlastní jednotlivé mikroslužby. (8)

2.4 Rozdíly s monolitickou architekturou

V následující tabulce jsou zobrazeny hlavní rozdíly mezi monolitickou architekturou a architekturou založenou na mikroslužbách.

Monolitická architektura	Architektura mikroslužeb
Jeden proces, ve kterém komunikují jednotlivé komponenty	Mnoho menších společně komunikujících služeb
Škálování probíhá duplikací celého systému	Možnost škálovat jednotlivé mikroslužby
Komunikace probíhá v rámci jednoho procesu	Komunikace probíhá po síti buď synchronním, nebo asynchronním způsobem
Pro nasazení úprav je vždy zapotřebí nasazení celé aplikaci	Je možné samostatně nasazovat jednotlivé mikroslužby
Přidávat nové funkcionality je možné pouze pomocí technologií, které byly vybrány na začátku	Je možné flexibilně přidávat nové funkce v jazycích, které jsou k tomu vhodné

Tabulka 1: Rozdíly mezi architekturou mikroslužeb a monolitickou architekturou

2.5 Vlastnosti určující kvalitu mikroslužeb

Při implementování mikroslužeb existují 2 klíčové koncepty. Prvním je takzvaná volná provázanost (angl. loose coupling) a druhým je vysoká soudržnost (angl. high cohesion). V následujících sekcích budou tyto 2 vlastnosti detailně popsány, protože nedodržení těchto konceptů může vést k velmi špatné výsledné implementaci. Tyto vlastnosti jsou známe již z objektově orientovaného programování. U mikroslužeb je však dodržení těchto konceptů mnohem důležitější.

2.5.1 Provázanost

Míra provázanosti určuje závislost jedné mikroslužby na ostatních. Jedna z výhod architektury mikroslužeb oproti monolitické je, že systém dokáže dělat změny pouze na jedné mikroslužbě a nemusí kvůli tomu nasazovat celý systém. Pro tvorbu volně provázaných mikroslužeb je potřeba řídit se principem zapouzdření. Tedy, že všechny mikroslužby by měly vědět pouze nejmenší možné detaily implementace ostatních služeb. Když jsou jednotlivé mikroslužby silně provázané, část systému, která se musí upravit, vzroste a může tak postupem času dojít k vytvoření distribuovaného monolitického systému. (1)

2.5.2 Soudržnost

Míra soudržnosti u jednotlivých mikroslužeb určuje, do jaké míry je jejich chování podobné. Jednou z charakteristik mikroslužeb je, že by měly být malé a soustředit se pouze na jednu věc a dělat ji dobře. Čím menší soudržnost jednotlivé mikroslužby mají, tím složitější je používat je na různých místech. Pro navrhnutí vysoce soudržné mikroslužby musíme najít hranice, které vymezují funkcionalitu, kterou se bude daná mikroslužba zabývat. (1)

3. Komunikace mikroslužeb

Rozhodnout, jakým způsobem spolu budou mikroslužby komunikovat, je jedno z nejdůležitějších a nejzákladnějších rozhodnutí při implementování systému založeném na mikroslužbách. Komunikace u mikroslužeb hraje daleko důležitější roli, než je tomu u monolitické aplikace, protože v ní se mohou jednotlivé funkce volat na úrovni kódu. Důležitou částí při vybírání prostředků pro implementaci komunikace je rozhodnutí, zda bude komunikace probíhat synchronně nebo asynchronně. Následující sekce popisují oba tyto způsoby.

3.1 Synchronní komunikace

Při používání synchronní komunikace mezi procesy pošle klient požadavek na službu. Služba poté zpracuje poslaný požadavek a zašle zpět odpověď. U většiny klientů existuje vlákno, které pošle požadavek a je blokováno do doby, než přijde odpověď. Existuje množství protokolů, ze kterých si vybrat. Mezi nejznámější typ synchronní komunikace patří REST. (14)

REST je architektonický styl, který se běžně používá při vyvíjení API pro webové služby. REST API je jedna z nejběžnějších metod pro vyměňování dat mezi dvěma službami bez ohledu na architekturu a jazyk, který pro ně byl použit. V tomto způsobu komunikace program, který je v roli klienta, posílá požadavek na serverem vystavené API pomocí HTTP protokolu, který mu následně pošle odpověď. Pro systém, který používá REST architekturu pro komunikaci mezi jednotlivými službami, je typické, že na každé službě také běží webový server na specifických portech, jako je 8080 nebo 443, a každá služba má vystavený několik koncových bodů, které umožňují interakci mezi jednotlivými službami. Pro každý koncový bod platí, že má své rozhraní, které určuje, jaké operace mohou ostatní služby zavolat. Tento způsob je podobný jako u monolitických systémů, akorát, že zde se rozhraní jednotlivých komponent píše ve stejném jazyce jako celý systém. V architektuře mikroslužeb specifikuje API kontrakt mezi 2 službami. Každé API má list operací se jménem, parametry a návratovými hodnotami. (14)

REST API se skládá ze 2 částí: z adresy a typu operace. V názvu adresy je typicky napsáno, pro jakou entitu se daná operace provádí, a v názvu operace je HTTP metoda, která uvádí, jakým způsobem se bude s danou entitou manipulovat. Například GET metoda značí požadavek na získání informací o dané entitě.

3.2 Asynchronní komunikace

Asynchronní forma komunikace a technologie vytvářené pro tuto komunikaci představují nové možnosti, jak pohlížet na komunikaci mezi jednotlivými službami. Správně aplikované asynchronní zpracovávání zpráv může zlepšit škálovatelnost aplikace a zvýšit její odolnost proti chybám. Asynchronní zpracování znamená odesílat požadavky, které neblokují běh aplikace. V asynchronním modelu ten, kdo zašle požadavek na službu, nečeká nečině na odpověď, ale pokračuje dál v konání své činnosti. (11)

Důležitá část asynchronní komunikace jsou fronty zpráv. Fronta zpráv je komponenta, která shromažďuje a distribuuje asynchronní požadavky. Jako zprávu si například můžeme představit XML nebo JSON soubor se všemi potřebnými daty pro provedení požadované operace. Zprávy jsou vytvářeny producenty a poté ukládány do jednotlivých front. Nakonec jsou tyto zprávy doručeny konzumentům těchto zpráv, kteří provádějí asynchronní akce, které producent těchto zpráv požaduje. Fronty zpráv mohou být implementovány různými způsoby. Například se může jednat o sdílenou složku, ve které mají jednotlivé služby práva na vytváření a čtení souborů, nebo se může jednat o sofistikovanější způsob, kde dochází k ukládání, přesměrování a odesílání zpráv. V případě potřeby, aby pro zprávy ve frontách mohly být použity funkce, jako je kontrola práv, přesměrování nebo zotavení se z chyb, se implementuje fronta jako odlišná nezávislá služba. Této službě se obvykle říká message broker nebo message-oriented middleware. (11)

Producenti zpráv reprezentují část aplikace, která vytváří asynchronní požadavky a posílá je do front. Odesílání zpráv do fronty se také někdy nazývá publikováním. Úkolem producenta je tedy vytvořit validní zprávu a odeslat ji do fronty. Konzumenti zpráv reprezentují komponenty, které provádí požadované asynchronní operace. Nejdůležitější činností konzumentů zpráv je získat a zpracovávat zprávy z front.

4. Migrace monolitické architektury na architekturu mikroslužeb

Před začátkem dekomponování monolitického systému je důležité stanovit si, kde začít a jakým způsobem se systém změní, ale hlavně jestli je vůbec výhodné na architekturu mikroslužeb přejít. Je třeba si uvědomit, že implementace nové architektury není cíl a že by volba migrace měla být založená na racionálních rozhodnutích. Také je potřeba přemýšlet nad tím, čeho s aktuální implementací systému nemůžeme dosáhnout a k čemu nám mikroslužby pomohou. Je špatné uvažovat způsobem, že když pro jednotlivé firmy, jako je Netflix nebo Amazon, jsou mikroslužby výhodné, tak musí být výhodné pro všechny.

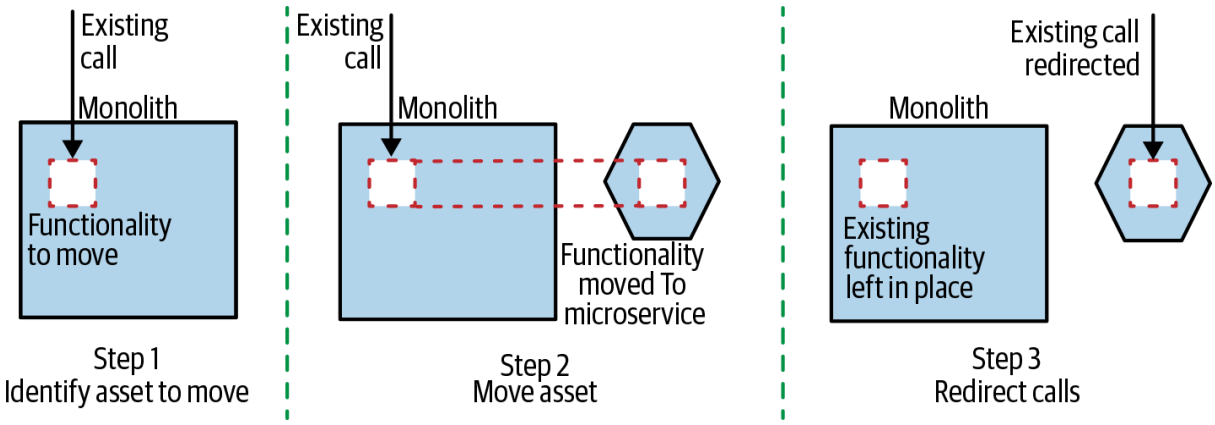
Při migraci musí být architekt velmi dobře obeznámený s oblastí, ve které daný software vytváří. Při neznalosti může dojít k nedobrému určení hranic mezi jednotlivými mikroslužbami, což může způsobit vytvoření vysoké provázanosti. Takto vytvořený software se po čase začne chovat jako distribuovaný monolit. Z toho vyplývá, že k migraci se nemá smysl uchylovat, jestliže je společnost nově založená a nemá v dané oblasti velké zkušenosti.

Nakonec je nejdůležitější stanovit si, jestli je vůbec možné systém změnit. Mohou nastat situace, které tuto změnu znemožňují. Například části systému mohou být psané v technologiích, které už aktuální programátoři na projektu neovládají, nebo systém je již v tak špatném stavu, že náklady na změnu by byly enormně vysoké.

4.1 Inkrementální migrace

Vytvořením sady nástrojů pro migraci na architekturu mikroslužeb se zabíral ve své práci Bakalalai aj. (16). Cílem této práce bylo poskytnout sérii migračních vzorů a strategií, jak je uplatňovat. Při zhodnocení navržených vzorů zjistil, že 85 % těchto návrhových vzorů bylo uplatněno při migrací 3 projektů, kterými se jako součást této studie zabývali. Můžeme zde nalézt návrhové vzory podle problému, se kterým se v aktuální fázi migrace potýkáme. Jeden ze vzorů například radí dekomponování monolitického systému pomocí doménově orientovaného designu nebo zavedení vyvažovače zátěže.

Jeden z velmi detailně popsaných způsobů, které se používají pro inkrementální migraci, se nazývá „Strangler fig“. Tento způsob byl vytvořen Martinem Fowlerem (7) a jeho myšlenkou je, že při udržování monolitického systému by jednotlivé funkcionality měly být inkrementálně přetvářeny do mikroslužeb. Klíčovým faktorem při migraci je určit způsob, jak starý systém nahradit mikroslužbami. Takto zásadní změna by neměla být provedena ukvapeně. Existuje mnoho věcí, které se mohou při pokusu o nahrazení starého systému nově vytvořenými službami pokazit. Při užití tohoto vzoru mohou být jednotlivé části monolitického systému postupně nahrazovány nově vytvořenými službami, přičemž monolitický systém bude z počátku stále zodpovídat za větší část fungování systému. Takto se budou jednotlivé části přidělávat, dokud nebudou všechny funkce systému plně nahrazeny nově vytvořenými mikroslužbami. Výhodou tohoto typu migrace je, že je postupná a dává nám možnost s ní ve kterémkoliv kroku přestat nebo jednoduše vrátit změny.



Obrázek 3. – Strangler fig pattern

Na obrázku výše můžeme vidět způsob, jak se tento návrhový vzor používá v praxi. V prvním kroku nalezneme určitou část v našem monolitu, kterou bychom chtěli vyjmout a přepracovat. V tomto kroku je potřebné dobře určit hranice části, kterou chceme vyjmout. Ve druhém kroku implementujeme naši mikroslužbu. Ve třetím kroku přesměrujeme volání z monolitické aplikace na nově vytvořenou mikroslužbu. Výhoda tohoto vzoru tedy spočívá nejen v možnosti vytvářet nové mikroslužby postupně, ale také v tom, že nově přidané funkcionality můžeme jednoduše vrátit do předešlého stavu pomocí přesměrování volání zpět na monolitickou aplikaci.

Tento způsob může být také využit při přidávání nové funkcionality. Například, když organizace potřebuje vylepšit funkcionalitu pro nějakou činnost, může při této příležitosti vytvořit celou novou mikroslužbu. Tímto přístupem se zamezí dalšímu růstu monolitu.

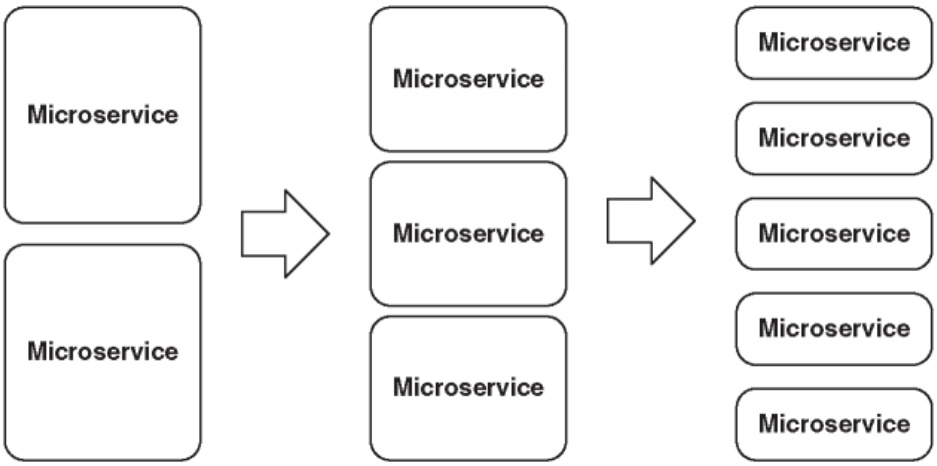
4.2 Nové vybudování systému

Tyto kapitoly popisují, jakým způsobem se dají podle Wolfa (10) budovat mikroslužby v projektu od počátku. Nejvíce ovšem klade důraz na správné rozdělení systému do jednotlivých domén při počátku jeho navrhování.

4.2.1 Začátek s velkými službami

Jedním způsobem, jak se vypořádat s problémy dělení mikroslužeb, je začít několika velkými službami, které potom dále můžeme rozdělovat na mikroslužby. Rozsah kódu pro jednotlivé mikroslužby by měl být stanoven podle schopnosti jednotlivých týmu kód spravovat. Na začátku, když v projektu existuje pouze několik mikroslužeb, je jednoduché se

vypořádat s jejich rozsahem, aniž by byl architekt zahlcen vnější složitostí. Systém může být tedy progresivně rozdělen z velkých služeb do mikroslužeb, což také umožní týmům pracovat nezávisle na sobě. Tento způsob je také dobrý z organizačního hlediska, protože jednotlivé týmy mohou být vytvářeny postupně. (10)



Obrázek 4. – Rozdělení velkých služeb

4.2.2 Začátek s malými službami

Dalším způsobem, jak začít, je rozdělit systém do mnoha malých mikroslužeb a použít tuto strukturu jako základ pro další vývoj. Tento způsob vyžaduje velmi detailní znalost obchodní logiky celého systému a také přináší velice komplexní vnější logiku, protože všechny mikroslužby spolu musí komunikovat. (10)

5. Migrace systému Damas

5.1 Popis a funkce aplikace

Platforma Damas byla navržena na základě hlubokých znalostí energetických trhů a dlouholetých zkušeností v oblasti zakázkového vývoje softwaru firmou Unicorn. Koncepce tohoto systému byla definována v souladu s nejmodernějšími trendy v oblasti vývoje informačních systémů jako systém založený na pravidlech (Rule-Based System). (13)

Damas představuje nástroj pro podporu obchodních procesů v energetice a prostředí, kde je možné tyto procesy v reálném čase modelovat a nasazovat přímo odborníky zákazníka bez účasti dodavatele. (13)

Možnosti modelování a konfigurace v prostředí Damas jsou zcela unikátní. Kromě procesního řízení jsou součástí modelu také datové struktury s libovolném časovým rozlišením (Time Series Engine), výpočetní výrazy (Calculation Engine), obrazovky systému – pohledy na data (FlexiGUI) v tabulkové a grafické podobě, komunikace různými kanály (EventsEngine) a rozhraní na okolní systémy (Enterprise integration). V neposlední řadě je zde také úzká integrace s kancelářskými nástroji (zejména MS Excel) a vestavěný reporting. (13)

5.2 uuApp Framework & The Architecture

Vývoj a architektura softwaru ve firmě Unicorn probíhá podle určitých pravidel. Tato pravidla jsou shrnuta a označována jako Unicorn Mobile-First IoT-Ready Cloud Architecture, zkráceně The Architecture, a stanovují návrhové vzory a klíčové principy pro vývoj aplikací. Jedná se o referenční architekturu, která si klade za cíl opakovanou použitelnost již vytvořených komponent. K implementaci The Architecture slouží uuApp Framework, což je soubor knihoven, nástrojů a procedur. (15)

Obrázky, které jsou použity v následující sekci, jsou vytvořeny pomocí jednoho z nástrojů uuApp Frameworku (15), který se jmenuje uuBML a používá se mimo jiné i pro modelování informačních systémů.

5.3 Zavedení architektury mikroslužeb do systému Damas

Jedním z cílů této bakalářské práce je zavedení architektury mikroslužeb do monolitického informačního systému. Tato část se zabývá možnými způsoby migrace a zároveň překážkami, které společně s migrací souvisí. Při psaní této práce bylo v aplikaci Damas implementováno několik mikroslužeb za pomoci různých technologií. V této části je popsána mikroslužba Computation Module pro zpracování sekundových dat a Energy Gateway, která umožňuje integraci výpočetního modulu se systémem Damas a zároveň připravuje systém pro integraci dalších mikroslužeb.

Damas je monolitický systém tvořený 20 moduly, postavený na architektuře .Net, jehož specifikum je vysoká konfigurovatelnost na základě tzv. metadat. Stále vyšší nároky na tento systém, primárně s ohledem na množství zpracovávaných dat a výpočty nad těmito daty, vedly k zavedení nových komponent, které jsou navrženy již v architektuře mikroslužeb.

Původní myšlenkou bylo postupně přetvořit celý systém do architektury mikroslužeb postupným nahrazováním jednotlivých modulů za mikroslužby. Tento způsob byl ale zavrhnut, protože v systému dochází k vysoké provázanosti mezi jednotlivými moduly.

Aktuální implementace počítá se zachováním stávajícího monolitického systému a přidáváním nových funkcí pomocí mikroslužeb. Tímto způsobem se přestane stávající část monolitického systému rozrůstat a postupem času se bude část, kterou v celém systému tvoří, zmenšovat. Firma konverguje k tomu, že se budou jednotlivé moduly přepisovat postupně za několik let a jednotlivé požadavky se budou iterativně upravovat na základě nově získaných zkušeností.

V nové části systému dochází ke změně komunikačního standardu. Monolitický systém komunikuje pomocí SOAP, zatímco novější část systému postaveného na mikroslužbách komunikuje pomocí architektury REST.

5.4 Důvody zavedení architektury a popis mikroslužeb

Damas je monolitický systém, který byl vytvořen v architektuře .Net. Jeho vysoká provázanost mezi jednotlivými moduly způsobuje složité přidávání nových funkcí. Zavedení mikroslužeb do systému má za cíl snížit míru provázanosti mezi jednotlivými částmi systému, a vytvořit tak lépe udržitelný systém. Dále má za cíl dosáhnout nezávislosti na technologii, ve které byl Damas vytvořen a získat větší možnosti škálování. V tomto případě je velmi velkým přínosem možnost horizontálního škálování, protože potřeba na výkon není stále stejná. Horizontální škálování nám umožňuje škálovat pouze potřebné části systému, a efektivně tak využívat přiřazené zdroje.

Důvod implementace modulu pro výpočty je splnění nároku na přesnější data klienta. Nejmenší časová jednotka, se kterou dokáže systém Damas operovat, je minuta. Nároky klientů na přesnost se neustále zvyšují a je potřeba do systému implementovat i možnost zpracování sekundových dat. To je výpočetně velmi náročné a v určitých situacích bude potřeba zvýšit výkon systému. Architektura mikroslužeb nabízí možnost horizontálního

škálování. Pro tento případ je tento typ škálování nejvhodnějším řešením, protože systém není pod zátěží celou dobu, a můžeme tak efektivně využít výpočetní zdroje.

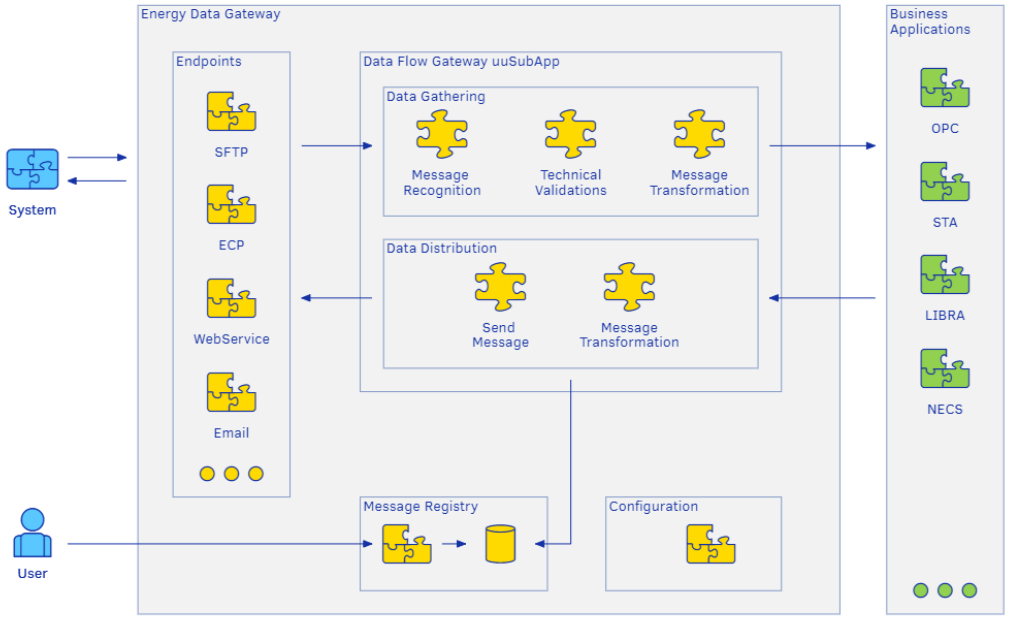
Důvod implementace gateway modulu je připravit prostor pro integraci dalších mikroslužeb a poskytnout jednotné API pro celý legacy Damas. Koncept gateway modulu je částečně vypůjčen z konceptu architektury orientované na služby a má za úkol řídit toky dat mezi jednotlivými mikroslužbami a odstínit detaily jejich implementace.

5.4.1 Energy gateway

Mikroslužba Energy Gateway pokrývá komponenty poskytující funkcionality vázané k výměně a ukládání zaslaných zpráv a poskytuje data interním aplikacím. Komponenty této mikroslužby jsou zodpovědné za výměnu a shromažďování dat, technickou validace obsahu příchozích zpráv a ukládání dat. Každá příchozí zpráva je nejdříve zkontrolována na základě definovaných technických pravidel, jako je jméno souboru a obsah souboru. Když je zpráva z technického hlediska zkontrolována, pošle komponenta zodpovědná za distribuci dat zprávu dále do systému, kde se odehrávají různé výpočty. V neposlední řadě zde také dochází k ukládání jednotlivých zpráv, které jsou následně skrze jednoduché rozhraní prezentovány uživatelům.

Energy gateway dále také slouží ke komunikaci mezi systémem Damas a externími systémy. Jednou z motivací pro vznik této mikroslužby bylo implementovat komunikaci systému Damas s externími systémy a integrace nových částí, vytvořených podle architektury mikroslužeb, s částí, která je napsaná v monolitické architektuře.

Damas je s Energy Gateway integrován pro synchronní komunikaci přes REST API a pro asynchronní pomocí front na brokeru. Způsob komunikace, který bude systém používat, je možné konfiguračně nastavit. Primární komunikačním protokolem Energy Gateway je REST API, a proto pro komunikaci směrem EGW -> Damas je použit tento protokol.



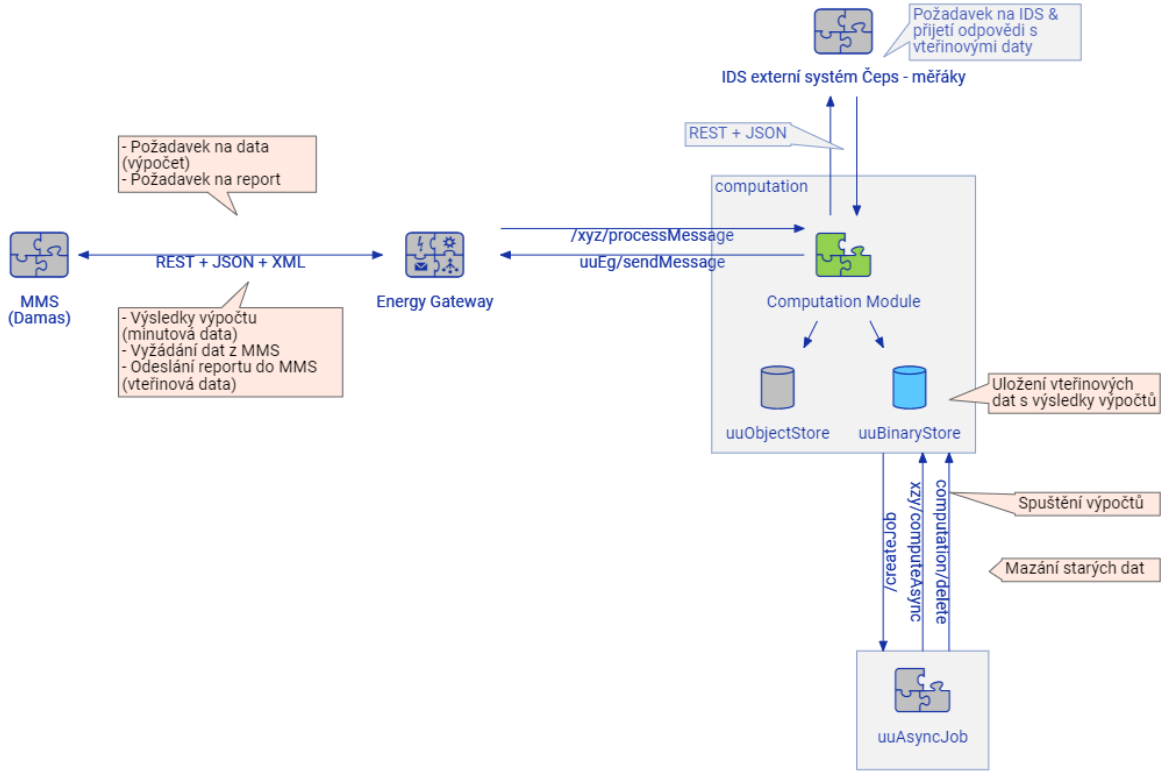
Obrázek 5. – Energy gateway

Na obrázku výše jsou popsány datové toky, které v mikroslužbě Energy Gateway probíhají. Do této mikroslužby přichází data od externích aplikací konfiguračně nastavenými koncovými body jako například email nebo SFTP. Tyto data se následně validují a transformují. Nakonec jsou transformovaná data odeslána do systému Damas, který je dále zpracovává.

5.4.2 Computation Module

Před začátkem popisu této mikroslužby je potřeba vysvětlit, co jsou to takzvané technické a byznysové mikroslužby. Technická mikroslužba je vytvořena velmi abstraktně, aby mohla podporovat různé typy byznysu, které Damas poskytuje. Příkladem jejího účelu je tvorba časových řad. Časová řada ukazuje data uspořádaná podle času nehledě na to, o jaká data se jedná. Byznysová mikroslužba je vytvořená pro konkrétní typ byznysu. Tyto pojmy jsou specifické pro projekt Damas a usnadňují orientaci v aplikaci. Energe Gateway je speciální typ mikroslužby, který zprostředkovává komunikaci, nejedná se tedy ani o technickou a ani byznysovou mikroslužbu.

Computation Module je nově vytvořená byznysová mikroslužba, která slouží ke zpracování sekundových dat z externího systému IDS. Tento systém poskytuje sekundová data o údajích z měřících přístrojů. Damas nedokáže s těmito daty pracovat, a proto byla vytvořena tato mikroslužba. Tato byznysová mikroslužba využívá technických mikroslužeb, kterými jsou například tsStore a tsCalc. Tyto technické mikroslužby zajišťují vytvoření časových řad a jejich uložení. Dále se zde vyskytuje mikroslužba uuAsyncJob, která spouští výpočty nad zpracovanými daty a maže data stará.

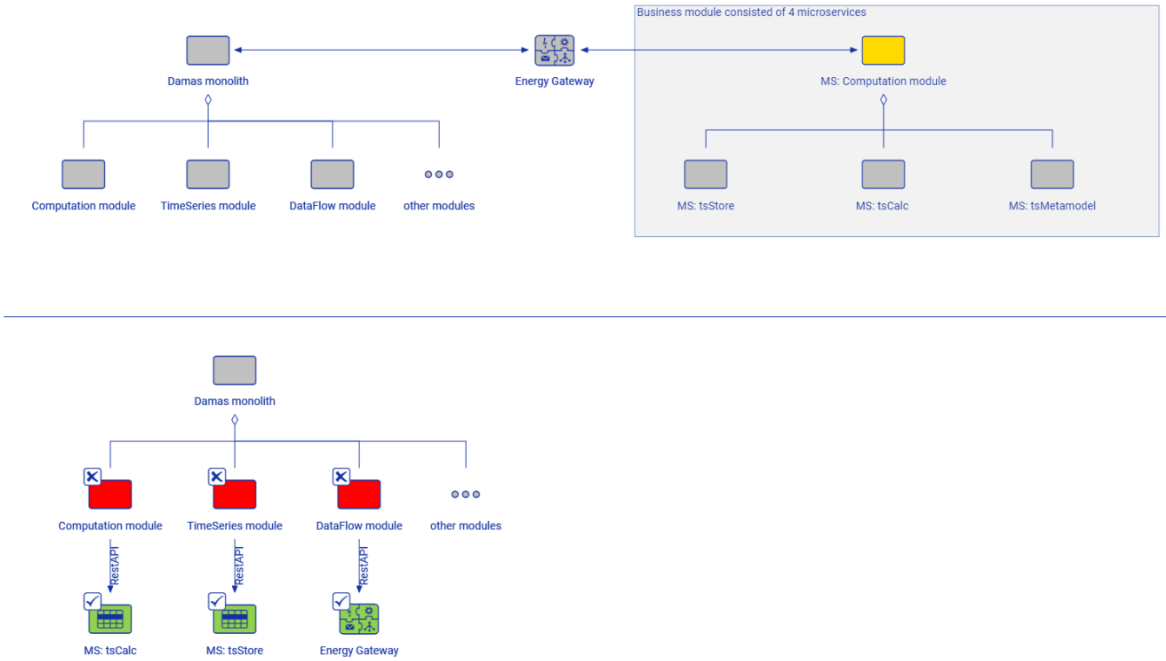


Obrázek 6. – Integrace mikroslužby

Na obrázku můžeme vidět integraci byznysové mikroslužby Computation Module se starou částí systému Damas a průběh požadavku na data. Zbytek systému Damas je v roli externího systému a může si definovat vlastní komunikační API. Zde bylo zvoleno API, které již v Damasu existovalo.

5.5 Průběh implementace

Před implementací bylo důležité určit, jakým způsobem budou jednotlivé mikroslužby do systému přidány. První možný způsob byl postupně odstranit jednotlivé moduly ze systému Damas a nahradit je nově vytvořenými mikroslužbami. Tento způsob je obecně možný, ale v tomto případě jsou jednotlivé moduly silně provázány a plně je nahradit by bylo téměř nemožné. Druhý způsob spočívá v ponechání původního systému a přidání nových mikroslužeb, které budou s tímto systémem komunikovat.



Obrázek 7. – Možnosti implementace

Na obrázku můžeme vidět oba možná způsoby přidání mikroslužeb. Způsob popsaný ve spodní části spočíval v tom, že jednotlivé moduly budou postupně nahrazeny a s aplikací Damas budou komunikovat přes REST API. Toto řešení by bylo velmi náročné, protože jednotlivé moduly jsou v aplikaci vysoce provázány a je velmi složité vyjmout jejich funkcionalitu z celé aplikace. Tento způsob migrace by mohl vytvořit silně provázané mikroslužby, a mohlo by tak dojít k vytvoření distribuovaného monolitického systému.

Aktuální způsob přidání mikroslužeb je popsaný ve vrchní části obrázku. Tento způsob spočívá ve vytvoření podpůrné mikroslužby Energy Gateway, která poskytuje jednotné API pro celý monolit Damas. Mikroslužby, které se následně přidávají, komunikují pouze s touto mikroslužbou, která následně přesměrovává jejich požadavky. V tomto případě

sledujeme přidání mikroslužby Computation Module, která má za cíl zpracování sekundových dat. Tato byznysová mikroslužba využívá dalších technických mikroslužeb ke svému fungování.

Pro možnost opětovného využití kódu při vytváření dalších mikroslužeb bylo potřebné nejdříve vytvořit jednotlivé technické mikroslužby, jako je například tsStore, tsCalc a tsMetamodel. Ty mají vždy vlastní databázi a bude docházet k jejich duplikaci, aby se zamezilo propojení mezi jednotlivými částmi systému a byly dodrženy základní principy architektury. Funkce těchto mikroslužeb jsou vytvořeny velmi abstraktně a jsou základem pro konkrétní typy byznysu, které Damas poskytuje.

5.6 Výzvy spojené s implementací

Největší výzvou byla velikost celé aplikace. Damas obsahuje již přes milion řádků kódu a na jeho tvorbě bylo stráveno více než 150 tisíc hodin. Jednotlivé moduly jsou tedy velmi objemné a bylo velice náročné vytvořit jednotlivé moduly jako mikroslužby. Další problém byl ve vysoké provázanosti jednotlivých modulů. Ty jsou totiž použity na mnoha místech v aplikaci, takže nebylo možné části jednoduše vyjmout a nahradit, jako je tomu u návrhového vzoru pro migraci Strangler Fig.

5.7 Shrnutí výsledků implementace

Vytvoření jednotlivých mikroslužeb způsobilo rozdělení členů projektu do jednotlivých týmů a začala se měnit celková struktura projektu. V monolitickém systému byla nutná koordinace všech členů projektu v testovací a nasazovací fázi, protože všechny části systému musely být vždy nasazeny současně. Nyní mají jednotlivé týmy zodpovědnost pouze za danou mikroslužbu a mají možnost svobodné volby při různých fázích vývoje. Dále také získaly možnost volby technologií, ve kterých budou danou mikroslužbu tvořit.

Nově vytvořené mikroslužby obsahují relativně málo kódu oproti monolitickému systému, a stávají se tak lépe udržitelnými. Díky dlouholetým zkušenostem v oblasti energetiky mohly být u jednotlivých mikroslužeb dobře určené hranice a rámec jejich funkcí. Tím došlo k naplnění kvalitativních atributů, jako je nízká provázanost a vysoká soudržnost.

Pro naplnění těchto atributů má každá mikroslužba vlastní databázi. Tato forma duplikace zvyšuje cenu provozu systému, ale zároveň ho dělá jednodušším pro celkové spravování a přidávání nových částí.

Díky struktuře nové architektury je nyní dále možné velmi efektivně škálovat nově vytvořenou mikroslužbu. Při velké zátěži může nyní dojít k vytvoření další instance pouze u jednotlivé části systému, a naopak při nízké zátěží mohou být jednotlivé instance ukončeny, aby se zamezilo zbytečnému využívání dostupných zdrojů.

6. Závěr

Cílem této bakalářské práce bylo:

- Nalézt limity architektury mikroslužeb v kontextu jejího zavádění do monolitického systému
- Ukázat, jakým způsobem může tato architektura vylepšit již existující monolitický systém

Architektura mikroslužeb se zdá být dalším krokem, co se týče vývoje korporátního softwaru. Navazuje na myšlenky, kterými se začali lidé zabývat již v 80. letech, kdy byly aplikace programovány se stále vyšším množstvím funkcionalit. Obsahovaly tedy velké množství kódu, který bylo potřeba roztřídit takovým způsobem, aby se všichni členové týmu v aplikaci vyznali a mohli efektivně pracovat.

Praktická migrace Damasu na architekturu mikroslužeb ukázala všechny potřebné věci, které se musí vyřešit před jejím začátkem. Ukázalo se, že mikroslužby je možné zavést do mohutného monolitického systému, který má vysoké závislosti mezi jednotlivými částmi. Jako způsob migrace bylo zvoleno postupné rozšiřování systému pomocí nově vybudovaných mikroslužeb, přičemž z počátku zajišťuje větší část poskytovaného byznysu monolitická část. Cílem této migrace byl požadavek na zvýšení přesnosti výpočtů u České elektroenergetické přenosové soustavy. Ačkoliv vytvořené mikroslužby ještě nejsou plně připraveny na produkční režim, ukazuje se, že tento krok vedl správným směrem.

V porovnání s monolitickým systémem je nyní možné při vyšším zatížení systému škálovat pouze část s výpočty. Zároveň došlo ke změně struktury celého systému. Jednotlivé mikroslužby jsou spravovány odlišnými týmy, jsou tvořeny v odlišných technologiích a mají své vlastní repozitáře se zdrojovým kódem. Vytvoření menších týmů a nezávislost na původní technologii se ukázalo být ve firmě Unicorn velmi užitečné. V předchozím monolitickém systému, kdy změna modulu ovlivnila spoustu funkcí, bylo potřebné znát logiku celého systému ještě před začátkem jeho vývoje. Po rozdělení do menších týmu došlo k efektivnějšímu přiřazování nových lidí na projekt, protože do menších aplikací se jednotliví vývojáři rychleji zařadí.

Závěrem je potřeba zdůraznit, že cílem této práce nebylo zavrhnout použití monolitické architektury, ale ukázat, že pro větší systémy se zdá být použití mikroslužeb vhodné, a poskytnout náhled na migraci.

7. Seznam zdrojů

1. NEWMAN, Sam. Building microservices. Sebastopol, CA: O´Reilly, [2015]. ISBN 978-1-491-95035-7.
2. NADAREISHVILI, Irakli, el al. *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, 2016. ISBN 978-1-491-95625-0.
3. Dragoni N. et al. Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham. (2017) https://doi.org/10.1007/978-3-319-67425-4_12
4. FOOTEN, John a Joey FAUST. *The service-oriented media enterprise: SOA, BPM, and web services in professional media systems*. Boston: Elsevier/Focal Press. ISBN 0240809777.
5. Singh, K., Çalışkan, M. and Mihályi, O. Java EE 8 Microservices. Packt Publishing, 2018. ISBN 978-1788475143
6. Microservices. martinfowler.com [online]. Copyright © Martin Fowler [cit. 15.07.2022]. Dostupné z: https://www.martinfowler.com/articles/microservices.html
7. NEWMAN, Sam. *Monolith to microservices: evolutionary patterns to transform your monolith*. Beijing: O'Reilly, 2019. ISBN 978-149-2047-841.
8. Goetsch, Kelly. *Microservices for Modern Commerce*. California: O'Reilly, 2017. ISBN 978-149-1970-874.
9. JOSUTTIS, Nicolai. *SOA in practice*. Sebastopol: O´Reilly, 2007. ISBN 978-0596529550.
10. Wolff, Eberhard. Microservices: Flexible software architecture. Boston: Addison-Wesley, 2017. ISBN 978-0-134-60241-7
11. EJSMONT, Artur. *Web scalability for startup engineers: tips & techniques for scaling your Web application*. New York: McGraw-Hill Education, [2015]. ISBN 9780071843652.
12. DRAGONI, Nicola, Ivan LANESE, Stephan Thordal LARSEN, Manuel MAZZARA, Ruslan MUSTAFIN a Larisa SAFINA. Microservices: How To Make Your Application Scale. In: PETRENKO, Alexander K. a Andrei VORONKOV, ed. *Perspectives of System Informatics* [online]. Cham: Springer International Publishing, 2018, 2018-01-18, s. 95-104 [cit. 2022-07-20]. Lecture Notes in Computer Science. ISBN 978-3-319-74312-7. Dostupné z: doi:10.1007/978-3-319-74313-4_8

13. uuDamasMMS – Application Model. Plus4U a.s., 2020 uuBookKit
14. BAKSHI, Kapil. Microservices-based software architecture and approaches. In: *2017 IEEE Aerospace Conference* [online]. IEEE, 2017, 2017, s. 1-8 [cit. 2022-04-26]. ISBN 978-1-5090-1613-6. Dostupné z: doi:10.1109/AERO.2017.7943959
15. uuApp Framework Knowledge Base - uuApp Framework. uuApp Framework Knowledge Base - uuApp Framework Knowledge Base [online]. Dostupné z: https://docs.plus4u.net/framework
16. Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T. Microservices migration patterns. Softw Pract Exper. (2018) https://doi.org/10.1002/spe.2608

8. Seznam použitých obrázků

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **Roman Šedivý**

Studium: I1900745

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název bakalářské práce: **Zavedení Microservice architektury v Legacy systému**

Název bakalářské práce AJ: Microservice Architecture Implementation in Legacy System

Cíl, metody, literatura, předpoklady:

Cíl:

- tranformace legacy monolitu na microservices architekturu
- návrh možností, PoC na existujícím modulu systému

Zadávající pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Datum zadání závěrečné práce: 26.1.2021