

# **VYSOKÁ ŠKOLA FINANČNÍ a SPRÁVNÍ**

Fakulta ekonomických studií

Studijní obor: **Aplikovaná informatika**

Navazující magisterské studium kombinované

Bc. Josef Rajský

**Událostně řízená mikroservisní architektura**

**Event Driven Microservices Architecture**

**DIPLOMOVÁ PRÁCE**

**Praha 2020**

Vedoucí závěrečné práce:

Ing. Luboš Dobda, Ph.D.

## **Poděkování**

Chtěl bych především poděkovat vedoucímu práce panu Ing. Luboši Dobdovi, Ph.D. za odborné vedení, cenné rady, inspiraci a velkou míru trpělivosti. Chtěl bych také poděkovat rodině, přátelům a kolegům za poskytnutou podporu během studia.

## **Prohlášení**

Prohlašuji,

že jsem tuto závěrečnou práci vypracoval/a zcela samostatně a veškerou použitou literaturu a další podkladové materiály, které jsem použil/a, uvádím v seznamu literatury a že svázaná a elektronická podoba práce je shodná. Současně prohlašuji, že souhlasím se zveřejněním této práce podle § 47b zákona č.111/1998Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Klepněte sem a napište datum \_\_\_\_\_

## **Abstrakt**

Hlavním tématem této práce je popis a zhodnocení výhod a nevýhod událostně řízené mikroservisní architektury a návrh aplikace na této architektuře založené. Teoretická část se bude zabývat popisem metodik vývoje softwaru a jejich využití pro různé architektury. V rámci rešerše budou srovnány mikroservisní, servisně orientovaná a monolitická architektura. Pro praktickou část práce, která bude obsahovat návrh aplikace, bude také provedeno srovnání rámců pro vývoj single-page aplikací pro tvorbu uživatelského rozhraní. Cílem praktické části je navrhnout a vytvořit aplikaci za použití zvolených technologií a navržení způsobu jejího napojení na informační systém podniku. Aplikace by měla podporovat podnikové procesy při řízení zaměstnanců, organizačních celků, plánovaní a řízení projektů.

## **Abstract**

The main topic of this thesis is to describe and evaluate the advantages and disadvantages of event-driven microservice architecture and to design an application based on this architecture. The theoretical part describes software development methodologies and their use for microservice architecture and other main architectures. The research compares microservice, service-oriented and monolithic architecture. The practical part of this thesis, which includes the design of the application also compares frameworks for the development of single-page applications at the frontend. The aim of the practical part is to design and create an application using selected technologies and to suggest a way of its connection to the company information system. The application should support business processes in managing employees, organizational units, and project management.

## **Klíčová slova**

Mikroservisní architektura, vývoj software, událostní řízení, mikroservis, .NET Core, C#, Docker, RabbitMQ, událostně řízené, CQRS, Agile, Scrum

## **Keywords**

Microservice architecture, Microservices, Software development, Event sourcing, .NET Core, C#, Docker, RabbitMQ, Event-driven, CQRS, Agile, Scrum

# **Obsah**

PODĚKOVÁNÍ.....	2
PROHLÁŠENÍ.....	3
ABSTRAKT .....	4
ABSTRACT .....	4
KLÍČOVÁ SLOVA .....	5
KEYWORDS.....	5
OBSAH .....	6
ÚVOD .....	9
<b>1 METODIKY VÝVOJE APLIKACÍ .....</b>	<b>12</b>
<b>1.1 Tradiční metodiky .....</b>	<b>13</b>
1.1.1 Vodopádový model .....	13
1.1.2 Spirálový model.....	15
1.1.3 RUP .....	16
<b>1.2 Agilní metodiky .....</b>	<b>18</b>
1.2.1 SCRUM .....	20
1.2.2 Extrémní programování (XP) .....	22
<b>1.3 Shrnutí k metodikám .....</b>	<b>24</b>
<b>2 MONOLITICKÁ ARCHITEKTURA (MA).....</b>	<b>26</b>
<b>2.1 Životní cyklus monolitu.....</b>	<b>27</b>
<b>2.2 Výhody a nevýhody .....</b>	<b>28</b>
<b>2.3 Příklad monolitického přístupu .....</b>	<b>30</b>
<b>3 SERVISNĚ ORIENTOVANÁ ARCHITEKTURA (SOA) .....</b>	<b>32</b>
<b>3.1 Webové služby.....</b>	<b>34</b>
<b>3.2 Komunikační rozhraní a protokoly.....</b>	<b>35</b>
<b>3.3 Procesy v SOA .....</b>	<b>36</b>
<b>3.4 Výhody a nevýhody .....</b>	<b>37</b>
<b>3.5 Servisně orientovaný přístup .....</b>	<b>39</b>
<b>4 MIKROSERVISNÍ ARCHITEKTURA (MSA).....</b>	<b>42</b>
<b>4.1 Servisy.....</b>	<b>43</b>
<b>4.2 Domény a doménově řízený návrh.....</b>	<b>45</b>
<b>4.3 Distribuovaný systém .....</b>	<b>47</b>

4.4	Service Discovery .....	49
4.5	Komunikace .....	50
4.6	Orchestrace a choreografie .....	54
4.7	Událostní řízení .....	57
4.8	API Gateway .....	62
4.9	Monitoring .....	63
4.10	Ekosystém .....	67
4.11	Výhody a nevýhody .....	68
5	TECHNOLOGIE V MSA .....	71
5.1	Docker .....	72
5.2	Kubernetes .....	75
5.3	Ocelot .....	76
5.4	Kafka .....	77
5.5	RabbitMQ .....	79
5.6	Swagger .....	80
5.7	HealthCheck .....	81
5.8	Uživatelské rozhraní .....	81
6	POROVNÁNÍ .....	86
6.1	SOA vs. MSA .....	86
6.2	MSA vs. Monolit .....	88
6.3	MSA a metodiky .....	92
7	APLIKACE V MSA .....	93
7.1	Role .....	93
7.2	Doménově řízený návrh .....	94
7.3	Scrum .....	96
7.3.1	Produktový backlog .....	96
7.3.2	Sprint .....	98
7.4	Vývojové prostředí .....	99
7.5	Založení mikroservisu .....	102
7.6	Struktura a komunikace .....	107
7.7	Události .....	110
7.7.1	Událostní řízení .....	111
7.7.2	Událostní Konzistence .....	112
7.8	Servis Gateway .....	114
7.9	Message Broker .....	114
7.10	Servis Monitor .....	115
7.11	Servis Event Store .....	115

7.12	Servis Kalendář.....	116
7.13	Prototyp .....	116
	ZÁVĚR .....	117
	SEZNAM BIBLIOGRAFICKÝCH ODKAZŮ .....	119
	SEZNAM PŘÍLOH .....	133
	SEZNAM TABULEK .....	133
	SEZNAM OBRÁZKŮ .....	134
PŘÍLOHA A	SCRUM – PRODUKČNÍ BACKLOG.....	136
PŘÍLOHA B	SCRUM - SPRINTY .....	147
PŘÍLOHA C	PRODUKČNÍ ROAD MAP .....	150
PŘÍLOHA D	REALIZACE SPRINTŮ 1-3.....	151
PŘÍLOHA E	ZÁKLADNÍ SERVIS .....	152
PŘÍLOHA F	SERVIS – UŽIVATEL API .....	155
PŘÍLOHA G	STRUKTURA PROJEKTU .....	164
PŘÍLOHA H	PŘÍKAZY A UDÁLOSTI, PUBLIKACE A ODBĚR .....	165
PŘÍLOHA I	SERVIS – GATEWAY.....	169
PŘÍLOHA J	SERVIS – MONITOR .....	172
PŘÍLOHA K	SERVIS – EVENTSTORE.....	176
PŘÍLOHA L	SERVIS KALENDÁŘ.....	179

# Úvod

Hlavním tématem diplomové práce je popis, zhodnocení výhod a nevýhod událostně řízené mikroservisní architektury v porovnání s dalšími architekturami. Cílem teoretické části je porovnání technologií směřujících a podporujících přístup k vývoji softwaru s využitím mikroservisní architektury.

Praktická část práce využívá teoretických poznatků při návrhu a realizaci projektu na vytvoření aplikace. Projektovaná aplikace, nazvaná Kancelář, využívá mikroservisní architektury v rámci evidenčního systému docházky, aktivit, úkolů a jednotlivých činností na úkolech v návaznosti na poskytnutí podpory pro zpracování mzdových podkladů. Ve finálním stavu se předpokládá její využití pro přímou podporu podnikových procesů v oblasti řízení personálu organizace, projektů a finančních prostředků. Cílem je přenést získané poznatky z úvodního porovnání do praktického prostředí a demonstrovat tak vhodnost využití organizacích, které chtějí realizovat, udržovat a dále rozvíjet obdobné systémy vlastními zdroji.

Mikroservisní architektura, která využívá událostního řízení, reaguje na zasílané příkazy k operacím a podle nich dále generuje a publikuje události, které dále odesílá do systému. Na publikované události reagují další prvky, které se k jejich odběru přihlásily, a podle informací z nich získaných dále reagují a vydávají další události. Ke konzumaci lze libovolně prvky přidávat nebo odebírat, aniž by se narušil samostatný chod ostatních částí. Systém tak na základě jednoho příkazu reaguje různým počtem událostí a mění tak svůj stav. Protože je systém řízen událostmi a ty jsou ukládány jako doklad do jednoho centrálního správce, který drží hlavní pravdu o stavu systému, je celý systém označován jako událostně konzistentní [78].

Způsob udržení událostní konzistence v mikroservisním prostředí má výhody, ale i některé nevýhody, které jsou dále v textu zhodnoceny. Ve standardním pojetí zpracování dat jsou většinou vytvářeny tabulky v relačních databázích, kde jsou záznamy provázány a drženy pouze jejich aktuální stavy. Bez využití dalších mechanismů není možné zjistit a dále analyzovat jakým způsobem se do aktuálního

stavu dostali, nebo v jakém stavu byly v určitém časovém bodě. Dochází tím k nevyužití a ztrátě zajímavé a v mnoha ohledech hodnotné informace [94].

Jednou z výhod využití událostního řízení, ukládání událostí a udržování událostní konzistence, je možnost sledovat historický vývoj entity v libovolném bodě její existence. Veškeré operace jsou zaznamenávány od jejího vzniku až po aktuální stav. V případě potřeby je pak možné provést rekonstrukci pro vybraný časový bod nebo úsek na základě aplikace událostí realizovaných v tomto časovém úseku. V základu architektury a principům událostního řízení tak je způsob zpracování, kdy nedochází ke ztrátě informace a je možné analyzovat vývoj o stavu zaznamenané skutečnosti[39].

Jako problém se může jevit způsob zpracování na úrovni transakcí tak, jak je praxí v relačních databázích. V distribuovaném systému mikroservis je zajištění jednotnosti, konzistence, izolovanosti a trvanlivosti operace. U událostně řízené mikroservisní architektury jsou události konzumovány a operace prováděny prvky systému, které jsou ohraničeny v tzv. doménovém kontextu. Mikroservisy nevytvářejí vazby na okolí přímo, mají vlastní datový zdroj. V případě výpadku může dojít k ohrožení prováděné komplexní operace a nelze dodržet bezpečně operaci v rámci transakce. S touto komplikací je z objektivních důvodů nutné při návrhu aplikace počítat a zajistit v této oblasti dostatečné kontrolní mechanismy [39][58][77].

Výhodou zavedení kontrolních mechanismů u událostního řízení při konstrukci mikroservisně založených aplikací je využití tohoto principu jako možného způsobu obnovy a migrace dat. V případě nekonzistence systému, nutnosti obnovení nebo změny platformy aplikace je možné pouze dosadit do systému požadovanou, logicky setříděnou a po sobě jdoucí, množinu událostí a systém sám provede rekonstrukci do požadovaného stavu všech entit [58][77].

Mikroservisní architektura je určena k zajištění efektivity vývoje a rychlosti zpracování změn. Díky principům samostatnosti je možné rozdělit prvky systému do dvou částí. První část tvoří zejména ty, které jsou zaměřeny na publikaci a druhou část ty, které jsou určeny především na konzumaci událostí. Obdobně se

rozdělují i funkce pro zápis a čtení stavu. Princip odděleného zápisu a čtení je označován jako CQRS (Command Query Responsibility Separation). Při uplatnění principu CQRS jeden prvek konzumuje zadávané příkazy, provádí operace zápisu a vydává události. Protože operace zápisu nejsou tak časté, nemusí být u tohoto prvku navýšen výkon. Jiná služba pak může konzumovat výsledky, i z více událostí, skládat je dohromady a tvořit pro své potřeby materializované pohledy. Na operace čtení může být naopak obrovský nápor a lze u ní separovaně navýšit požadovaný výkon bez nutnosti rozšiřovat všechny prvky včetně těch méně využívaných [39][58][77].

V teoretické části práce je provedeno porovnání mikroservisní architektury s jinak běžnou, monolitickou a servisně orientovanou, architekturou z pohledu přínosů pro rozvoj informačního systému podniku. Porovnání bylo provedeno na základě parametrů využitelných pro podporu vývoje nových projektů, pro údržbu a provoz stávajících systémů a případného napojení na systémy další. Porovnání dává pohled na způsob, jakým způsobem je možné reagovat na požadavky moderního způsobu rozvoje a reakcí organizací na rychle se měnící prostředí a zvyšující se časové nároky těchto reakcí. Pro praktické využití je také v úvodu zpracováno porovnání metodik použitelných při vývoji aplikací.

V rámci praktické části práce je navržena struktura a realizace modulu pro správu personálu s využitím událostně řízené mikroservisní architektury. Hlavní myšlenkou nově navržené aplikace je zpřehlednit a usnadnit evidenci a plánování aktivit personálu organizace, a dále podpořit pracovní procesy, včetně schopnosti lépe reagovat na případné organizační změny a požadavky. Výsledkem správné formy realizace a nasazení by měla být mimo jiné i úspora času pro zaměstnance na podpůrných aktivitách a získání času na hlavní činnost. Pro účely projektu je provedeno porovnání metodik vývoje softwaru a vybrána jedna agilní metodika na které je demonstrováno vhodnost využití v projektech společně s mikroservisní architekturou. V aplikaci je zařazeno několik různých technologií.

Souhrnným cílem a přínosem práce je získání přehledu o vlastnostech a rozdílech jednotlivých servisně orientovaných architektur, jejich využití při tvorbě projektů a v obecné rovině i pravidla a způsob jakým se podobné projekty řídí.

# 1 Metodiky vývoje aplikací

Úvodní část této práce je nejdříve zaměřena na porovnání tradičních a agilních metodik využívaných při vývoji aplikací. Vývoj aplikace má typické znaky projektu, jako jsou cíl, čas, jedinečnost a zdroje. Je to sada procesů, která je stanovenými cíli ohraničena a je potřeba se v těchto mezích držet.

Otázka, jak stanovit správné cíle, jak jich postupně dosahovat, jak analyzovat požadavky, jak získat představu o čase, který bude potřeba, a jaké zdroje a kdy je využít je důvod, proč využít některou z prověřených metodik.

Metodiky jsou nápomocné pro provedení správné analýzy požadavků, zpracování cílů a jakým způsobem pak k těmto cílům směřovat. Udávají směr, kterým se má vývoj projektu ubírat, jak využívat zdroje, kdy je využít a na které oblasti problémů si dát pozor a neopomenout je.

Každá složitější aplikace, která je vyvíjena jedním vývojářem nebo celým týmem, potřebuje mít na začátku projektu zpracovaný návrh a vizi čeho má ve dosáhnout. Pokud je vše podloženo dobrým plánováním na základě metodik, bude pak vývoj efektivní a nedojde k opakování chyb z minulosti [47].

Při prvních počátcích vývoje softwaru v 70. a 80. letech to bylo neprobádanou oblastí s velkými očekáváními se spoustou neznámých. První průkopnické začátky byly živelné a z dnešního pohledu chaotické. S rychlým nástupem a rozvojem technologií dospěla oblast vývoje softwaru ke krizi. Charakteristickými znaky tohoto období bylo neúnosné prodlužování a pozdržování projektů, nízká kvalita programů, velká složitost údržby a tvorba inovací, špatná produktivita práce programátorů[47][84].

U takto neefektivního vývoje bylo těžké odhadovat, jak bude projekt probíhat nebo kdy vůbec dospěje do svého úspěšného konce. Neřízený vývoj přinášel mnoho problémů. Mezi hlavní patřila špatná komunikace v rámci týmu nebo mezi zadavatelem a zpracovatelem projektu, nesprávný přístup k vývoji SW, špatné

návrhy a plánování, nízká produktivita práce, neznalost pravidel nebo podcenění hrozeb a rizik spojených s technologií nebo interakcí s okolím [48]..

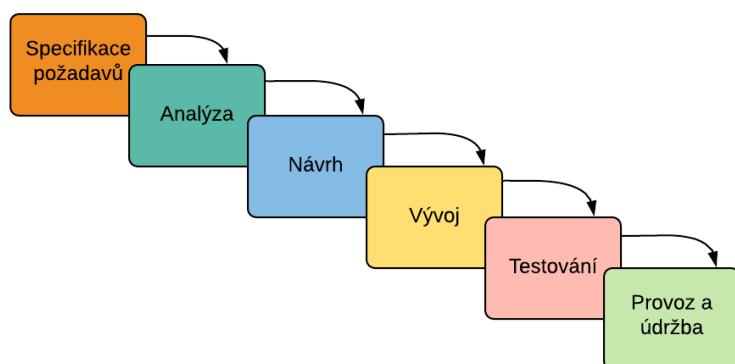
Z nutnosti řešit tyto problémy se postupně začal vyvíjet obor SW inženýrství a tím i základy prvních metodik. Definovaly se cíle tohoto vědního oboru ve snaze o usměrnění a stanovení pravidel činností v rámci projektů vývoje aplikací. Prvními pokusy o stanovení takových metodik byly návrhy americké organizace PMI, která vydala dokument, který obsahoval zatím nejlepší dosažené poznatky z praxe, a na tomto principu pak vznikaly další metodiky [45][47].

## 1.1 Tradiční metodiky

Z tradičních metodik jsou v této práci uvedeny modely **Vodopádový, Spirálový Rational Unified Process (RUP)**.

### 1.1.1 Vodopádový model

Tento model je nejznámější a také nejstarší z tradičních metodik. V oblasti vývoje softwaru se tento model objevil už kolem roku 1970. Model rozděluje postup prací na projektu do několika fází, které na sebe postupně navazují. Grafické znázornění připomíná vodopád (Obrázek 1 Vodopádový model), a proto se od něj odvozuje i samotný název. Model rozděluje celek na fáze **specifikace požadavků, analýzy, návrhu, vývoje, testování a integrace, nasazení do provozu a údržbu** [91][11][40][69][84].



Obrázek 1 Vodopádový model Zdroj: [40], vlastní úprava

Tyto fáze na sebe logicky navazují, a teprve až po ukončení jedné fáze může být zahájena další.

**Specifikace problému** je nejdůležitější částí projektu, během níž zjišťujeme u zákazníka, co bude předmětem řešení, informace o prostředí a jakých přibližných cílů je potřeba dosáhnout. Nejedná se v této fázi o jasnou a přesnou specifikaci, ale především o sběr všech potřebných údajů k hlubší analýze [91][40].

**Analýza** všech získaných poznatků o problému pokrývá také důkladnou specifikaci problému, stanovení jasných požadavků uživatele na cílové řešení, stanovení termínů a způsobů komunikace. Důležité je přesně porozumět záměru řešení, jinak se výsledek může lišit a projekt bude neúspěšný. V této fázi je nutné vše důkladně zdokumentovat a všechny strany seznámit s výsledky analýzy. Jedná se tedy o přesný popis zadání a cíle, nikoliv konkrétních technologií využitých k jeho dosažení [91][40].

**Návrh** projektu se zabývá konkrétním výběrem technologií a návrhem architektury, která bude pro realizaci řešení daných problémů nevhodnější. Zahrnuje otázky jako jaké bude mít cílový produkt rozvrstvení, jaké moduly bude obsahovat, jaké bude uživatelské rozhraní a jakou formou bude prezentovat své výstupy. Výstupem této fáze bude odsouhlasení zvolené technologie, architektury, jasná definice cíle projektu a cenová a časová kalkulace [91][40].

**Implementace** by podle předpokladu tohoto modelu zahrnovala pouze přetvoření požadavků v realitu bez nutnosti hlubšího přemýšlení. Předpokládá se, že dokumentace bude naprosto přesně a jasně specifikovaná jako kuchařka a vývojový tým jen provede automatizaci. V případě nutnosti provedení změn by se vše mělo vrátit do první fáze specifikace a celý postup analýzy a návrhu by se musel provést znovu [91][40].

Pro **integraci a testování** existuje velké množství způsobů a metodik. Podstatou této části je se nějakým způsobem přesvědčit, že všechny části aplikace jsou funkční přesně tak, jak bylo požadováno, a projekt dosáhl svých stanovených cílů a řeší zadaný problém. V případě nalezení nesrovonalostí jsou poznatky vráceny na začátek

procesu a provede se jejich analýza, opětovné odsouhlasení projektu a jejich zpracování. Proto je pro tento případ důležitá kvalitní dokumentace z úvodních fází, aby se určila odpovědnost za odchýlení se od cíle [91][40].

**Provoz a údržba** je závěrečnou fází projektu. Aplikace bude uvedena do produkčního prostředí a předpoklad je ten, že po provedení finálního nastavení bude aplikace přesně splňovat specifikované vlastnosti z úvodu projektu. V případě nesrovnalostí pak záleží na domluveném servisu a provedení úprav ještě v rámci údržby nebo se návratu na začátek celého procesu [91][40].

Vhodnost vodopádové metodiky je pro jasně specifikované požadavky na produkt, které se během vývoje příliš nemění. V současném prostředí je takový stav spíše utopie, a proto se tato metodika pro svou nepružnost nevyužívá v takové míře [91][40].

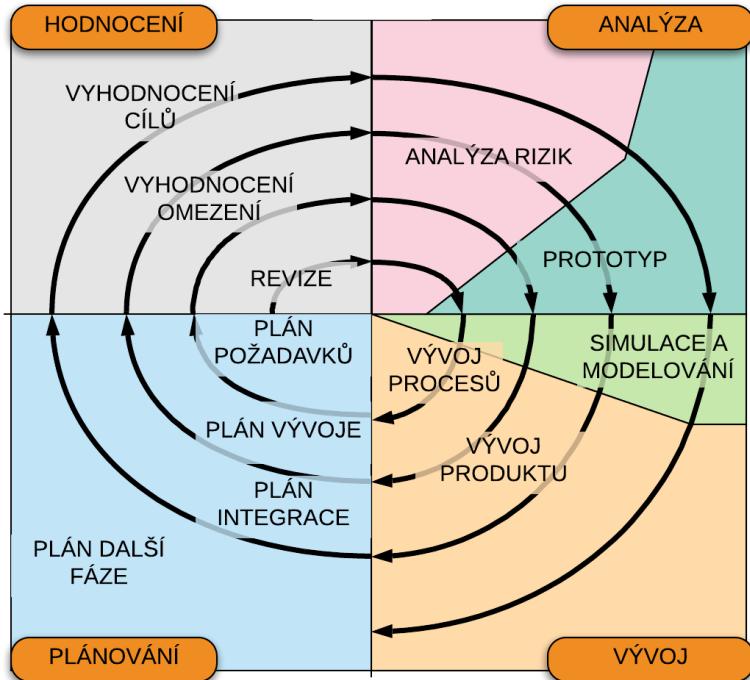
### 1.1.2 Spirálový model

První návrhy spirálového modelu vydal v roce 1985 Barry Boehm. Návrh, který byl založen na jeho zkušenostech vývoje softwaru na základě předchozích metodik, odrážel potřebu lepší reakce na změny v průběhu realizace projektu novativním přístupem v tomto modelu je představa, že jsou práce na projektu prováděny v cyklech. Cílem je odbourat snahu o úplnou specifikaci požadavků v úvodní fázi a místo toho vybrat a zpracovat to nejdůležitější a po uvedení do praxe provést kontrolu a další vylepšení. Základem je vydat co nejdříve funkční část, která je pak při každé iteraci cyklu zdokonalována[91][11].

Součástí jednoho cyklu vývoje je také velký důraz na analýzu rizik. Rizika odráží všechny aspekty oboru informatiky z pohledu zhodnocení nákladů, splnění legislativních požadavků, zachování konkurenční výhody, spolehlivosti a výkonnosti hardwaru. Tato analýza rizik zhodnotí na začátku každé fáze, jak plánovaná činnosti ovlivní tyto oblasti [91][11].

Tyto cykly tvoří spirálu a smyslem je v každé iteraci vytvořit nějakou přidanou hodnotu k projektu. V horizontální rovině je cyklus rozdělen na fáze **plánování, hodnocení, analýzy a vývoje** (Obrázek 2 Spirálový model). Ve vertikálním směru

je pak cyklus aplikován pro fázi konceptu, specifikace, návrhu, designu, implementace, testu a vydání produktu.



Obrázek 2 Spirálový model Zdroj: [91], vlastní úprava

Spirálový model má široké uplatnění v celém oboru informatiky i mimo něj. Například je součástí standardů pro ISMS (systému řízení bezpečnosti informací), kde hraje hlavní roli v zavádění celého systému v organizaci a analýza rizik je jeho stěžejní součástí. Tento spirálový sebezdkonalující efekt je označován obecně jako cyklus PDCA nebo také jako Demingův sebe-zlepšovací cyklus [91].

Hlavním přínosem a rozdílem oproti vodopádovému modelu je to, že reaguje pružně a efektivně na případné změny požadavků v průběhu projektu a lépe tak odráží vývoj v reálném prostředí informační společnosti.

### 1.1.3 RUP

Rup je zkratka z The Rational Unified Process, byla původně vyvinuta společností Rational Software Corporation, kterou v roce 2003 odkoupilo IBM, které dál metodu podporuje. Základní význam této metodiky je soustředit nejlepší a komerčně

úspěšné poznatky z praxe na jednom místě a poskytnout tak kvalitní znalostní základnu pro zpracování dalších projektů ve vývoji softwaru. V tomto smyslu metoda neobsahuje konkrétní využitelný model, ale spíše množství specifických metod, které lze využít přímo pro konkrétní projekt.

Při metodice RUP je projekt rozdělen od několika zpracovávaných vláken, které jsou zavedeny souběžně na začátku projektu. Dále je projekt rozdělen do jednotlivých fází vývoje rozdelených do krátkých iterací vývoje. Práce jsou prováděny postupně v jednotlivých fázích, kdy je každé vlákno zpracováváno s různou intenzitou. Vlákna tak na sebe nečekají jako ve vodopádovém modelu, navzájem se překrývají a je jim věnována různá pozornost ve všech probíhajících fázích. Vlákna jsou rozdělena na oblast **modelování procesů, zpracování požadavků, analýzy a návrhu, implementace, testování, nasazení do provozu, konfigurace/řízení změn a organizaci prostředí** (Obrázek 3 Schéma RUP) [51][48].

**Modelování procesů** je zahajovací fází projektu, kdy se poznává oblast, na kterou bude vývoj aplikace zaměřen a jsou stanoveny základní parametry a možnosti prostředí [68][5].

**Zpracování požadavků** se provádí téměř souběžně s mapováním prostředí. Po sběru požadavků jsou strukturovány a dále je sledováno jejich plnění až po koncový produkt [68][5].

**Analýza a návrh** je oblastí zpracovávající získané požadavky, vyhodnocující jejich smysluplnost a možnosti realizace v prostředí. Po jejich analýze požadavků je možné navrhnout plán implantace [68][5].

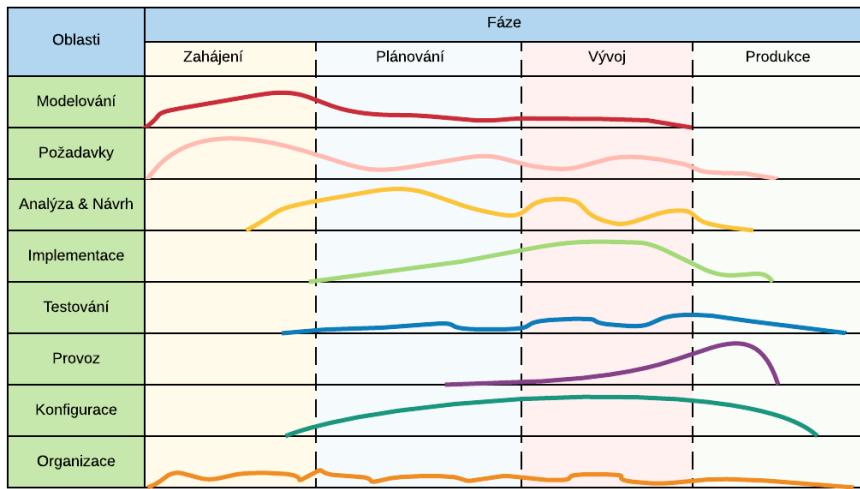
**Implementace** je hlavní oblastí procesu, který generuje hlavní hodnotu. Na základě analyzovaných požadavků a prostředí je prováděn vývoj produktu [68][5].

**Testování** se provádí v různé míře od fáze plánování, kdy jsou společně s vývojem produktu plánovány testy, které jsou následně ve fázi implementace dále rozvíjeny a před uvedením do provozu provedeny [68][5].

**Provoz** je plánován souběžně s probíhajícími aktivitami v oblasti implementace testů, proces je chystán a největší pozornost je mu logicky věnována v závěru projektu [68][5].

**Konfigurace** a řízení změn je činnost probíhající v různé míře během od implementační fáze až do fáze provozu včetně. Reaguje na poznatky získané během těchto aktivit a provádí jejich zapracování do projektu [68][5].

**Organizace** prostředí je zaměřena na zajištění činnosti a organizace vývojového prostředí a podpory všech týmů provádějících činnosti na projektu [68][5].



Obrázek 3 Schéma RUP [51], vlastní úprava

V metodice je vhodnou praxí udržení postupného iterativního vývoje, kdy je kladen důraz na to, aby vývoj probíhal v jednotlivých krocích a v každém bylo hlavní úsilí zaměřeno na ty nejdůležitější problémy a oblasti. Řízení požadavků má jako hlavní úkole kvalitně zpracovat dokumentaci, popsat funkcionality a provést rozhodnutí o realizaci na základě analýzy. Vhodné je zapojení kontrolních a vyhodnocovacích mechanismů pro jednotlivé fáze projektu. Díky zpětné vazbě a poučení se z ní lze zabránit propagaci a opakování chyb po zbytek průběhu projektu. Malá opomenutá chyba v úvodu projektu má v pozdějších fázích zásadní následky a její odstranění vyžaduje řádově vyšší úsilí [68][51][48].

## 1.2 Agilní metodiky

Druhou kategorií metodik vývoje aplikací jsou ty, které označujeme jako agilní. Základní myšlenkou těchto metodik je aktivní přístup k vývoji a sledování změn, jejich rychlá akceptace a zapracování v projektu. Nepostupuje se ve zdlouhavých nepřetržitých krocích směrem k cíli s dlouhými intervaly produkce výsledku, ale více

se spoléhá na krátké svižné cykly, častou komunikaci se všemi zúčastněnými stranami a rychlou prezentaci, byť i ne úplně dokončeného produktu. V každém průchodu cyklem je prováděna analýza, zda byly požadavky splněny, využívá se zpětná vazba zákazníka a s pomocí okamžitého zapracování všech připomínek se vývoj posune o malý přírůstek směrem k cíli. Tyto cykly se opakují až do splnění hlavního cíle [84].

Agilní metody vznikly na základě potřeby reagovat na časté změny prostředí a nároky už během vývoje aplikací. Projekty označované jako agilní jsou charakterizovány častými a malými přírůstky přidávanými k aplikaci, plánování a realizace dílčích cílů ve velmi krátkých cyklech a práce v malých autonomních týmech s jasně ohraničeným zadáním a výstupem pro každou iteraci cyklu [111][11][93][48][84].

Během jednotlivých fází projektu vývoje jsou aplikovány změny podle požadavků zákazníka nebo v reakci na změnu prostředí. Agilní metodiky tak reagují mnohem pružněji na současné nároky, kvalitu a rychlosť vývoje. Cílem je, aby při dokončení výsledného produktu byly co nejvěrněji splněny aktuální požadavky uživatele aplikace [1][84].

Se zrychlujícím se vývojem technologií byly a jsou tradiční metodiky pro některé projekty ne zcela efektivní. Na vývoji agilních metodik se zásadním způsobem podílí komunita, která je sdružena pod „Agile Alliance“ a která definovala základní požadavky formou Manifestu agilního vývoje [1].

**Manifest agilního vývoje** definuje v několika bodech důraz na agilní metody před tradičními. Udává směr, kterým by se vývoj měl ubírat a jakých zásad by se měl držet, aby byl stále efektivní a schopný reakce. Klade důraz na individuálnost a vzájemnou interakci před zaběhlými procesy a nástroji. Funkcionalitu softwaru před vytvářením obsáhlé dokumentace, spolupráci se zákazníkem před zdlouhavým vyjednáváním o podmínkách dodávky, reakci na změny před dodržováním plánu za každou cenu [1][84].

Podle cílů definovaných v tomto manifestu by měl být každý projekt především zaměřený na potřeby zákazníka, rychlé dodání funkční části produktu za účelem

generování hodnot a postupném přidávání dalších. V jakékoliv části by projekt reagoval a zpracovával případné změny a udržoval tak konkurenční výhodu.

Důležité je udržení akčnosti a motivace vývojového týmu, otevřenosti novým myšlenkám a přístupům nejen z oblasti informatiky a zajištění efektivní komunikace v rámci všech zapojených rolí na projektu. Hlavním měřítkem je funkční software bez nutnosti vytváření obsáhlých dokumentací. Toho lze dosáhnout vytvářením jasné čitelného a sám o sobě vypovídajícího kódu a vložením velké důvěry vývojovým týmům. Nutné je podporovat a udržovat technickou vyspělost a lehkost v produkci během jednotlivých cyklů a nastavení vhodné velikosti cílů nebo intervalů opakování [1].

Mezi nejznámější metodiky, které označujeme jako agilní, jsou například **Scrum**, **Extrémní programování**, **Kanban**, **Feature-Driven Development**, **Lean Development**, **Crystal**, **Dynamic Systém Develeplment Method**, **Test driven development** a mnoho dalších známých i méně známých metodik [1][84].

V této části bych pro potřeby této práce vybral a popsal první dvě agilní metodiky.

### 1.2.1 SCRUM

Jako metodika je zmíněn v roce 1986 na v článku v časopise Harvard Business Review pod názvem „The New New Product Development Game“ [83].

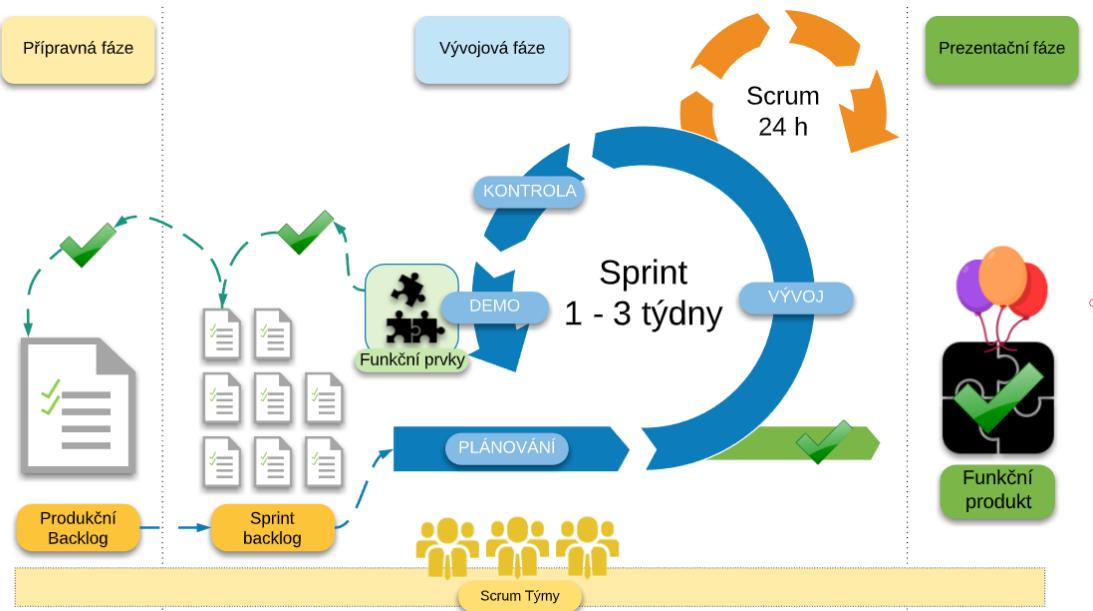
Do oblasti vývoje software zpracoval tuto metodiku Ken Schwaber a Mike Beedl v roce 1995. Oba prošli několika společnostmi a týmy, v nichž se podíleli na vývoji softwaru, a na základě svých zkušeností a poznatků se pokoušeli vyvinout flexibilní metodiku, která dokáže lépe reagovat na často se měnící požadavky při vývoji softwaru, a způsob zpracování těchto nových požadavků do výsledné aplikace [48][104][85].

Název **Scrum** je odvozen z terminologie amerického fotbalu, který označuje stav rozehry míče. Obdobně jsou tedy rozdeleny činnosti mezi aktéry v rámci projektu realizovaném metodikou Scrum. Vytvoří se několik Scrum týmů, mezi které se pak přidělují úkoly v rámci jedné rozehry. Proces zpracování je rozdělen do přípravné fáze, vývojové fáze a prezentační fáze. V přípravné fázi je připraven Produkční

backlog. Ve vývojová fáze probíhá v jeden až tří týdenních iteracích, ve kterých se realizuje plánování, vývoj, kontrola stavu a představení výsledku v podobě funkčního dema. V rámci každého sprintu pak probíhají v rámci týmu na stejném principu denní rozehry kolem stanoveného cíle, tedy Scrum [104][23].

Projekt je zahájen tvořením **Produkčního backlogu**, tedy dokumentu obsahujícího seznam prioritizovaných, ohodnocených a slovně popsaných očekávání. Slovní popis požadavku je označován jako „**Story**“ (Příběh). Je v něm uveden, kdo tento cíl požaduje, popis očekávané funkcionality s odůvodněním, proč cíl realizovat [83][93][104].

Na základě zpracovaného Backlogu se dále cíle rozdělí mezi produkční týmy, které vypracují obdobně Sprint backlogy pro jednu iteraci a všech fází sprintu pak provádí denní iterace kolem cílů ze svého Sprint backlogu. Vývojové týmy jsou složeny přibližně ze šesti členů v rolích vlastníka (Product owner), vedoucího (Scrum master), vývojáře (Team member) a běžného uživatele (Stake holder) [83][93][104][85].



Obrázek 4 Schéma Scrum[100], vlastní úprava

V rámci každého Sprintu má tým za úkol dosáhnout splnění stanoveného počtu bodů, které získává splněním ohodnocených cílů, které si uvedl do svého Sprint backlogu. Na konci každé iterace je pak představen splněný funkční celek v podobě

dema. Splněné sprint backlogy se pak vyhodnocují a zapisují zpět do Produkčního backlogu a postupuje se tak dlouho dokud nejsou splněny všechny definované cíle a je představena kompletní aplikace [48][85].

Týmy si v rámci jedné iterace Scrumu pro přidělené úkoly vytváření grafický přehled, kde dále rozdělují přidělené úkoly dělí story na konkrétní úkoly, které postupně realizují a mění jejich stav. Každý z týmu si vybere úkol, na kterém chce pracovat a označí si ho například svou barvou (Obrázek 5 Scrum – úkoly).



Obrázek 5 Scrum – úkoly Zdroj: vlastní tvorba

Bodová náročnost a rychlosť sprintů se vyhodnocuje v každé iteraci, dokud se nenajde optimální míra. V případě, že tým nesplnil očekává množství bodů, přenáší si cíle do dalšího sprint logu. Pokud neměl tým se splněním dané bodové hranice problémy, do další iteraci si pomyslnou cílovou pásku v podobě bodů opět o něco navýší. Tím je dosaženo optimálního a rychlého vývoje aplikace, kdy je pravidelně předávána zpětná vazba zadavateli a uživatelům, kteří okamžitě vidí, zda jsou jejich splnění naplněna. Krátké iterace a udržení zákazníka v procesu zvyšuje jeho nadšení a spokojenosť při představení finálního produktu [83][85].

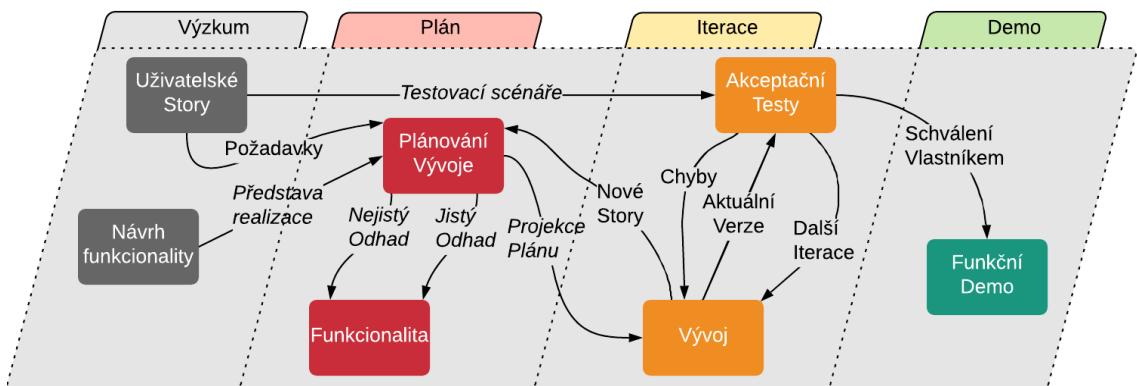
### 1.2.2 Extrémní programování (XP)

Vznik XP se datuje do roku 1999, v němž představil zakladatel XP, známý expert na vývoj softwaru a na softwarové metodiky Kent Beck, jeho základní myšlenky. Extrémní programování je metodikou vhodnou pro malé a středně velké týmy, které potřebují pružně reagovat na často se měnící podmínky řešeného problému.

Obecně lze přístup této metody popsat jako velký důraz na praktické a svěží uvažování o způsobech, jak postupovat při vývoji softwaru [8][48][93].

Činnost je plánována ve velmi krátkých cyklech a iteracích tak, aby co nejvíce vyhovovala zúčastněným členům vývojového týmu složeného z vývojářů a vlastníků procesu. Vývojovým týmem je ponechána kreativita a možnosti vlastní realizace a představy o vytváření nových funkcionalit, který by bylo možné využít. Činnost vývoje probíhá technikou párového programování, vzájemné kontroly kódu, podrobně zpracovaný návrh je upozaděn a je preferováno vytvoření funkčního celku. Když je nalezen způsob spolupráce s nejvyšším přínosem, je na tato kombinace dále rozvíjena do maximálního využití až extrému. Proto tedy extrémní programování. Využívá se komparativních výhod a schopností každého jednotlivce podílejícího se na projektu [8][91][23][23].

**Projekt je zpracováván ve fázích vývoje, plánování, iterace a představení funkčního dema.**



Obrázek 6 Schéma XP Zdroj:[108], vlastní úprava

Ve výzkumné fázi jsou popsány požadavky formou strukturovaného příběhu ze strany uživatele. Uživatel volnou formou popíše svou představu pro co nejlepší popsání situace produkčnímu týmu a ty pak provedou analýzu. Ve výzkumné fázi je zároveň nabízena zajímavá funkciálnita, kterou by bylo možné využít [23].

Plánování se zaměřuje na analýzu uživatelských požadavků, vytvoření plánu jejich implementace a provedení odhadu nejistého odhadu před zahájením plánování a jistého odhadu po provedení plánu [108].

Zpracovaný plán je předán do krátké iterace, kde se provede vývoj, revize formou naplánovaných akceptačních testů, zda jsou splněny požadavky definované ve story. V případě nedostatků se provedou další krátké iterace [108].

Po dokončení vývojového cyklu se představí zpracovaná funkční část, která je odsouhlasena vlastníkem oblasti projektu.

Pracovníci jsou spokojeni a motivováni, že dělají přesně to, co je baví a co jim jde. Manažeři jsou nadšeni rychlými přírůstky a pracovním nasazením a zákazník dostane svůj produkt v kratším čase [8].

Hlavními rysy extrémního programování jsou pravidelná a častá zpětná vazba, krátké iterace v cyklu plánování, využití testovacích mechanismů v kódu, častá verbální komunikace a evoluční přístup k vývoji projektu s vědomím, že situace může být proměnlivá, a je tudíž nutné počítat s drobnými úpravami a změnami směru a metod. Posledním parametrem je velká míra vzájemné kooperace při vytváření kódu [8][91][108].

### 1.3 Shrnutí k metodikám

Metodiky pomáhají při vývoji softwaru obsáhnout komplexní nároky, které jsou na něj kladený a pomáhají udržet nevhodnější postup směrem k cíli.

Agilní metodiky a jejich definice zprostředkovaná v Agilním manifestu dávají vývojářům více než jen technologický rámec o tom, jak postupovat, ale především inspiraci a návod k nastavení svého přístupu a myšlení tak, aby nezůstávali v zaběhnutých kolejích. Schopnost vnímat věci z více stran je klíčovou schopností adekvátní reakce na proměnlivé prostředí, kterému současný vývoj musí čelit. Tato schopnost je také zásadní pro zvolení a návrh vhodné architektury na začátku projektu. Každé zadání projektu má různé specifické požadavky, každý vývojový tým má různou velikost a schopnosti. Někdy je lepší postupovat velmi tradičně a jindy jsou potřeba zásadní inovace a změna [111][11][93][48][8].

Následující část práce se zabývá popisem architektonických rámců vývoje aplikací, v nichž se v různé míře uplatňují zmíněné metodiky. V některých je vhodné využití

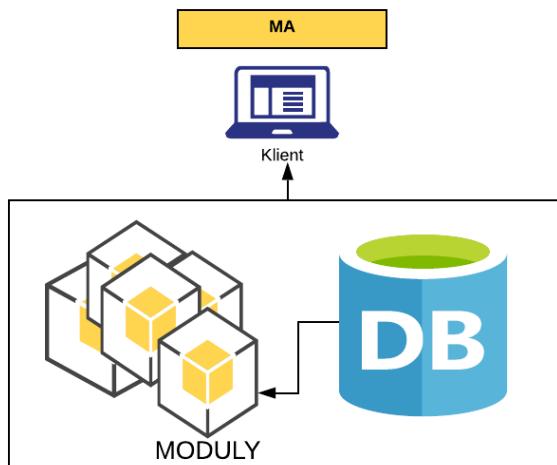
tradičních přístupů, v jiných je nutné postupovat více agilně. Rozdíl ve vnímání a myšlenkových přístupech je například u monolitické architektury a architektuře založené na mikroservisech. Pro to, jakou metodiku, v jaké architektuře a fázi projektu zvolit je potřeba rozhodnutí na základě kombinace jejich vlastností [40][69][51][48].

Vlastnosti	Tradiční			Agilní	
	Vodopád	Spirála	RUP	SCRUM	XP
<b>Sekvenční zpracování</b>	Ano	Ano	Ano		
<b>Rychlosť</b>			Ano	Ano	Ano
<b>Iterace</b>		Ano	Ano	Ano	Ano
<b>Flexibilní</b>				Ano	Ano
<b>Pevný proces</b>	Ano	Ano	Ano		
<b>Reakce na změny</b>		Ano		Ano	Ano
<b>Dlouhodobé plánování</b>	Ano	Ano	Ano		
<b>Zdokonalování</b>		Ano		Ano	Ano
<b>Pravidelné výstupy</b>				Ano	Ano
<b>Dokumentace</b>	Ano	Ano	Ano	Ano	
<b>Komunikace zákazníka</b>				Ano	Ano

Tabulka 1 Porovnání metodik vývoje software

## 2 Monolitická architektura (MA)

Nejčastějším způsobem návrhu softwaru je i v dnešní době stále způsob návrhu aplikací monolitickým způsobem. Souhrnné označení pro tento způsob tvorby přišel především s příchodem konkurenčního návrhu v podobě servisně orientované architektury a mikroservisní architektury. Název architektury je odvozen od představy, že je aplikace rozměrově velká, celistvá a po částech těžko uchopitelná jako kamenný monolit [28][99][80].



Obrázek 7 Monolitická architektura Zdroj: vlastní tvorba

Konstrukce aplikací navržených jako monolit se převážně rozděluje podle třívrstvého modelu. Nejobecnějším rozdělení je na klientskou, serverovou a datovou část. Podle rozložení hlavní výpočetní práce se aplikace rozděluje na serverové (server-side) nebo uživatelské (client-side), přičemž se takové rozdělení často označuje jako tlustý a tenký klient. V tomto případě je klientem myšlena část aplikace na straně uživatele [28][58].

Podrobnější dělení modelů třívrstvých architektur vychází z použitých technologií, podle způsobu přístupu k databázi a podle vzájemného provázání klientské a serverové částí z pohledu zobrazení uživatelského rozhraní.

Tímto způsobem navržené aplikace mají pevný cyklus návrhu realizace, provedení sestavení a nasazení do provozu. Pro vývoj takových aplikací je nejrozšířenějším způsobem vodopádový model nebo jiné tradiční techniky návrhu softwaru.

Jednotliví vývojáři nebo vývojové týmy přispívají do společného zdroje kódu (dále codebase). Po úspěšném uvedení aplikace do produkčního prostředí může dále probíhat vývoj nových funkcí a vydávání nových verzí, ale také přichází na řadu práce na údržbě aplikace a řešení případných chyb ve funkčnosti [28][58].

## 2.1 Životní cyklus monolitu

Vývoj aplikace v monolitické architektuře je na začátku projektu ta nejrychlejší a nesnadnější varianta. Rozdíly v rychlosti jsou v závislosti na zvolené metodice a propracovanosti návrhu, modelu a následné implementaci do prostředí organizace. V ideálním případě aplikace splňuje přesně očekávání zadavatele a požadavky stanovené na začátku projektu. Má jasně popsané funkcionality, zpracovanou dokumentaci do posledního detailu a je připravena podávat maximální výkon v oblasti pro kterou byla vytvořena. Svět a prostředí se neustále vyvíjí a organizace musí držet krok, přizpůsobovat se. Musí rozvíjet své schopnosti a měnit své vnitřní procesy, aby obstála v konkurenčním prostředí [79].

Systémy a aplikace musí na tyto změny reagovat, nebo přestanou být aktuální a budou nahrazeny. Pokud aplikace nevyhovuje, může být nahrazena jinou, konkurenční aplikací. Proto je znakem úspěšné aplikace schopnost změny. Z toho důvodu se naplánuje další iterace vývoje podle metodiky, zpracují se požadavky a opět se do aplikace přidá několik řádek kódu [79].

Aplikace se stává stále komplexnější a mohutnější s řadou nových a vzájemně provázaných funkcionalit, úprav, oprava i při sebelepší metodice, kvalitní dokumentaci a dodržování nastavené kultury při psaní kódu se s narůstající komplexitou roztáčí spirála chaosu a neudržitelnosti a komplikace se postupně nabízí jako sněhová koule. Kvalita dokumentace všech metod, využití code repository s komentovanými příruštky (commity) nebo kvalitně zpracované vnitřní testování tento jev zpomalí, oddálí, ale nezabrání mu [80].

Po čase stojí vývojový tým před výzvou, jak provést další úpravy, aniž by poškodil celek. Otázkou je v takovém případě čas. Jednou možností je využít zkušenosť kmenových zkušených pracovníků na projektu, nastudování dokumentace

a analýza všech možných vazeb, implementace nové funkcionality, provedení kontroly a testování, zadokumentování změny, nasazení do produkčního prostředí a v závěru krátká modlitba za úspěch. Druhou možností je začít běh od začátku, využít zkušenosti a vytvořit novou verzi a vymanit se tak na nějaký čas ze sevření monolitického pekla [79].

## 2.2 Výhody a nevýhody

Hlavními výhodami monolitické architektury jsou **jednoduchost, Testovatelnost a správa zdrojů** [99][58].

**Jednoduchost** je výhodou především při návrhu aplikace, zahájení nového projektu a v jeho prvních fázích. Zároveň je monolitické zpracování podporováno řadou nejběžněji užívaných nástrojů [58].

**Testování** lze provádět jako celek pomocí zakomponovaných testů při startu aplikace nebo při jejím nasazení do produkčního prostředí. Protože je aplikace v jednom balení a vnitřní komunikace probíhá přímým voláním metod, je možné provádět testování a procházení kódu snadněji a tím i lépe odhalit případnou chybu [58].

**Správa zdrojů** spočívá ve snadném navýšení zdrojů jednoho zařízení a nasazení aplikace jako jednoho celku do produkčního prostředí. Vždy se na jeden server připraví jedna instance celé aplikace [58].

Celá řada nevýhod se pak odvíjí právě od principu monolitické architektury a návrhu aplikace. Některé výhody se s postupujícím časem vývoje a větší velikostí aplikace postupně mění v zásadní nevýhodu.

Mezi hlavní nevýhody se řadí **složitost údržby, velikost aplikace jako celku, zapouzdření, konflikt zdrojů, spolehlivost, technologická svázanost a komplexnost** [99][58].

**Složitou pro údržbu** se aplikace stává při vzniku vývojového týmu, příchodu nového člena nebo provedení výraznějších změn v pozdější fázi projektu, je nutné provést zpětnou

analýzu a je složité odhalit a pochopit všechny vazby tak, aby nedošlo k jejich ovlivnění chybným zásahem.

**Velikost aplikace** ovlivňuje dobu pro nasazení do provozu. Čím větší velikost, tím delší dobu trvá samotná komplikace, zasazení do produkčního prostředí a start aplikace v případě nutnosti načíst všechny referenční knihovny. Jsou tak limitovány možnosti opakovaného nasazení aplikace do produkčního prostředí v krátkých vývojových cyklech [58].

**Zapouzdření** je hlavní problém pramenící už z návrhu monolitu. Vše v aplikaci je součástí jednoho kompaktního celku, a pokud chceme upravit sebemenší drobnost v kódu nějaké méně významné funkce, je nutné celou aplikaci znova zkompilovat a znova ji celou nasadit do produkčního prostředí [58].

**Konflikt zdrojů** nastává v případě, při němž si různé moduly v aplikaci konkurují při využití jednoho stejného zdroje produkčního prostředí. V tomto úzkém místě pak dochází k výraznému zpomalení celé aplikace. Zároveň v případě rozdělení práce na projektu do několika týmů dochází ke vzájemnému ovlivnění a provázanosti. To zvyšuje nároky na organizaci práce a tím i rychlosť vývoje [58].

**Spolehlivost** nebo spíše nespolehlivost je následný problém zapouzdření. Jeden špatně navržený proces může vyčerpat zdroje celého produkčního prostředí a způsobit tak nefunkčnost aplikace a kvůli malé chybě tak úplně omezit uživatele [58].

**Technologie** pro vývoj aplikace jsou vybrány na samotném začátku projektu a v průběhu budování nebo po nasazení aplikace do provozu už je změna technologie nereálná nebo velmi obtížná. Bylo by nutné aplikaci celou předělat od začátku [58].

**Komplexnost** je problém, který se objevuje v pokročilejších stádiích projektu, pokud je zapojeno větší množství vývojářů, týmů nebo při snaze zařadit agilní metodiky vývoje softwaru. Při velkém objemu funkcí a vzájemných vazeb dochází ke konfliktům odpovědnosti za některé prvky v podobě konkurenčních úprav databáze nebo naopak rozšíření některého z prvků systému, o kterém už nikdo neví, k čemu vlastně sloužil [58].

## 2.3 Příklad monolitického přístupu

Pro ukázkou monolitického návrhu si představme Prahu jako jednu velkou, monoliticky navrženou a fungující aplikaci.

V Praze je mnoho významných historických památek, které budeme vnímat jako funkce poskytované svým zákazníkům, v tomto přirovnání turistům. Památky jako Pražský hrad, Karlův most nebo Vyšehrad jsou nedílnou součástí celku Prahy, ale jsou na sobě svou funkcí pro uživatele nezávislé.

V rámci této aplikace nutně přijde moment, kdy například na Karlově mostě bude nutné vyměnit jednu sochu, tedy změnit jednu z procedur v jedné z knihoven aplikace. Pokud taková potřeba vznikne v reálném světe a reálné Praze, je optimální omezit práce pouze na tuto sochu, krátkodobě ji zakrýt a provést výměnu bez nutnosti zasahovat do celkového chodu města. Událost je to významná, všichni o ní vědí, ale návštěvníci nejsou příliš omezeni a funkce Prahy jako atraktivního města není zásadně omezena.

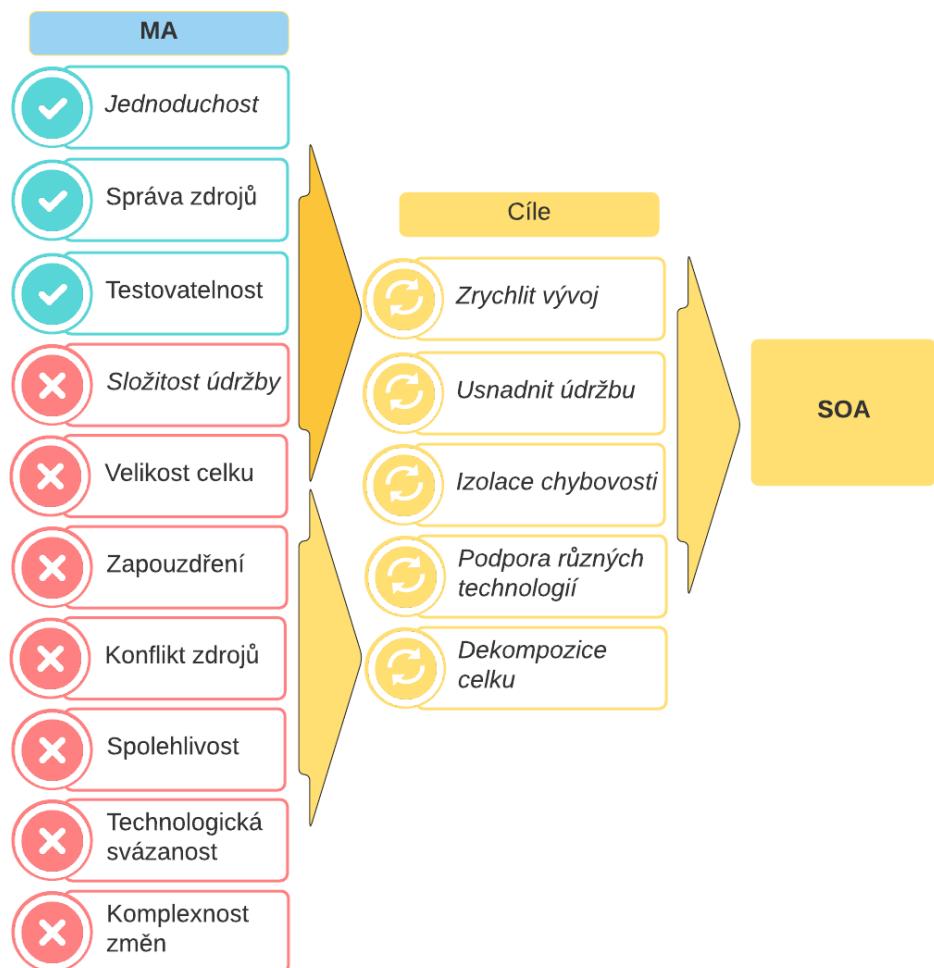
Co by znamenala oprava jedné procedury v aplikaci, která je v produkčním prostředí a je intenzivně používána? Připravil by se produkční build nové aplikace, upozornili se všichni uživatelé, že bude celá aplikace z důvodu údržby mimo provoz, pro jistotu by byla udělána záloha funkčního buildu a záloha databáze a pak by se aplikace nasadila.

Pokud bych tento postup převedl na zmíněný příklad výměny sochy, znamenalo by to, že vyšleme všem turistům zprávu, že bude celá Praha kompletně uzavřena po určitou dobu pro všechny turisty, protože se bude vyměňovat jedna jediná socha na Karlově mostě. Takové řešení není ideální pro pověst města a určitě by nevyvolalo obecné nadšení.

Pokud v reálném světě existuje způsob, jak provést údržbu a neomezit všechny funkce, musí to jít i v tom virtuálním při návrhu aplikace.

**Je třeba rozdělit celek na jednotlivé části** a ty se pokusit spravovat autonomně tak, aby neomezily funkce ostatních. V našem příkladu by to znamenalo, že Karlův most,

Pražský hrad a Vyšehrad budou spravovány samostatně, a tudíž může přes jistá omezení Praha dále pro turisty **poskytovat servisy**.



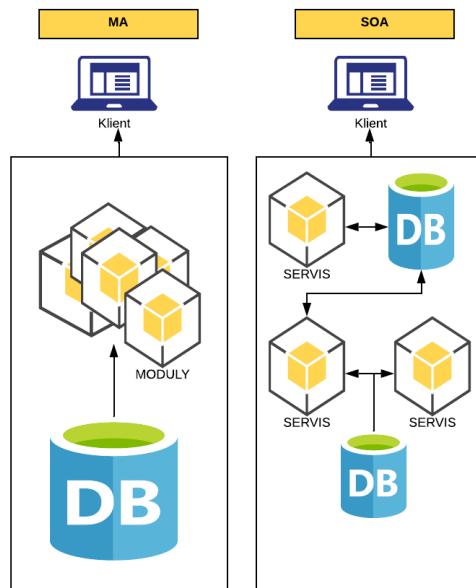
Obrázek 8 MA na SOA, Zdroj: vlastní tvorba

V případě aplikace ji tedy rozdělíme do několika servisů podle oblasti zaměření. Pokud nebude fungovat jedna z nich, ostatní mohou i nadále poskytovat svou funkcionality, a aplikace tak bude v omezené míře a za určitých podmínek stále v provozu. Takovou aplikaci označujeme jako **servisně orientovanou**.

### 3 Servisně orientovaná architektura (SOA)

Jedním ze základních principů servisně orientované architektury (service oriented architecture, SOA) je rozdelení celku na menší, vzájemně spolupracující prvky. Jak moc velké by takové stavební kameny aplikace měly být, není přesně stanoveno [63][75][34][13].

Každý vývojář, vývojový tým nebo projekt má jiné preference na způsob a velikost jednotlivých komponent a uplatňuje jinou míru rozdrobení systému označovanou jako granularita (Obrázek 9 Struktura MA, SOA) [63][75][34][13].



Obrázek 9 Struktura MA, SOA, Zdroj: vlastní tvorba

Pro komponenty se používá pojem **služba** nebo **servis**.

Hlavními znaky servis jsou **malá velikost, vymezení funkční oblasti, nezávislost na ostatních komponentách, možnost konzumace svého datového zdroje, popis svého rozhraní a funkcí ve srozumitelném formátu a poskytování svých funkcí okolí** [36][56].

Servisně orientovaná architektura má různé definice a vysvětlení v několika směrech a úrovních abstrakce.

**SOA** je sada komponent, které mohou být zavolány a které mají popsané rozhraní, které je veřejné a je možné ho nalézt [36][109].

**SOA** je způsob, jak stanovit pravidla vzájemné komunikace několika softwarů [90] [109].

**SOA** je architektura, která umožňuje budování aplikace pomocí agilních metod [96].

**SOA** je architektura orientovaná na byznys a podporu podnikových procesů a činností [90][109].

**SOA** je systém umožňující přiřadit zdroje na základě aktuálních potřeb, přičemž prostřednictvím sítě poskytuje tyto zdroje dalším entitám, které jsou schopny se k poskytovaným službám připojit. To umožnuje menší technologickou a logickou svázanost na úrovni databáze [81].

**SOA** je vzor, jakým způsobem organizovat a poskytovat schopnosti různých služeb, od různých vlastníků a poskytovat jednotné rozhraní pro nalezení služeb, napojení na ně a získání potřebných dat v předepsané očekávatelné struktuře [54].

Servisně orientovaná architektura je tedy nejen návrhový vzor, ale obecně i filozofie, jakým způsobem řídit a koncipovat podnikové procesy podle jednotného standardu za účelem získání větší dynamiky a rozvoje podnikových procesů [63].

Jednotlivé služby pak vzájemně komunikují prostřednictvím sítě. Aby byly schopné se vzájemně dorozumět, jsou jejich aplikační rozhraní popsány pomocí protokolů zakotvených v této architektuře. SOA je složena z několika rolí a komponent [63][75].

**Consumer** – Konzumentem je služba, interní komponenta nebo externí entita, která konzumuje poskytované prostředky.

**Provider** – V roli poskytovatele je ta služba, která nabízí své rozhraní konzumentům.

**Directory / Repository** – Adresář a Repozitář pro správu a poskytování informace o službách na základě požadavku klienta nebo klientské aplikace.

Klíčovou součástí je také **ESB – Enterprise Service Bus**, který má za úkol integrovat jednotlivé komponenty do databázové vrstvy na jedné straně a na druhé zajistit spojení mezi službami a **front-endem**, částí aplikace určené pro zobrazení zpracovaných dat v uživatelském rozhraní aplikace [44][65][13][2].

Aby bylo možné rozdělit jeden velký celek monolitické aplikace nebo vytvořit nový pomocí samostatných služeb, je potřeba jasně stanovit, jak bude komunikace probíhat. Co když budou jednotlivé komponenty zpracovány odlišnými technologiemi, několika různými vývojovými týmy, s přidanými externími produkty nebo dokonce bude potřeba napojení na systém dodavatele?

Aby nedocházelo k chaosu, je potřeba stanovit jednotný jazyk, kterým se všechny služby mohou najít, spojit a komunikovat. Jednotlivé služby proto nabízí své funkce pomocí standardů a protokolů jako WSDL a SOAP [2][65][91][109].

### 3.1 Webové služby

Základní stavební kameny servisně orientované architektury jsou webové služby (Web Services) [6].

Webové služby jsou malou monolitickou aplikací a ve společné formaci poskytují své služby a pro uživatele vytvářejí zdání jedné aplikace. Využívají standardů webového prostředí pro komunikaci se svým okolím. Každá taková služba splňuje podmínu, že je samostatně běžící, spravuje jednu oblast činností, je zapouzdřená jako jeden celek vůči svému okolí a není datově závislá na jiné entitě [3][6][35][35].

Tento způsob dělení aplikace na části a vymezení hranic odpovědnosti přináší hned několik výhod, který by v případě monolitické aplikace nebyly tak výrazné, nebo dokonce vůbec nebyly možné. Můžeme využít agilních metod vývoje softwaru a pro každý tým stanovit odpovědnost za jednu určitou službu, která bude mít jasné vstupy a výstupy a jasně popsané rozhraní. Způsob aplikace servisně orientované architektury tak více odráží potřeby reálného prostředí [6][41][75][86].

## 3.2 Komunikační rozhraní a protokoly

Aby samostatná a ohraničená služba, která si spravuje své zdroje a může existovat sama o sobě, byla využitelná ve větším systému, je nutné stanovit pravidla.

Stejně tak jako různí lidé mluví různými jazyky, i služby mohou být vytvořeny různými technologiemi, které si navzájem nemusí rozumět. K tomu, aby si služby v rámci propojeného systému rozuměly, jsou definovány standardy jazyka, jakým se budou bavit, a stejně tak konvence, jak se budou navzájem představovat a jak se o sobě vlastně dozví [3].

Architektura vytvořená v síťovém prostředí a založená na technologii webových služeb musí mít nastavena jasná pravidla komunikace a jednotný jazyk [28][75].

Část služeb využívá protokoly **WSDL** a **SOAP** popsané ve standardu XML jako svou vizitku pro další zájemce, konzumenty služby. Dále využívá protokoly **BPEL**, **WS-I**, vytváří **kontrakt** a využívá pro komunikaci **REST** [28][3][12][35].

**WSDL - Web Service Description Language** je veřejná vizitka webové služby.

Poskytuje model a formát XML pro popis služby. Je to prostředek, kterým se služba může zájemcům představit a dát jim vědět, jakým způsobem je možné ji využít, k čemu je určená, jaké funkce poskytuje a kde si je možné z ní požadovaná data převzít [43][41][65][91].

**SOAP - Simple Object Access** udává standard jazyka, kterým je možné se navzájem dorozumět, a slouží k tvorbě zpráv, kterými servisy komunikují. Stanovit takový univerzální jazyk v dynamickém světě informačních technologií je velký úspěch, kterého SOAP dosáhl. SOAP je jednotný jazyk, předávající vizitku WSDL, která je strukturovaná jako XML [41][6][65][91][109].

**BPEL - Business Process Execution Language** je standard umožňující uživateli popsat procesy byznysu a strukturou a použitím webových služeb, jejich vzájemné vazby a jak mají jako celek fungovat za účelem dosažení stanovených úkolů [65][91][35].

**WS-I – Web Services Interoperability** stanoví a podtrhává nároky, podle kterých by měl WSDL a SOAP při své komunikaci postupovat a které by měl splňovat. Podle toho, jaké úrovně naplnění požadavků podle WS-I služby dosáhnou, jsou zařazeny

do jednotlivých profilů. Stanovuje tedy něco jako úroveň dospělosti služeb a určuje jejich důvěryhodnost pro služby, které by s ní chtěli spolupracovat. Tento protokol je ve správě společnosti se stejnojmenným názvem, která byla dříve součástí IBM a nyní OASIS [2][44][91].

WSDL a SOAP jsou rozšířené a užívané standardy. Tyto dva komunikační standardy jsou dále rozšířeny o **UDDI** a **WSIL** [2][3][6][35].

**UDDI – Universal Description, Discovery and Integration** definuje způsob uveřejnění a vyhledání informací o webových službách. Jedna složka standardu je založena na protokolu SOAP a definuje, jakým způsobem komunikovat s registrum, který je obsahem druhé části [2][43][65][44][91].

**WSIL – Web Services Inspection Language** obsahuje také vyhledávací mechanismy, které UDDI doplňují. Není pak nutné využívat registr jako u UDDI, ale je možné se přímo dotádat na poskytnutí požadované služby u poskytovatele služeb [44].

**Rest - Representational State Transfer** slouží jako jeden ze způsobů komunikace prostřednictvím HTTP protokolu. REST bude v této práci více popsán v části mikroservisní architektury [49][12][65][66][91].

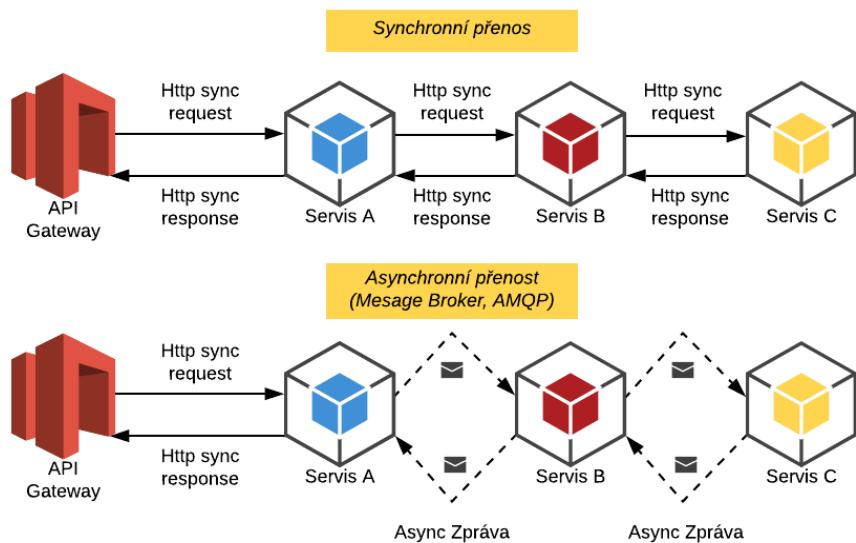
### 3.3 Procesy v SOA

Základním procesem v SOA je komunikace na principu **mechanismu požadavek/odpověď**. Klient odešle požadavek směrem k systému a ten zachytí ESB, který jej zpracuje, nalezne adresu služby, které se týká, a provede směrování. Služba požadavek přijme, zpracuje a vypracuje odpověď, kterou odešle směrem k ESB. Odpověď může obsahovat návratová data nebo pouze informaci o tom, zda byla operace provedena úspěšně nebo neúspěšně [46][34][56].

Způsob volání ze strany klienta je stejný jako při volání své vlastní metody – zavolá metodu a čeká návratovou hodnotu – s tím rozdílem, že je zde vypracování provedeno vzdáleně. V případě **synchronního volání** je klientský proces blokován, dokud se nevrátí odpověď. Protože se zpracování provádí vzdáleně, bude celková rychlosť procesu limitována kapacitou přenosového média a zatížením vzdáleného

zdrojového prostředku. V případě selhání zdrojové služby nedostane klient žádnou odpověď a v případě chybějících kontrolních mechanismů může dojít i u něj k selhání [46].

Proto je doporučováno využít **asynchronního volání** a synchronní způsob je vnímán jako nevhodný. Klient odešle požadavek, ale nečeká na doručení odpovědi a dál pokračuje ve své činnosti. ESB požadavek zachytí, přepošle ke zpracování, servis vrátí odpověď a ESB vypracuje další nové volání, tentokrát směrem na klienta. Obě volání se provedou nezávisle, proto se princip označuje „vystřel a zapomeň“, dvojice jednocestných volání, odborně asynchronní neblokující volání typu požadavek a odpověď [2][46].



Obrázek 10 Komunikace Servis-Gateway [46], Zdroj: vlastní úprava

### 3.4 Výhody a nevýhody

Mezi výhody SOA patří **strukturovanost, opakovatelnost, snadnost údržby, škálovatelnost, spolehlivost** a využití **více aplikačních platforem**.

**Struktura** a rozdělení jednolitého celku na několik menších a spravovatelných částí, které je možné využít opakovaně, rozdělit odpovědnost za jejich vývoj do více částí. Rozdělená a struktura podporuje více agilní metodiky a podporuje tak rychlejší proces vývoje softwaru. V případě potřeby je i možnost některou z komponent přebrat od dodavatele nebo se jen na služby dodavatele připojit [74][31][81][86].

**Opakovatelnost** každé z komponent v různých místech aplikace. Díky tomu je možné vytvářet různé kombinace vazeb a tím lépe podporovat modelování procesů [74][64][41][109][31][81].

**Snadná údržba** je možná díky rozdelení struktury do komponent, u kterých lze omezit dopad případných změn pouze na jejich oblast. Po provedení změn je možné nahradit komponentu do produkčního prostředí bez nutnosti kopírovat znova celý systém a tím tak zkrátit dobu případného výpadku v produkčním prostředí [74][41][90].

**Škálovatelnost** a jeho efektivnější provedení díky rozdelené struktuře. Lze navýšit prostředky jen pro jednu konkrétní vytíženou službu prostřednictvím vertikálního navýšení výkonu serveru nebo přidání dalších replikací v horizontální rovině a možnost vyčlenit pro vytíženější servisy více výpočetních prostředků [74][13][109][31].

**Spolehlivost** je zvýšena díky možnosti izolovat chyby jedné služby více v jejích vlastních hranicích a nešířit je dále do systému. V případě klíčové servisy ale dojde i tak k zásadnímu ovlivnění a výpadku. V případě klíčového prvku jako je systémová sběrnice zajišťující komunikaci, je chybovost stále soustředěna na jednom místě [74][41][31][90].

**Multiplatformní** prostředí je vlastnost umožňující zařazení více různých technologií do jednoho systému. Lze tak pro každou ze služeb využít vhodnější technologii a při zachování standardních protokolů pak komunikovat s okolním prostředím [74][64][31].

Mezi nevýhody SOA patří nižší **rychlosť**, vyšší **cena**, **komplexnost** a vyšší **datový přenos**.

**Rychlosť** zpracování v porovnání s MA je při rozsáhlých operacích nižší z důvodu distribuovaného zpracování procesu napříč větším množstvím služeb a rychlosť je tak limitovaná nejen jejich výkonem, ale také prostupností přenosové sítě. Další problém nastává v případě, že je jedna služba zapojena do více procesů a tím je způsobeno zpomalení všech proces. Při vzájemné komunikaci musí služby ještě před odesláním požadavek připravit do předepsaného formátu pro výměnu [74][13][31].

**Komplexnost** vývoje je vyšší od samého začátku vývoje systému. Jednotlivé komponenty jsou snadnější na vývoj, ale jejich integrace a vytvoření vazeb v systému za účelem dosahování složitějších operací a procesů vyžaduje vyšší úsilí na tvorbu modelu a jeho udržení. Pro realizaci aplikací menšího rozsahu není tento postup nevhodnější [64][31][81].

**Cena** v podobě času a vynaloženého úsilí při realizaci projektu je vyšší z důvodu větší komplexnosti návrhu. Jednotlivé prvky jsou sice jednodušší a rychlejší na vývoj ale jejich integrace v jeden komplexní systém je díky tomu složitější i z důvodu možného využití různých technologií. [74][31][81].

**Datový přenos** a schopnosti komunikační sítě je důležité kritérium rychlosti zpracování operací, které v monolitické architektuře probíhali přímým voláním metod. Oddělené služby potřebují navzájem komunikovat a objem zpráv roste souběžně s jejich narůstajícím počtem [74][64].

### 3.5 Servisně orientovaný přístup

Stejně jako v reálném světě existuje způsob, jak provést údržbu a neomezit všechny funkce, musí existovat stejná možnost i ve virtuálním světě při návrhu aplikace.

Řešením je **rozdělení celku na jednotlivé části** a jejich autonomní správa, která neomezuje funkce ostatních. Jak již bylo na příkladu uvedeno, Karlův most, Pražský hrad a Vyšehrad budou spravovány samostatně, a tudíž může přes jistá omezení Praha dále pro turisty **poskytovat servisy**.

Aplikaci je třeba rozdělit do několika servisů podle oblasti zaměření. Pokud nebude fungovat jedna z nich, ostatní budou stále poskytovat svou funkcionalitu a aplikace bude (v omezené míře a za určitých podmínek) stále v provozu. Takovou aplikaci označujeme jako **servisně orientovanou**.

Stejně jako lze popsat monolitický přístup na neinformatickém příkladu, lze obdobně převést na stejný problém i servisně orientovaný přístup. Pokud bude opět město Praha vystupovat jako aplikace s funkcionalitou Karlova mostu, Pražského hradu a Vyšehradu, budou tyto části rozděleny na autonomní celky schopné být

samostatně spravovatelné a svou chybovostí izolované. Pokud bude potřeba provést údržbu jedné sochy na Karlově mostě, neomezí toto opatření celkové možnosti Prahy a po dobu údržby bude uzavřen pouze Karlův most. Omezení funkcí může ale nastat v případě cesty z Vyšehradu na Pražský hrad a na trase by nebylo možné využít Karlův most – toto omezení funkce by mělo i tak částečný dopad na ostatní prvky, protože byla vyřazena klíčová část vyhlídkové trasy.

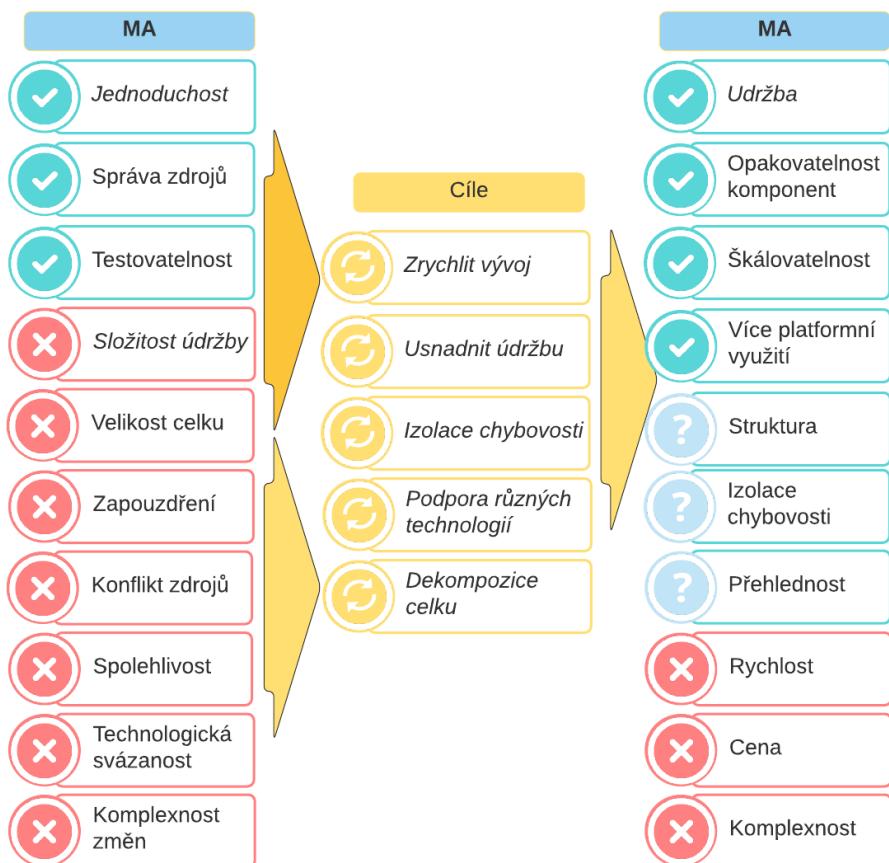
V případě IT aplikace to znamená, že je rozdělena na autonomní významné části poskytující servis a své funkce. V případě výpadku jedné z klíčových služeb zajišťující vazbu mezi všemi ostatními, je výrazně omezena funkce celého systému. V omezené míře je systém stále funkční, ale funkce není optimální a nemusí splňovat požadované standardy.

Jako problém je možné vnímat nedostatečné členění prvků a vzájemné pevné vazby s dopadem na celkovou funkcionality. Na příkladu je tedy nutnosti provedení údržby na Karlově mostě povede, jako na centrálním pojícím prvku, k uzávěře pouze malé oblasti a údržby jediné sochy. Dopad na celkovou funkcionality Karlova mostu je minimální a celkový dopad na systém je sice pozorovatelný, ale téměř nulový, protože ostatní funkce jsou zajištěny.

V oblasti IT to znamená, že jednotlivé domény systému, které jsou zahrnuté do velkých organizačních celků, rozdělíme v rámci mikroservisního přístupu na ještě drobnější domény. V případě velkého rezervačního systému můžeme rozdělit jednotlivé katalogy na jednotlivé malé domény podle jejich druhu. Správu uživatelských účtů, jejich organizačních součástí a správy výplat v personálním systému, kde do teď vystupovaly jako jeden servis nebo modul, můžeme rozdělit na menší domény.

Návrh, jak provést rozdelení na jednotlivé malé části je součástí prvních fází projektu a vzniká na základě zpracování modelu a využití technik doménově řízeného návrhu. Základní myšlenkou je jasně definovat základní funkce a procesy bez hlubších vazeb a autonomně fungující. Cílem servisně orientovaného přístupu je více se přiblížit reálnému fungování světa a stejně jako v uvedeném příkladu zajistit

celkovou funkčnost. Toho lze dosáhnout pomocí izolace chybovosti v malé oblasti a nevytváření kritických vazeb a závislostí. Definice takového cíle a snaha o jeho dosažení dovedla původní myšlenku servisně orientované architektury v její vylepšenou formu označovanou jako mikroservisní architektura.



Obrázek 11 Porovnání MA a SOA, Zdroj: vlastní tvorba

## 4 Mikroservisní architektura (MSA)

Architektura založená na mikroservisním přístupu je v současnosti při návrhu aplikací velmi populární. Vznikla na základě potřeby reagovat na rychlý vývoj v informační společnosti a klade důraz na přesnější reflexi reality při potřebu podpory podnikových procesů. Oproti monolitickému přístupu nabízí v této oblasti efektivnější řešení problémů a vybírá to nejlepší, že servisně orientovaného přístupu. Mikroservistní architektura zakládá na principech SOA, které rozvíjí tam, kde SOA nedostálo očekávání. Často je MSA označováno jako SOA verze 2 a jako jeho evoluční nástupce. Přináší rychlosť, různorodost, nový pohled a změnu myšlení i přístupu k návrhu aplikací [76][61][58][18][34][29].

Technologie založené na mikroservisní architektuře využívají společnosti jako Netflix, Spotify, Hbo nebo Asos [15].

MSA přináší mnoho výhod, ale zároveň i několik komplikací. Na některé případy může být stále s ohledem na mezní užitek lepší využít monolitický přístup nebo hruběji členěnou SOA. Každá z těchto tří architektur má své výhody a nevýhody a cílem je využít kombinaci to nejlepšího z nich. V literatuře a odborných konferencích je často zmiňováno, že MSA není vše řešící „stříbrnou kulkou“ na všechny problémy. Je ale obecně vnímána jako nová alternativní a svěží cesta, otevírající velké množství možností, jak dělat věci jinak a snad i lépe. [76][63][58][18][29].

Základním principem mikroservisního přístupu je návrh nebo dekompozice existujícího systému na malé autonomní části. Tyto části, jako základní stavební jednotka v MSA, nejsou nic jiného než velmi malé monoly. Tyto části jsou schopny vzájemně spolupracovat za účelem dosahování cílů definovaných v rámci celku. Stejně jako u SOA je touto základní stavební jednotkou služba, neboli servis [61][34][15].

Aktuálně velmi populární a užívaný termín MSA má své počátky spojené s historií SOA a první zmínky o návrhu a užití pochází z 90. let. Myšlenka byla rozvíjena a využívána v praxi až do roku 2015, kdy došlo k dalšímu

posunu, a v této formě dospěla do současného stavu širšího využití v praxi [110]**Chyba! Nenalezen zdroj odkazů.**[9].

Mikroservisní architektura jako systém uceluje kolekci malých autonomních částí, které jsou snadno udržovatelné, nezávislé, samostatně provozované, spolupracující ve volné vazbě a zaměřené konkrétní oblasti. Tento styl umožňuje rychlý vývoj s častou frekvencí přírůstku aplikovaných změn a je ideální pro využití agilních metodik při vývoji aplikací [9][34].

Kromě servisů samotných je v MSA obsažena celá množina prvků a aspektů, které vytvářejí hlavní vlastnosti celé architektury: Servis, Ohraničený kontext, granularita, doména, distribuovaný systém, kontejnerizace, Service Discovery, orchestrace a choreografie, událostní řízení, API Gateway a Monitoring.

## 4.1 Servisy

Servisy jsou základním stavebním prvkem. V rámci MSA se vyznačují několika hlavními vlastnostmi. Hlavní vlastnost, které servis určuje, jsou jejich velikost a hranice. Vyznačují se svou samostatností a nahraditelností [61].

Servis je malý a samostatně provozovaný monolitický blok aplikace, který poskytuje své metody prostřednictvím komunikačního rozhraní svému okolí. Každý servis je navržen tak, aby jej bylo možné libovolně přidat, nahradit nebo i odebrat ze systému a nebyla tím narušena činnost ostatních servisů. V případě, že je servis přidán nebo nahrazen, je okamžitě schopný komunikovat standardním způsobem a začít poskytovat svou funkcionality systému, bez nutnosti složitého vytváření dalších vazeb. Není tak nutné provádět dlouhé analýzy a rozbor kódu v případě údržby nebo předání kompetencí na jiný vývojový tým či vývojáře [61].

Při návrhu nového servisu je potřeba stanovit jeho **optimální velikost a hranice působnosti** [61].

Jak z názvu architektury vyplývá, měla by být velikost vnímána jako „mikro“, tedy jako něco velmi malého. Neexistuje přesná definice a měřítko, co znamená malá velikost. Může to být počet řádků kódu, objem zdrojů, které spotřebovává nebo

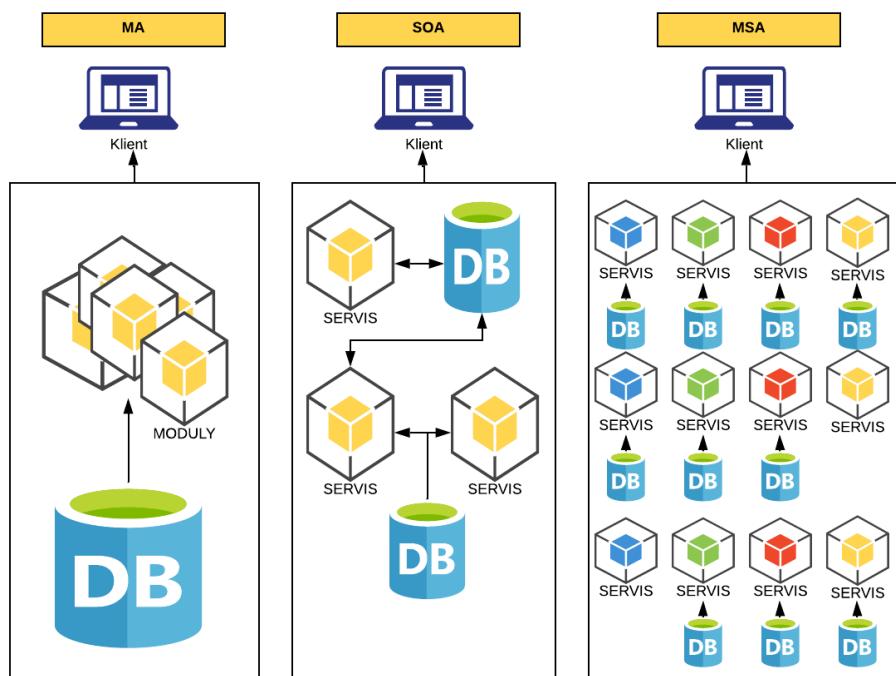
odpovědná oblast funkcionalit v rámci systému. Velikost by měla být vždy relativně malá vůči celkovému rozsahu systému, ve kterém působí, z pohledu časové náročnosti vývoje nebo velikosti své domény [61].

**Ohraničený kontext** (Bounded Context) znamená, že má servis jasně vymezené hranice v oblasti, za kterou plně odpovídá, přičemž tuto oblast má plně ve své správě a zároveň nezasahuje do oblasti působnosti jiných servisů.

Díky ohraničenému kontextu je možné garantovat izolaci chybovosti v rámci systému. To znamená, že pokud uvnitř servisu dojde k chybě, nebude se závada šířit a ovlivňovat další servisy nebo funkcionality. Podle tohoto kritéria je potřeba uvažovat i při návrhu databáze. Každý servis by měl mít k dispozici databázi pouze pro sebe a v databázi by měly být maximálně dvě až tři tabulky. Ideálně pak pouze jednu. [61].

Způsob rozdělení odpovědnosti, zdrojů a ohraničení závisí na míře granularity systému a na doménově orientovaném návrhu aplikace [61].

**Granularita** je pojem označující míru rozdělení systému. Při mikroservisním návrhu je právě způsob, jakým bude celek rozdroben na jednotlivé části, tou největší výzvou a základním rozhodnutím pro efektivitu a rozvoj budoucího systému [76].



Obrázek 12 Struktura MA, SOA a MSA, Zdroj: vlastní tvorba

Z míry granularity vychází i název monolitické architektury. Ten je, jak bylo popsáno výše, vnímán jako jeden celistvý blok. Granularita a způsob, jakým je rozčleněna struktura. To je také první jasný rozdíl SOA a MSA. Zatímco SOA je vnímáno jako hrubě strukturovaná (coarse grained), mikroservisy jsou strukturované jemně (fine grained) [76][75].

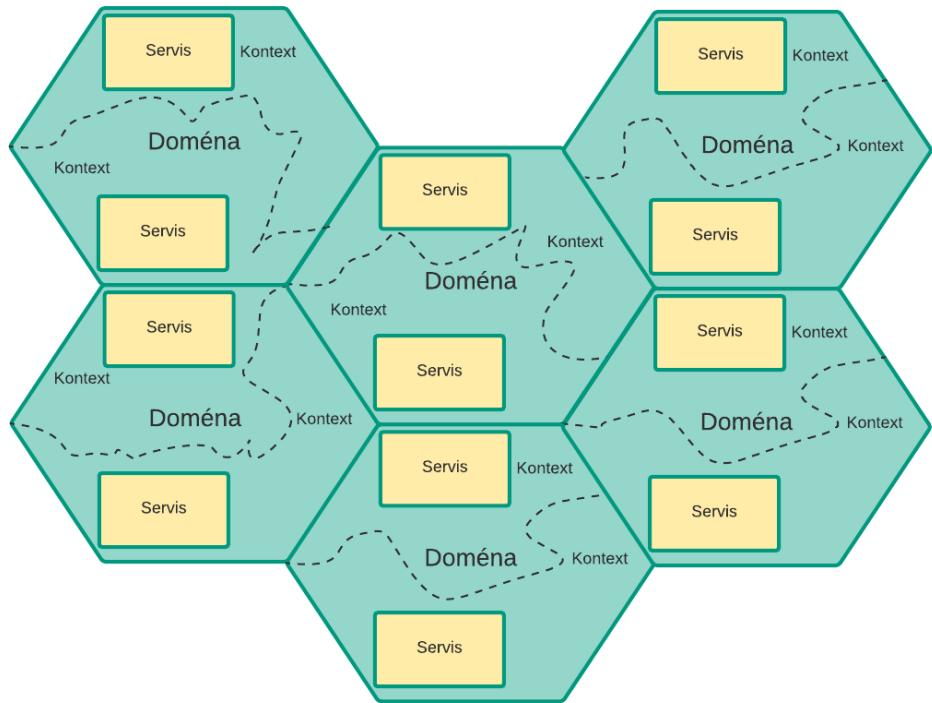
V případě mikroservisů je nutné najít správnou míru granularity. Každý servis bude mít v systému potřebu komunikovat a při velkém počtu servisů to znamená i větší potřebu komunikace, větší složitost vykonání složitějších operací, větší zátěž sítě, delší dobu odezvy a větší možnost selhání v případě přetížení úzkých míst systému. Proto je u systémů čítajících stovky mikroservisů určení správného rozdělení na části zásadním krokem [75].

Doporučený postup optimalizace a nalezení správné granularity je začít s hrubě strukturovanými servisy a analyzovat jejich funkčnost. Pokud je při této analýze nalezena vhodná hranice, lze provést rozdělení na více servisů a následně provedena opětovná analýza dopadů na odezvu a funkčnost systému. [75][63].

## 4.2 Domény a doménově řízený návrh

**Doménově řízený návrh** (Domain Driven Design, DDD) obsahuje metody a návrhové rámce popisující způsob rozdělení systému do částí. Je to způsob, jak na základě zpracovaného modelu reality rozdělit, efektivně vyvíjet, provozovat a spravovat aplikace řešící netriviální podnikové procesy a oblasti [58][59].

Funkčnost systému je rozdělena do jednotlivých domén podle typu podnikových procesů a jejich hodnoty pro podnik, podle druhu procesů nebo podle objektů, se kterými se bude nakládat. Tím je určena i oblast působnosti každého servisu a odpovědnost je tak jednoznačně distribuovaná [58][37][52][59].



Obrázek 13 Schéma doménově řízeného návrhu, Zdroj: vlastní tvorba

Pro vypracování modelu domén lze zvolit několik různých strategií. Těmito metodami jsou **hlavní proces, spolupráce, výzkum a experiment, komunikace, porozumění problému a rozvoj** [27][37][59].

Při metodě **hlavního procesu** se určí klíčové procesy a z těch jsou pak vytvořeny domény. Hlavní doména je taková, která přináší podniku největší ekonomický přínos, nebo je zásadní pro funkčnost vytvářené celé aplikace [37].

Dělení podle **spolupráce** spoléhá na komunikaci mezi zpracovateli modelu a osobami odborně zainteresovanými v řešené oblasti procesu. Výsledný model je kompromisem mezi po hledem z technické stránky věci a ze strany byznysu [37][52].

Metoda **výzkumu a experimentu** spoléhá na prvotní analýzu, při níž je zpracován úvodní návrh ihned konfrontován s využitím v praxi. Při uvedení návrhu do praxe se provádí testování se na skupině vybraných osob, jsou sbírány metriky a je vypracována zpětná vazba. Podle reakcí a údajů z reálného prostředí pak lze navrhnout další úpravy modelu [38][53].

Metoda **komunikace** slouží jako hlavní způsob poznání reality. Kladen je velký důraz na úvodní analýzu a jasné komunikování řešeného problému a procesů mezi všemi

zpracovateli. Je vytvářen srozumitelný schématický model a spolu s ním i technické zpracování návrhu. Model je upravován a zdokonalován na základě diskuse, dokud není na první pohled všem zainteresovaným dobře pochopitelný výsledek, dosažen kompromis nebo třeba i poznání, že není nutné problém vůbec řešit [38][53].

Při metodě **porozuměním** je klíčovým prvkem jasné stanovení používaných pojmu a jazyka použitého při tvorbě modelu. Tento krok je důležitý z toho důvodu, aby všichni aktéři zasahující do procesu, kteří přicházejí z různých oblastí a odborností, měli ujasněné názvosloví a chápali komunikované problémy stejným způsobem. Opatření zabraňuje chybné interpretaci na základě lidského jazyka, protože každý může pod jedním termínem různou představu, co to znamená. Cílem je do hloubky a jasně popsat všechny aspekty modelu tak, aby splňoval správně všechny požadavky [38].

Cílem hlediska **vývoje** je udržovat vytvářený model tak jednoduchý a přehledný, jak je to jen možné. Oblast řešení se může neustále vyvíjet a je potřeba reagovat na případné změny už během vývoje nebo následně v provozu systému. Proto je jasně definovaný a pochopitelný model nutností [38][53].

Výstupem doménově řízeného návrhu jsou **jasně stanovené domény**, do kterých lze zasadit jednotlivé servisy, které si v dané doméně vymezí svou hranici a vytvoří kontext, který budou v této doméně řešit. V ideálním případě by měl tento kontext pokrývat celou vytvořenou doménu. Při tvorbě servisů v MSA je kladen důraz právě na **co nejmenší velikost**. To může do jisté míry omezit možnosti při návrhu tvořené domény, ale zároveň je to dobrou návodou a pomůckou, jak postupovat jasně a stručně a udržet tak přehlednost vytvářených modelů. Uvedené metody také nápadně korespondují s některými postupy zmíněnými v **agilních metodikách** (např. v extrémním programování spolupráce na úrovni zákazník a vývojový tým) [61] [38][58].

### 4.3 Distribuovaný systém

**Systém budovaný v MSA je systémem distribuovaným.** Aby byl systém opravdu plně distribuovaný, je nutné udržet výše zmíněné principy samostatnosti

a ohraničeného kontextu. Jednotlivé domény lze rozdělit do správy konkrétním vývojovým týmům a pracovat tak efektivně, paralelně a nezávisle na sobě. Stejné výhody platí i provozu a údržbě systému [76].

Distribuované systémy mohou být řízeny centralizovaně nebo decentralizovaně. V MSA je převážně využíván decentralizovaný způsob [37][53].

Hlavní výhodou distribuovaného systému je možnost nezávislosti na jedné technologii. Pro každou z činností systému je pak možné využít různé technologie, nejlépe se hodící pro konkrétní oblasti. Například využití různých typů databází [58].

Výhodou distribuovaného systému je možnost decentralizace. Jejím cílem je omezení centrálních prvků vykonávajících řízení, na nichž by mohly být jednotlivé servisy závislé. Proto je třeba se vyhnout tvorbě prvků, jako jsou systémové sběrnice (enterprise service bus, ESB) a dávat přednost přístupům podporující choreografii jednotlivých služeb před orchestrací (kapitola 4.6). Vhodnou praxí je využití jednoduchých prostředníků, poskytujících komunikaci asynchronním způsobem. V práci s daty je nutné udržovat přísně stanovené hranice každého servisu získané z doménově řízeného návrhu [63].

Systémy s větším počtem samostatných servisů mají díky svému decentralizovanému přístupu větší stabilitu v případě selhání některé z komponent. Pokud jeden servis bude v chybovém stavu, není zbytek systému významně ovlivněn a nedojde k výpadku všech funkcionalit jako v případě monolitického přístupu. Chyba je izolována pouze na svůj kontext a doménu. Ne všechny činnosti klienta systému budou vždy jen a pouze omezeny na jednu doménu. Aby mohl být systém řízen decentralizovaně a zároveň vystupovat směrem ke klientovi jako jeden celek musí existovat pravidla, jak spolu mají servisy komunikovat, fungovat v harmonii a zároveň si zachovat svou autonomii [63].

**Kontejnerizace** je proces provozu decentralizovaných, kontextem omezených prvků ve vlastním prostředí a s vlastní správou zdrojů. Možnost umístění servisu do vlastního kontejneru je podmíněna nulovou vzájemnou závislostí kódu mimo konkrétní servis. Každý servis je virtualizován ve svém kontejneru a pro tyto účely je v rámci ekosystému v MSA možnost využití několik nástrojů, které tento proces

virtualizace podporují. Mezi nejznámější a nejpoužívanější kontejnerizační nástroje se řadí Docker, Kubernetes, Spring, Virtual Box [37][20][109][19][19].

#### 4.4 Service Discovery

Pokud chceme složit systém, ve kterém je zapojeno množství samostatných komponent, založených na různých technologiích, a virtualizovaných každá ve svém prostředí, je nutné zavést způsob, jakým by se o sobě měly navzájem dozvědět a jak komunikovat [63].

Technologie a metody řešící tento problém se souborně označují jako Service Discovery. Jsou klíčem ke vzájemné komunikaci servisů jako součástí distribuovaného a proměnlivého prostředí. Protože je prostředí mikroservisů takto proměnlivé, servisy zanikají nebo běží v několika svých instancích z důvodu zvýšení výkonu, je potřeba využívat nástroje a metody, které si s těmito vlastnostmi dokážou automatizovaně a efektivně poradit [63][79].

Prvním z možných způsobů je přímé pevné nastavení konkrétní IP adresy a portu, na kterém má servis vystaven svůj přístupový bod. Je to základní přístup pro vzájemné propojení servisů, které je nezávislé na použité technologické platformě při komunikaci v síti [63].

Nevýhodou je, že tento způsob obnáší vyšší nároky na údržbu a přináší komplikace v případě, že provedeme změny a na straně klienta jsou stále načtené staré adresy servisů. Tento problém pak řeší centrální prvek pro vyvážení výkonu (*Load Balancer*), který tyto záznamy udržuje a u kterého můžeme v případě změny ručně provést změnu, která bude propagována ke klientovi. Pokud jsou změny prováděny velmi rychle a často, je tento způsob efektivně neudržitelný. Sám o sobě nedrží přímou relaci na servisy, ale pouze na místo, kde se servis nachází nebo právě nacházel [63].

V SOA je téma Service Discovery řešeno definováním vlastností služby pomocí formátu UDDI, prostřednictvím něhož je služba registrována na server, a dále je tato informace předávána prostřednictvím zprávy v protokolu SOAP. Pokud pak klient

zkouší vyhledat požadovanou funkcionality služby, kontaktuje server UDDI, získá WSDL informaci o službě a může se k ní připojit [82].

Další možností je použít vhodný pomocný nástroj s většími možnostmi správy, které všechny provozované servisy registrují ve své paměti a poskytují je všem, kdo s nimi potřebují komunikovat. Obecně jsou tyto nástroje označovány jako orchestrátory, například Kubernetes, Consul nebo Ocelot. Podrobněji budou tyto programy v popsány v kapitole použitých technologií.

## 4.5 Komunikace

Vzájemná komunikace mnoha nezávislých komponent je klíčovým prvkem distribuovaného systému. V případě mikroservisní dobře propracované komunikace by systém nemohl fungovat, byl by pomalý a málo efektivní. Každá z komponent poskytuje své aplikační komunikační rozhraní (Application Program Interface, API). Návrh správného způsobu je důležitý z hlediska zachování samostatnosti a všech komponent a platformní nezávislosti. Při konstrukci komunikačního rozhraní, které má za úkol překlenout volný prostor se využívá technologií pod označením WEB API [61].

Komunikace mezi komponentami probíhá skrz síťové prostředí použitím metody vzdáleného volání (Remote Procedure Calls, RPC) nebo zasíláním zpráv na jejich koncové body (End Points) vystavené na svém API [28].

Návrhových vzorů, jak nastavit komunikaci je mnoho, ale v mikroservisech je nejčastěji používáno schémata HTTP+REST [28][49].

Základním protokolem využívaným ke konstrukci API je HTTP. Je to základ podporovaný všemi platformami. Je to standard, který definuje základní pravidla a typy zpráv, které se předávají [28][80].

Jako je v SOA využívaný stavový protokol pro volání SOAP, tak v MSA je zde využíván především nestavový standard REST (Representational State Transfer). REST je navržen jako ideální standard provozovaný na protokolu HTTP sloužící pro distribuované systémy. Jeho hlavní výhodou je jednoduchá implementace, oddělení

části klienta a zdroje a podpora jednoduchých standardizovaných formátů zpráv typu XML nebo JSON. Typy zpráv jsou standardně rozděleny do **GET**, **DELETE**, **POST**, **PUT**, **HEAD**, **OPTIONS**, **PATCH** a **LINK/UNLINK** [28][12][80].

**GET** je základním typem volání, které vrací informaci o zdrojovém prvku nebo entitě.

**DELETE** je typ příkazu, který na zdrojovém prvku nalezne záznam podle předaného identifikátoru a provede jeho trvalé odstranění.

**POST** označuje zprávy sloužící k zaslání a vytvoření nového prvku.

**PUT** je volání obsahující identifikátor a model objektu zdrojového prvku, který je pomocí identifikátoru nalezen a modelem nahrazen.

**HEAD** slouží stejně jako GET pro získání informace. V tomto případě je návratovou hodnotou informace o zdrojovém prvku. Nemusí vracet data, ale pouze informaci o nich. Tento příkaz slouží k testování dostupnosti zdroje.

**OPTIONS** je zpráva, která zjistí informace a požadavky cílového prvku a nezpůsobí žádnou jeho akci. Je čistě informativní.

**PATCH** je obdobou příkazu PUT s tím rozdílem, že slouží pouze pro úpravu části cílového prvku. PATCH ale není tak široce podporován a je lepší použít příkaz PUT.

**LINK/UNLINK** – vytvořen a zrušení vazby mezi zdrojovým prvkem a vlastním prvkem.

V praktickém použití jsou nejčastějšími příkazy první čtyři, které stačí na realizaci všech potřebných operací. Je to základní standardní způsob ideální vhodný pro heterogenní prostředí MSA. Bohužel má ve svém základu omezení, a to je možnost pouze synchronního přenosu. Proto je v mnoha případech vhodné využít rozšíření pro HTTP klienta, které dokáže asynchronní způsob komunikace i za pomocí těchto metod zprostředkovat a neblokovat tak původce volání [28][80].

Kromě výměny objektů mezi klientem a cílovým prvkem je také možná identifikace odpovědi pro potřeby testování. Tak jako je standard pro volání, tak je stanoven standard odpovědi formou kódů. Těchto kódu existuje velké množství, a proto uvedu pro vývoj a testování API ty nejdůležitější [66][80].

**200 OK** – úspěšné volání a splnění požadavků volání.

**201 Created** – úspěšné přijmutí příkazu a vytvoření prvku na straně poskytovatele.

**304 Not Modified** – prvek byl nalezen, ale nebyl modifikován.

**400 Bad Request** – špatně strukturované volání.

**401 Unauthorized** – nedostatečné oprávnění pro přístup.

**403 Forbidden** – volání metody není povoleno.

**404 Not Found** – špatná adresa volání a nenalezení cílového bodu.

**500 Internal Server Error** – indikuje nedostupnost zdroje nebo chybu způsobenou na straně příjemce, který očekává jiný objekt nebo parametr zasláný v odpovědi od serveru.

Tyto chybové kódy jsou velmi důležité při vytváření a úvodním testování API. Pomocí nástrojů pro volání, jako je například Postman, můžeme nasimulovat volání a sledovat, jaký objekt a volání se nám vrátí. Později při již vytvořené kompletní aplikaci je vhodné tyto kódy sledovat pro účely monitoringu a logování [80][37].

Objekty, které jsou zasílány nebo přijímány, jsou stejně jako struktura odpovědi strukturovány do XML nebo JSON. JSON je v MSA používán častěji, neboť disponuje střízlivějším formátem, a tudíž i menší velikostí zasílaných zpráv. Výhody JSON jsou jednoduchost pro psaní, zápis a čtení a snadnost serializace a deserializace objektů. Struktura je tvořena identifikátorem objektu, který má název atributu, a hodnotou, která může být opět jednoduchá nebo objekt pole [80].

V MSA je REST API základní způsob, jakým servisy staví své rozhraní k okolí a jakým způsobem navzájem volají a přijímají odpovědi. Díky mechanismům Service Discovery získají svoji základní adresu, což v základu postačuje na navázání spojení, neboť stačí podle potřeby prověřit standardní koncové body podle REST a na ty pak zasílat požadavky. Tyto metody stačí pro všechny CRUD operace (Create, Read, Update, Delete, CRUD). Princip je nastavený ideálně pro zapojení navzájem nezávislých prvků bez nutnosti je dále popisovat, nicméně je vhodné zapojit více způsobů a metodám dávat vhodná popisná jména, protože se není nutné při vývoji omezovat jen na základní operace [66][79][80].

Pro popis API rozhraní každého servisu je vhodné stanovit jasné mechanismy už při vývoji. Jednou ze standardních metod je použití prostředníků pro generování uživatelsky přívětivého prostředí, například použitím nástroje Swagger. Vystavení metody na standardní end point je obecnou slušností a nutností především v případě, že počítáme s napojením externích klientů, které nevyvýjíme vlastními prostředky.

Díky těmto postupům je překlenut prostor mezi servisy a je umožněna standardizovaná komunikace. Dalším důležitým faktorem je určení stylu a metodiky, jakým se bude zpracování řídit. V MSA se uplatňují především dva styly, a to jsou **Message Oriented** a **Hypermedia driven** [37][80].

**Messsage-oriented**, jak z názvu vyplývá, je způsob předávání zpráv mezi komponenty, díky čemuž může v pozadí probíhat nezávislý vývoj a úpravy mechanismů a při dodržení struktury zpráv se není nutné obávat nekompatibility s okolím. Je to logické rozdělení vnitřního modelu a obrazu, který je prezentován jako očekáváná návratová a odesílaná hodnota. Tímto tématem se zaobírá následující kapitola Event Sourcingu [79][37].

**Hypermedia-driven** je další možnost nastavení komunikace. V předchozím případě byla logika rozdělena do zpráv, které příjemce zasílá, a proto bylo nutno znát logiku a jaký další příkaz zavolat, pokud vyžadujeme akci. Hymermedia-driven obsahuje tuto informaci už přímo v sobě. Využívá základních vlastností, na kterých je postaven samotný web a webové prohlížeče. Objekt, který klient během komunikace získá, je už strukturovaný HTML formát, který kromě osazení daty obsahuje také adresy volání pro následující akce s objekty. Logika akcí je ponechána a koncentrována na straně zdrojového prvku. Tento princip je také označován zkratkou **HATEOAS API** (*Hypermedia As The Engine of Application State*) [61][37][63].

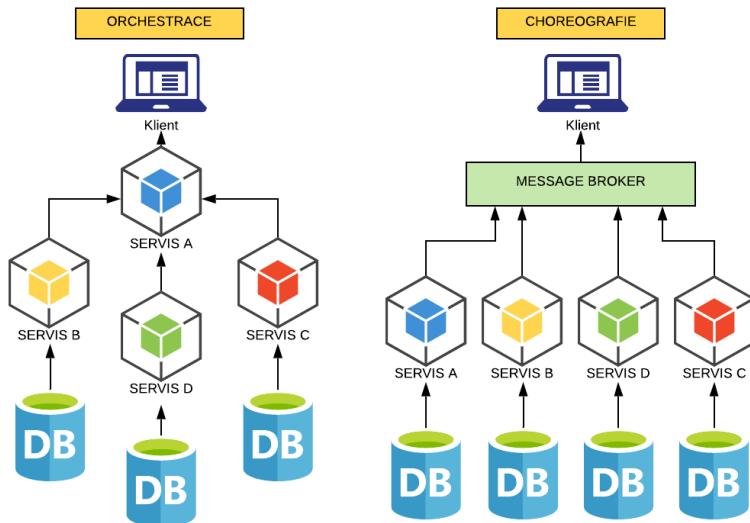
Je to vhodná varianta pro uvolnění vazby mezi klientem a serverem ve velmi pochopitelném a lidsky přirozeném formátu. Klientská aplikace vůbec nemusí řešit případné změny na struktuře předávání zpráv. Na druhou stranu je nevhodou, že je klient odkázán na metody jemu poskytnuté, a pokud je chce využít svým způsobem,

musí je v odpovědi dohledat a pak je implementovat tak, jak potřebuje ve své logice. V případě volby způsobu hypermedia-driven je proto vhodné myslet na to, že je vhodný jen na určité případy užití a nemusí vyhovovat všem [63].

Na závěr je dobré říci, že REST API je sice široce používáno pro svou jednoduchost a snadnost implementace, ale v některých případech a delších horizontech může právě tato jednoduchost způsobit komplikace, na které je nutné brát zřetel. V případě zasílání velkých objektů je důležité rozhodnutí, jakým způsobem budou ukládány v databázi. V případě, že je chceme ve strukturované podobně, je nutné během komunikace provádět serializaci a deserializaci. To může mít zásadní nepříznivý dopad na výkon efektivity systému [63].

## 4.6 Orchestrace a choreografie

V MSA je každý servis navržen tak, aby fungoval samostatně a měl v odpovědnosti pouze jednu entitu systému, u které poskytuje všechny potřebné funkce. Tyto malé vymezené oblasti ale nemohou samy o sobě pokrývat všechny požadavky na operace reálného světa. Aby bylo možné v systému těchto servisů dosáhnout komplexnějších operací, je nutné nastavit posloupnost, jak se budou dílčí operace vykonávat. V relační databázi jsou tyto operace realizovány formou transakce. To je v distribuovaném systému nemožné. V MSA je proto nutné vytvořit způsob, jak společné složené operace nastavit a jak provádět zápis obdobně jako v transakci. Díky mechanismům Service Discovery a standardizovanému API mohou servisy komunikovat. Vzájemná koordinace a provedení složených operací probíhá v režimu **orchestrace** nebo **choreografie** [61][63][87].



Obrázek 14 Orchestrace a Choreografie Zdroj: [87], vlastní úprava

V případě orchestrace se využívá jednoho centrálního prvku, který sice servisy přímo neřídí a nesvazuje v jejich samostatné činnosti, ale komunikace za účelem vzájemné spolupráce na něm závisí. Stejně tak jak na příkladu hudebního orchestru, kdy jednotliví hráči hrají na různé nástroje podle svých not a své části skladby, ale zároveň všichni sledují dirigenta, který drží celý soubor v tempu a vzájemné harmonii [63][87].

V oblasti informačních technologií a vývoje aplikací by to na příklad znamenalo, že založím nový účet uživatele jedné servisu a v závislosti na této akci provede jiný servis odeslání notifikačního emailu a jiný servis zase založí uživateli kartu v evidenci docházky nebo zažádá o přidělení vstupů do kancelářských prostor podle organizační současti, do které byl účet zařazen. Následné akce jsou tedy volány pomocí API příslušných služeb. Centrálním dirigentem se pak stává servis, který účet uživatele založil [63].

Všechny tyto servisy pracují samostatně, ale díky centrálnímu bodu pro výměnu informace o nastalé události dokážou zareagovat a provést svou část komplexnějšího procesu. V tomto případě už je ale vytvářena pevnější vazba, která ubírá na přísné decentralizaci systému [63].

Na jedné straně může být výhodou, že všechna logika týkající se uživatelského účtu a následných operací je uložena v příslušném servisu a své doméně. Lze možné snadno a rychle změnit pravidla akcí týkající se této domény. Na druhé straně právě

jistý stupeň centralizace a soustředění pravomoci v jediném místě přináší nutnost udržet tento prvek prioritně v provozu a vytváří v rámci komplexní operace pevnější závislost. V případě, že by i částečná centralizace byla brána jako zásadní problém, je možné využít další způsob jak i tuto vazbu v komplexní operaci odlehčit pomocí choreografie [63].

**Choreografie** služeb je dalším možností, jak ještě přísněji dodržovat způsob decentralizace systému při složených procesech a operacích. V obecném příkladu to znamená, že každý servis zná svůj úkol a místo v systému, vnímá své okolí a na základě probíhajících akcí, které ho zajímají a na které ví, že má zareagovat své akce provede. Podle předchozího příkladu by to znamenalo, že servis, který účet uživatele založí a vydá o výsledku akce zprávu ve formě události. Servisy obsluhující zasílání notifikací, správu docházky a uživatelských přístupů vědí, že je tato operace zajímá, vyslechnou si jí a podle toho zareagují. Servis zakládající účet tedy přímo navazující operace nevolá a ani ho v zásadě nemusí zajímat, kdo všechno na jeho operaci zareaguje [63][79].

Velkou výhodou tohoto přístupu je možnost zapojit do skupiny, která reaguje na událost o založení účtu, libovolně další nové servisy za běhu systému, vyměnit stávající bez nutnosti starat se, jestli jsme změnili endpoint pro volání vzdálené procedury nebo můžeme servis úplně odstranit. Pokud bychom například zapojili do skupiny servis zajišťující na základě nového účtu objednání nového pracovníka na vstupní školení, zrušili jsme zasílání notifikačního emailu a upravili správu docházky tak, aby rovnou zaměstnanci nastavila plán docházky na celý průběh zkušební doby se všemi zapsanými kurzy. Kroky celého procesu jsou v případě využití choreografie v asynchronním režimu a vzájemná vazba mezi servisy je výrazným způsobem omezena na nezbytné minimum. Minimalizace vazby a rozdelení logiky do několika nezávislých míst je v mnoha ohledech výhodou pro funkci systému a jeho variabilitu. Možnou nevýhodou může být zvýšená náročnost a vyšší složitost analýzy procesu. Je nutné provádět důkladnější monitorování, zda vše proběhlo korektně a samozřejmě je nutné zpracovat kvalitní dokumentaci popisující procesy [63].

Je tedy nutné volit mezi částečnou centralizací, snadnější údržbou a řízením za cenu vyšší svázanosti servis. Nebo mezi velmi volnou decentralizovanou vazbou, stabilnějším provozem a možností snadných úprav na úkor nutnosti zvýšené práce na realizaci.

V tomto případě je logika zpracování rozdělena mezi několik servis, a proto je nutné zavést vhodné způsoby monitorování procesu a mít dobré zpracovanou dokumentaci, která vzájemné vazby popisuje [63].

## 4.7 Událostní řízení

Událostní řízení (Event driven) je označením pro činnosti a práce s daty na základě a podle pravidel událostně řízené architektury (Event-Driven architecture, EDA). Pojem Event driven se také překrývá s označením Event Sourcing, kdy je pojem více soustředěn na rozdělení čtení a zápisu pomocí CQRS a tvorbě agregovaných pohledů (views) z více oblastí. Tato architektura je jako přístup ideálním doplňkem a rozšířením mikroservisní architektury. Je navržena pro použití v systémech obsahujících minimum vzájemných vazeb mezi komponenty a vhodná pro asynchronní způsob zpracování. Je to tedy společným cílem EDA i MSA [63][79][107].

V klasickém pojetí práce s daty a realizace CRUD operací v relačně strukturovaných databázích je možné u operací snadno zajistit ACID (Atomicity, Consistency, Isolation, Durable) transakční přístup. Atomicitu, kdy je zaručeno, že se operace provedou v jednom celku, konzistentní stav databáze, izolované provedení transakcí a jistotu, že budou buď provedeny všechny, nebo žádný z kroků v transakci. Zároveň je jednoduché vytvořit bohaté view bez nutnosti duplicity dat [79].

V mikroservisně odděleném systému je vytvoření dostatečně bohatých view bez duplicity dat nemožné z důvodu snahy nevázat se. To ale často vede k tomu, že potřebný zdroj není dostatečně obsáhlí a musí se tak spolehnout na dohledání údajů na straně klienta. Řešením je právě způsob agregovaných view, které jsou schopny pomoci event sourcingu s principem CQRS (Command Query Responsibility Segregation). Tento vzor rozděluje odpovědnost za zápis a čtení. Data jsou pomocí

servisy jsou data ihned zpracována a zapsána jako základní model a později díky vydaným eventům obohacena a připravena pro čtení. Výhodou tohoto vzoru je možnost škálování výkonu. Pokud je zápis méně častá operace než čtení, můžeme přidělit více prostředků pouze pro čtení [100].

Hlavní myšlenkou událostního řízení je, že celkový stav systému je předáván a udržován prostřednictvím vydávání událostí (eventů) na základě vykonaných operací nad entitami v jednotlivých doménách. Zprávy o událostech jsou dále šířeny v celém systému k využití u dalších servis, které by mohly tyto události zajímat. Všechny vydané události jsou také ukládány pro pozdější využití ve specializovaném skladu (Event Store). Kromě hlášení událostí o operacích a změnách se tím automaticky vytváří i záznam o historii, který lze později využít pro zpětnou rekonstrukci. Systém se díky tomu stává událostně řízený a událostně konzistentní [26].

Výhodou událostně řízeného přístupu je i možnost kromě aktuálního stavu získat také informace o historickém vývoji entit a jejich stavu od vzniku až právě po aktuální stav.

Každá uložená událost obsahuje atribut, který odkazuje na předchozí událost, na jejímž základě byla aktuální událost založena. Tento událostní archiv využít jako obsáhlou zálohu systému, kterou lze ukládat mimo systém do externího úložiště. V případě potřeby je tak možné zálohu přenést, a na jejímž základě celý systém obnovit do původního stavu. Tento způsob obnovení je možné použít v případě, že se rozhodneme vytvořit celý systém na úplně jiné platformě, jiném druhu databáze nebo způsobu ukládání dat. V případě migrace mezi strukturálně různými databázemi není nutné migrovat všechny data, ale pouze událostní log a ten pak nechat zkonzumovat službami [100][26].

V událostně řízené architektuře se vyskytují základní pojmy jako **Command**, **Event**, **Message Broker**, **Event Store**, **Publisher** a **Subscriber**, které tvoří jádro systému založeného na návrhovém vzoru **event sourcingu**.

**Command** je rozkaz a povel, který je cíleně adresován servisu. Je to příkaz k vykonání přesné operace nad konkrétní entitou. Podle svého určení by měl být

také výstižně a jasně pojmenován. Doporučený standard je využít označení příkazu a název formulovat přikazovací formou jazyka objasňující aktivitu. Srozumitelnost je velmi důležitá, protože příkaz je prostřednictvím API vystaven jako hlavní komunikační objekt pro všechny servisy a externí prvky. Úmysl musí být zřejmý [39].

Například příkaz pro operaci vytvoření nového uživatele lze zadat jako „cmdVytvorNovyUzivatel“. Příkaz dále obsahuje informaci o původci, datumu zadání a další potřebné údaje o uživatelském účtu. U názvu je také vhodné, aby pro přehlednost uváděl název entity, případně korespondoval s modelem určeným pro následné čtení [39][26].

Po přijetí příkazu servisem je provedena validace, zpracování, uložení entity a vydání události oznamující provedení. Zpětná vazba může být v případě synchronního zpracování odeslána klientové okamžitě nebo až později na základě vydané události o provedení. V každém případě je, nezávisle na tom, zda konzumována, zpráva vydána a uložena do Event Store [39][26].

**Event** jako událost je záznam o skutečnosti, že v systému došlo ke změně nebo vykonání nějaké akce. Jak bylo zmíněno výše, na jeho základě ostatní servisy reagují a provádějí akce nebo vydávají další příkazy. Na rozdíl od příkazu by měla být událost pojmenována v minulém čase a opět ve srozumitelném a pochopitelném formátu. Ideálním způsobem opět v názvu uvést, že se jedná o událost, uvést název entity a vykonanou akci v minulém čase. Například „evtNovyUzivatelVytvoren“.

Událost je v rámci systému hlavním nositelem pravdy o stavu entity. Je ukládána a v případě nutnosti je možné je znova vydat do systému a stav entit obnovit.

Hlavním rysem **Eventu** je jeho **neodstranitelnost** z důvodu zachování konzistentního stavu systému. V případě nutnosti provést inverzní operaci, nelze událost smazat, ale je nutné vydat kompenzační událost, která místo vymazání předchozího eventu jeho obsah anuluje [39][26].

**Publisher a Subscriber** je označení pro dva režimy činnosti servisů. Režim **publisher** znamená, že servis je původcem události, kterou zpracovával, a kterou publikuje do systému. V režimu **Subscriber** je servis přihlášen k odběru událostí. Svůj zájem

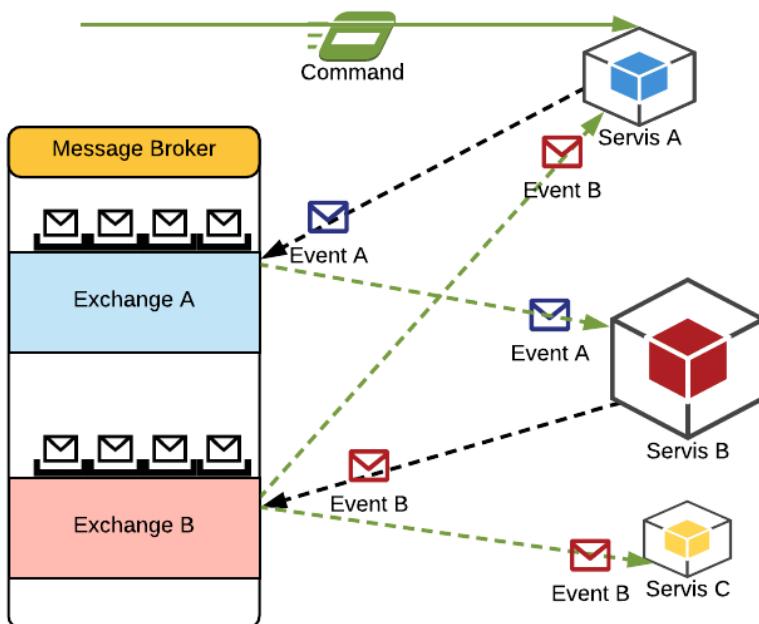
konzumovat vybrané servisy oznámí prostředníkovi a ten je této servise v případě vzniku zájmových událostí zašle [39][26][107].

Způsob tvorby názvu příkazů a událostí je dobré určit v předpise při modelování systému. Je vhodné používat jednotnou formu, uloženou například ve sdílené knihovně a jasně určit správce, který bude mít pravomoc tuto knihovnu upravit. Důležité je tuto pravomoc centralizovat a neponechat jí distribuovanou volně na vývojových týmech spolupracujících na projektu z důvodu zabránění nekontrolované tvorby svým významem duplicitních událostí.

Tvorba vydávaných událostí by měla korespondovat s doménovým návrhem a kopírovat reálné aktivity a procesy v rámci cílové organizace, pro kterou je systém navržen.

Postup pak odpovídá skutečné pracovní činnosti. Například pokud je v podniku vydán příkaz k činnosti konkrétnímu zaměstnanci, ten zahájí práci a po jejím splnění vydává hlášení o jejím dokončení. S tímto hlášením nemusí obíhat a hledat v které kanceláři je někdo koho by to zajímal. Odešle ho na standardizované rozhraní nebo místo, kam mají přístup všichni ostatní zaměstnanci. V případě, že někoho toto hlášení zajímá, dostane upomínku a může si hlášení vyzvednout a zpracovat na jeho základě své úkony a opět podá hlášení. Vytváří se tak tedy kartotéka všech operací, která je zároveň auditem a může být monitorována.

V systému tedy servisy reagují podle sebe na vzniklé události, ale nemusí je nezbytně zajímat jaký další endpoint mají po splnění zavolat, protože to už pro svou práci nepotřebují a není to v popisu jejich činnosti. Prostředník, kam zasílají svá hlášení a ke kterému se zájemci přihlašují, se označuje jako **Message broker**.



Obrázek 15 Schéma Message broker Zdroj: vlastní tvorba

**Message Broker** je prostředníkem, který je vyplňuje volný prostor mezi servisy a je kontaktním místem pro zpracování a výměnu všech vydaných hlášení. V rámci decentralizovaného systému nám message broker poskytuje možnost vykonávat operace vyžadující určitou míru atomicity. Je schopen potvrdit převzetí události i potvrdit jejich vyzvednutí. Pro potřeby klienta, který vyžaduje operaci provedenou v režimu atomicity, je možnost poskytnout pseudotransakci ze strany message brokera nebo potvrzující událost, na kterou musí svým způsobem. Garance je slabší než princip ACID (*atomicity consistency isolation, durability, ACID*), který lze udržet v relačně strukturovaných databázích. Mechanismy událostního řízení a publikování událostí nám naopak poskytuje jistotu v podobě událostní celistvosti (*Eventual consistency*). Jistota centrálního bodu pravdy je držena v jednom místě pomocí vrstveného logu událostí. Tento princip je historicky ověřený a je totožný s principy uplatňovanými v bankovnictví ještě před začátkem využití informačních technologií [79].

**Event store** slouží jako sklad a log událostí a je hlavním zdrojem pravdy o systému. Funguje jako samostatný servis, který obsahuje jednu databázi o jedné tabulce, která je strukturovaná a předem určena na zápis velkého množství záznamů. Každý záznam obsahuje svůj hlavní identifikátor, identifikátor předchůdce

a strukturovanou zprávu. Případně je možné pro přehlednost doplnit další údaje jako datum vytvoření, název události nebo připojit zpracovaný příkaz, podle kterého událost vznikla. V případě rozsáhlejších aplikací může datový sklad splňovat požadavky na zpracování v režimu Big Data. Především v rozsáhlých systémech jsou tyto charakteristiky u komponenty Event Store jasně pozorovatelné. Díky vysoké míře komunikace mezi servisy požadavek velkého objemu a rychlého nárůstu záznamů. Jediná operace ze strany klienta může díky nastavené konzumaci a zájmu o událost vyvolat desítky následně vydaných událostí. Pro účely analytického zpracování je vhodné zařadit další popisné atributy [79].

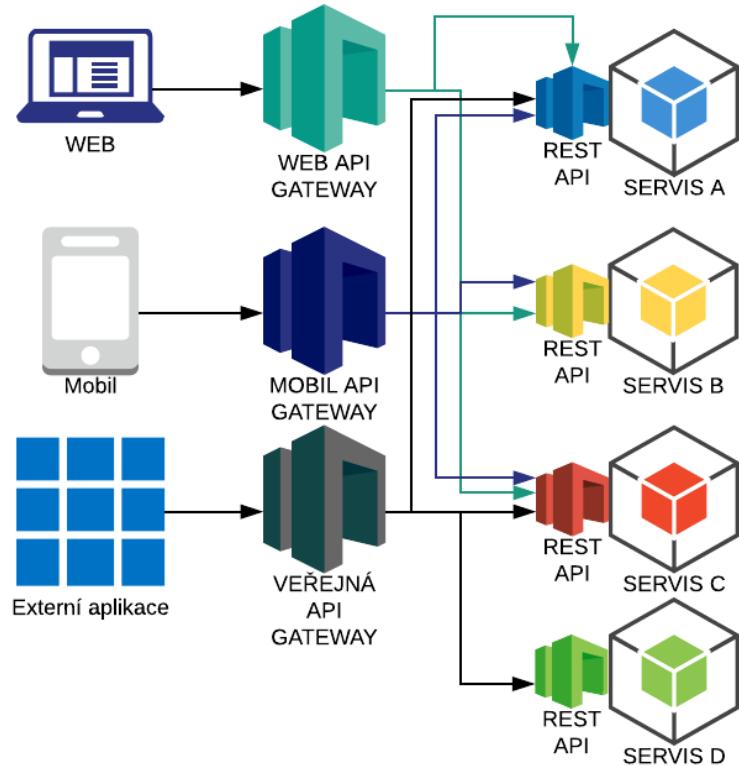
## 4.8 API Gateway

Až do této kapitoly bylo v této práci řešeno fungování jednotlivých částí systému v mikroservisní architektuře pouze z pohledu funkcí vnitřního prostředí a vzájemných vazeb mezi komponentami. Z vnitřního pohledu jsou komunikace a vzájemné vztahy mezi servisy zajištěny. Mechanizmy komunikace sice umožňují přímé napojení servisy na klientské aplikace systému, ale tento způsob by byl velmi nepraktický, zranitelný z pohledu bezpečnosti a složitý z pohledu škálování výkonu a hlídání všech vazeb v procesech [89][63][29].

Klient by musel znát všechny potřebné servisy a jejich konkrétní rozhraní. Pokud by měl k dispozici dokumentaci vzájemných vazeb a znal způsob orchestrace nebo choreografie celého systému, byl by tento způsob plně funkční. Takový způsob využití by přinesl příliš mnoho komplikací, rizik a nežádoucí přenesení logiky na klienta. Logika procesů a vnitřních vazeb by měla být zapouzdřena uvnitř systému. Pro odstranění této komplikace je nutné vytvořit jednotící prvek a přístupový bod jako kontrolovatelnou vstupní bránu celého systému [89][63][29].

**API Gateway** je označení pro tuto vstupní bránu. Je to řešení, jak odstranit nutnost na straně klienta vyznat se v drobně rozkouskovaném distribuovaném systému a bludišti vnitřní komunikačních pravidel. Tato vstupní brána přístup obaluje jednotným způsobem volání a je vhodným místem pro řešení bezpečnostní kontroly, provedení autorizace přístupu a správy výkonu. Přístupové rozhraní lze

díky tomu také přizpůsobit specifickým potřebám různých klientů v pohledu použitého zařízení (např. standardní PC, terminál, tablet nebo mobil). Lze vystavit různé metody pro různé druhy webových aplikací nebo tlusté klientské aplikace. Klientská a systémová část je pak rozdělena na **Frontend** a **Backend** [79][58][14].



Obrázek 16 Schéma Backend For Frontend (BFF) Zdroj: [58], vlastní úprava

Jakým způsobem pak v tomto rozdělení zvolit těžiště logiky a využití zdrojů se zabývají další návrhové vzory. Například vzor **BFF** (Backend For Frontend), který právě řeší rozdělení univerzální API Gateway na specializované části pro různá zařízení například z pohledu šířky displeje nebo nároků na přenos dat [14].

V souhrnu tedy API Gateway zapouzdřuje a uceluje celý systém vzhledem ke svému okolí a umožňuje multiplatformní komunikaci směrem od klienta [79][29].

## 4.9 Monitoring

V systému s velkým počtem pohyblivých částí jsou mechanismy zohledňující stabilitu, kvalitní monitoring a preventivní testování funkčnosti velmi důležitou součástí návrhu a provozu. V MA jsou všechny funkce koncentrovány do celistvého

celku a selhání některé se projeví v celém systému. V MSA je oblast testování a monitoringu podstatně komplikovanější. Chybovost v rámci systému je izolována pro každou servisu zvlášť, proto musí i testování být rozděleno individuálně. Dále jsou zapojeny další prvky jako message broker nebo API Gateway, které jsou klíčové pro zajištění správné činnosti systému. Může také dojít k logické chybě, která se nemusí projevit chybovým hlášením, ale pouze nevalidním chováním, které je složitě odhalitelné [63].

V rámci prevence je vhodné přemýšlet už při konstrukci systému přemýšlet o toleranci chybovosti a rovnou postupně zavádět monitorovací mechanismy na všechny případná místa selhání. Je to součástí prvních kroků analýzy, kdy je potřeba se zaměřit vždy na několik základních pojmu a určit **body selhání**. V ekosystému MSA se dělí body selhání do několika tříd na **interní, selhání závislosti a selhání autority** [28].

**Interní selhání** označuje oblast chybovosti uzavřenou v oblasti kódu a logiky jednoho servisu. Izolace a řešení chyb tohoto typu je záležitostí kvalitně propracovaného návrhu, dodržování stanoveného standardu pro vývoj a provedení dostatečného testování před uvedením do provozu. Dodržení praktik uvádění do provozu pak může být realizováno automatizovaně včetně vyhodnocování naměřených dat z monitoringu u všech zájmových metrik [28].

**Selhání závislosti** se odhaluje složitěji. V tomto případě není zřejmá jasná identifikace původce, který chybu do prostoru mezi servisy šíří. Důležité je tedy dodržení návrhu a izolace chyb na úrovni servis, zdokumentování tvořených vazeb, správa jasně stanovou autoritou a možnost kompenzačního řešení vydaných událostí. Chybovost může odhalit analýza vydávaných událostí podle nastavených scénářů kopírující procesy [28].

**Selhání autority** je způsobeno výpadkem některého z klíčových prvků klíčových prvků systémů. Označují prvky, které tvoří možný centrální bod selhání jako je API Gateway, message Broker nebo event store. Každý servis může sám o sobě pracovat korektně, ale pokud spolupracují na dosažení komplexních procesů, musí být i pro tyto případy zajištěna odolnost už při návrhu. Řešením je schopnost opakování

navázat spojení s orchestračním prvkem, zpětná kontrola stavu a konzumace zpráv za účelem dorovnání stavu. U centrálních prvků je pak možnost zapojit mechanismy pro rychlé obnovení, a pokud to není možné, tak tento prvek provozovat v několika instancích. Důležité je pak odhalit pomocí kvalitního monitoringu chybu včas a rychle zjednat nápravu [28].

Je tedy důležité počítat s obnovovacími mechanismy a propracovanou odolností vůči chybám už při návrhu systému. Během provozu je vhodnou metodou řízené zkušební selhání. Naplánovaný scénář je proveden v předem avizovaném čase, kdy je systém v nemenší zátěži. Je lepší chybu způsobit řízeně, provést měření výkonosti obnovovacích mechanismů a odhalit případné nedostatky dřív, než nastanou v nečekanou chvíli, kdy je to nejméně vhodné [28].

Ve všech bodech je zmíněn kvalitní monitoring, který obsahuje sledování stanovených metrik a kontroly zdraví všech prvků. Situace je v tomto ohledu dále komplikovaná, pokud je provozováno více instancí stejné třídy na několika serverech. V tomto případě je nutné ještě zařadit navíc identifikaci jednotlivých instancí do logovacích mechanismů. Sledované metriky mohou dále obsahovat například informaci o původci volání, adresátovi, stavové kódy protokolu http obdržené z API, hardwarové nároky, počet přístupů, odezvu a další parametry potřebné pro analýzu stanovených smluvních požadavků (service level agreement, SLA) [28][63].

Monitoring je vhodné provádět na několika místech systému. Prvním možností je provoz zachytávat **už na úrovni Gateway** a logovat všechny neúspěšně odpovědi http requestů. Dalším způsobem je zavést monitoring v místě, **kde je uložena logika** složených operací, a to v případě orchestrace na těch částech, kde dochází ke zpracování a distribuci zpráv. V případě choreografie je pak logika **na straně každé servisy**. Ty po naměření hodnot vystaví hlášení na domluvený endpoint na kterém si další specializovaný servis hlášení vyzvedne, provede agregaci na jednom místě a dále je vyhodnotí ve svém prostředí a dá k dispozici v přehledném uživatelském rozhraní, případně provede okamžitou opravu. Pro tento účel je možné využít několik specializovaných nástrojů, které jsou popsány dále v části práce o použitých technologiích [28][63].

Nejlepším způsobem, jak řešit chyby systému je jim předcházet. Díky monitoring je můžeme odhalit po jejich vzniknutí, ale díky testování je možné jim předcházet.

**Testování** je vhodné zavést u každého servisu formou automatizovaných testů. U rozsáhlých distribuovaných systémů je tento způsob nejfektivnější možností, jak se přesvědčit o správné funkčnosti všech jeho pohyblivých částí. Zavádění testovacích mechanismů znamená více práce při úvodních fázích vývoje, ale výrazně pak přispívá ke stabilitě systému při jeho dalším rozvoji a provozu [63].

Automatizované testy jsou rozděleny do skupin podle zaměření na oblast kódu, zátěže a náhodných selhání [28].

**Testování kódu** spočívá v kontrole, zda byl kód strukturován podle standardu projektu a jestli jsou metody pojmenované podle jednotného standardu. Dodržení konvence je důležité pro přehlednost, rozšiřitelnost, analýzu kódu a nezávislost na autorovi. V této oblasti jsou také zahrnuty Unit testy, aplikované na určitou funkcionality nebo část servisu. Integrační testy následně spojují několik testovaných částí dohromady a porovnávají jejich vzájemnou kooperaci. V případě distribuované struktury propojené pomocí vzdáleného volání metod je nutné provádět i End-To-End testy. V každém místě, kde probíhá zpracování operace umístit kontrolní testovací bod napříč celým distribuovaným procesem od začátku až po konec. Testy je pak možné automatizovat do skupiny tak, aby byly prováděny před každým nasazením servisů do provozu a byla automaticky ověřena jejich funkčnost [28].

Účelem **zátěžových testů** je zjistit a změřit výkonost vybrané části systému a kontrola, zda tato část splňuje předepsané a očekávané požadavky. U těchto testů je důležité vhodné rozvržení jejich načasování a v případě testování na produkčním prostředí zajistit dostatečnou izolovanost od vazeb na ostatní prvky, které nejsou součástí testu [28].

**Náhodné testování** je cílené vyvolání chaosu v systému, jeho vyvedení z rovnováhy a zjištění, jak je systém odolný a jak se dokáže sám vyrovnat s výpadkem. Testy jsou předem naplánovány, aby neohrozili reálný provoz, a jsou pečlivě monitorovány [28].

Hlavním smyslem monitoringu a testování je zavést měřící mechanismy a s jejich pomocí neustále systém monitorovat. Na základě získaných údajů pak adekvátně reagovat a pracovat na zlepšení, odstraňovat slabá místa a zvyšovat tak celkovou odolnost systému proti výpadkům.

## 4.10 Ekosystém

Mikroservisní architektura je tvořena souborem výše popsaných termínu, mechanismů, návrhových vzorů. Díky jejich objasnění více upřesnit definici celé architektury a jakým způsobem je systém v ní vytvoření postaven. **Systém vybudovaný v MSA je distribuovaný a decentralizovaný, složený z kontextově ohraničených autonomních prvků spravující svou vlastní funkční doménu, provozovaný ve svém virtualizovaném prostředí, servisů navzájem schopných se nacházet a komunikovat standardizovanými protokoly, vystupujícími na venek jako jeden chybově odolný, stabilní a testovatelný celek.** Je to také multiplatformní systém schopný provozovat prvky založené na různých technologiích, programovacích jazycích a systém schopný udržovat kromě svého aktuálního stavu i historický kontext díky zpracování a logování všech proběhlých událostí. Za pomoci logovacích mechanizmů je možné provádět monitoring a reagovat na případné výpadky velkého rozsahu dalšími mechanizmy pro obnovu do původního stavu.

Dále se systémy vytvořené v MSA vyznačují vlastnostmi, jako jsou Škálovatelnost, Dostupnost, Odolnost, Flexibilita, Nezávislost, Decentralizace řízení, Izolace Chybovosti, Automatizace a Udržení rozvoje. Škálovatelnost je zařízena nástroji virtualizačního prostředí pracujících s kontejnery na úrovni API Gateway [61][58].

Dostupnost je zaručena díky implementaci testovacích mechanizmů, které lze pro tyto potřeby automatizovat. V celém ekosystému je aplikován monitoring všech případných bodů selhání a systém je navržen tak, aby vzniklé chyby byly co nejvíce izolovány ve svém doménovém kontextu [61][58].

Z pohledu vývoje je systém velmi flexibilní. Možnost připojit, upravit nebo odebrat součásti bez nutnosti zastavení provozu celku je jeho velkou výhodou. Vývoj systému lze pak rozdělit na jasné části a ty přidělit do odpovědnosti a správy

jednotlivým týmům nebo osobám. Ekosystém je díky tomu připravený k rozvoji a úpravám jednotlivých částí s využitím agilních metodik v pracovních postupech. Díky tomu lze rychleji dosahovat stanovených cílů, přijímat rychlá opatření v reakci na změny požadavků a zajištění nepřetržitého potupu směrem ke stanoveným cílům [61][58].

## 4.11 Výhody a nevýhody

**Výhodou MSA je multiplatformní prostřední, univerzálnost, opakovatelnost, škálovatelnost, agilita, izolace chybovosti a odolnost, udržovatelnost.**

**Multiplatformní** prostředí a **universálnost** je tou nejvýraznější výhodnou vlastností stejně tak jako u SOA na které je MSA založena. Umožňuje využití širokého spektra technologií při tvorbě komponentů nebo i funkcí na nižší úrovni. Dále je to možnost libovolně zapojit i komponenty třetích stran nebo spolupráce s dodavateli [63][18][17][24].

**Opakovatelnost** a možnost využití jednotlivých servis nebo jejich částí v různých místech systému je výhodou, díky které lze lépe budovat nové vazby a nové procesy [63][18][17][24].

**Škálovatelnost** je možné provést horizontálně i vertikálně. Díky úspornějšímu způsobu virtualizace prostřednictvím kontejnerů jsou v obou případech sníženy celkové náklady. Není nutné navýšovat kapacity pro celý systém, ale je možnost navýšit výkon pouze pro ty nejvíce vytížené části. Toto navýšení je díky velmi drobnému členění efektivní a přesně zaměřené podle konkrétní potřeby. Nedochází tak k nadměrné alokaci zdrojů v místech, kde to není nezbytně nutné [63][18][17].

**Agilita** a podpora agilních metodik vývoje je zajištěna díky možnosti efektivně rozdělit jednotlivé oblasti do malých celků, za které lze přidělit odpovědnost. Lze provádět časté a rychlé iterace vývoje, rychle nahradit prvek nebo změnit kombinaci v rámci procesu. Lze lépe plnit potřeby reálného prostředí a přizpůsobovat se změnám v podnikových procesech [63][18][17][24].

**Izolace chybovosti a odolnost** jsou hlavní znaky správně realizované mikroservisní architektury. Při dodržení samostatnosti a volných vazeb je případná chyba izolována pouze v servisu, ve kterém vznikla. Díky tomu je zbytek systému stále funkční. V systému MSA jsou také často zavedeny techniky pro monitoring a testování, které v případě selhání chybu včas detekují, je možné i vyřešit a zvýšit tak odolnost systému [7][15].

**Snadná údržba** je umožněna díky malé velikosti servisů, jejich omezené odpovědnosti, automatizovanému testování a nasazení do produkce. Na příklad díky virtualizačnímu nástroji Docker lze připravit otestovaný Docker image, který se vloží do úložiště a s minimální prodlevou se spustí jako nový kontejner [63][18][17].

**Nevýhody** MSA jsou vysoká **komplexnost návrhu**, **velký objem komunikace**, **cena** a **náklady**, **náročnost testování**, monitoringu a verzování.

**Komplexnost systému** je překážkou při úvodních fázích návrhu systému. U aplikací drobného rozsahu, kde se neočekává velký budoucí rozvoj, je stále lepší zvolit monolitický přístup. Zároveň je v systému zapojeno mnoho různých technologií, které kladou na vývojáře velké nároky a na jejich širokou úroveň znalostí [24].

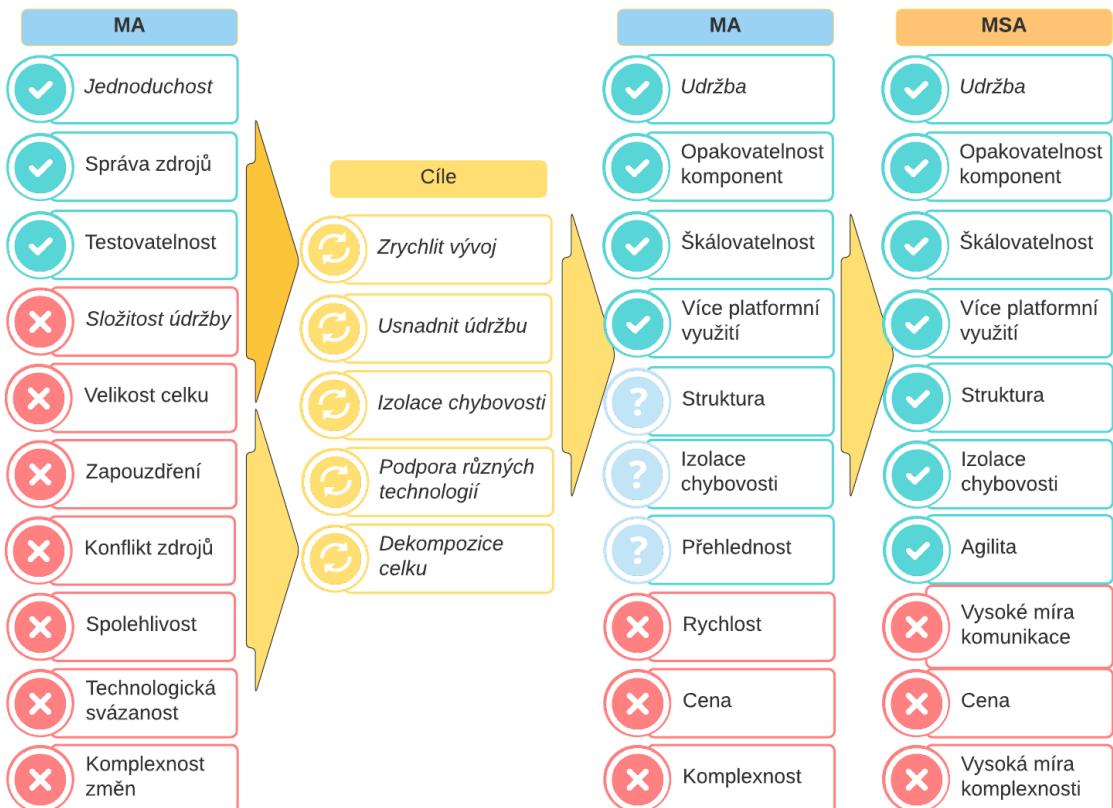
Kvůli nutnosti servisů spolu často a ve velkém objemu komunikovat je vyvíjena i vysoká zátěž na komunikační infrastrukturu. Při běhu systému v MSA dochází k mnohem většímu využití přenosových kapacit než v případě SOA. Servisy často i při základních operacích zasílají velké množství zpráv. Pro potřeby monitoringu je také v častých a pravidelných intervalech zasíláno hlášení stavu. Pro zachování efektivnosti a rychlé odezvy v systému je tedy při vyšší komunikaci vyžadováno kvalitnější a robustnější prostředí, a to znamená vyšší náklady na pořízení a údržbu [63][18][17][24].

**Testování a monitoring** je mnohem složitější v distribuovaném systému než v monolitu. V případě mikroservis jsou výhody plynoucí z drobného členění zároveň také nevýhodou pro návrh testovacích mechanizmů [63][18][17][24] .

**Verzování** je důležité pro zachování výhody libovolné nahraditelnosti servis a zpětné kompatibilitě s publikovanými zprávami v eventstore. Pokud je funkcionalita servisu

změněna a zahájí vydávaní nové verze události, musí být stále schopná zpracovat i ty události, které byly publikovány na jejím vzniku [63][18][17].

**Cena** se navýšuje na jedné straně nutností zajistit kvalitní komunikaci i při vzrůstajícím počtu zařazených komponent. Dále se pak finanční nároky zvedají při nutnosti využití nástrojů a komponentů třetích stran a nákupu potřebných licencí [63][18][17].



Obrázek 17 Výhody a nevýhody MA, SOA, MSA Zdroj: vlastní tvorba

## 5 Technologie v MSA

Před realizací projektu nutné provést základní rozhodnutí, které technologie pro vývoj zvolit. Mikroservisní ekosystém je technologicky univerzální a podporuje široké spektrum možností. Každá organizace nebo tým má své preference na vývoj, a proto je i velké množství projektů realizovaných různými způsoby a kombinacemi nástrojů.

Většina ekosystémů v MSA je provozována na variantách operačního systému Linux, převážně tedy CentOS, Debian nebo Ubuntu. Další možností je jít cestou společnosti Microsoft a technologiích založených na .NET [28][4].

Pro přijmutí rozhodnutí o základní platformě je kromě osobních nebo podnikových preferencí a pravidel vhodné zvážit varianty a spektrum technologií, které v dané variantě můžeme využít. Nejpoužívanějšími programovacími jazyky jsou **Java**, **Python**, **C++**, **Node.Js** a jazyky **platformy .NET** [4][105].

**Java** je nejrozšířenější platforma při tvorbě aplikací v MSA. Syntaxe je dobře čitelná, existuje široká základna vývojářů a k využití je velký výběr vhodných podpůrných nástrojů nebo hotových komponent. Obsahuje standardy jako JAX-RS pro tvorbu API, JPA pro práci s daty a CDI pro vzájemné předávání závislostí mezi prvky [4].

Je primárně podporovanou platformou pro nástroje service discovery jako jsou **Consul**, **Netflix Eureka** nebo **Amalgam8**. Při vytváření architektury je nejčastěji využíván framework **SpringBoot**, **Dropwizard**, **Restlet** nebo **Spark** [4].

**Python** je další často užívanou platformou. V porovnání s ostatními frameworky umožnuje rychlejší tvorbu prototypů, využívá RESTfull Api pro tvorbu komunikační rozhraní a je kompatibilní s jazyky ASP a PHP pro tvorbu front-endu. Společně s Python se využívají frameworky jako **Flask**, **Bottle**, **Nameko** nebo **CherryPy**. Pro message broker **RabbitMQ** je Python první doporučenou variantou. Pro komunikační rozhraní v rámci backendu pak **Flacom** [4][105].

**C++** je programovací objektově orientovaný jazyk. Velké uplatnění má v oblasti databázových servisů, automatizovaných aplikací a robotiky. Jím vytvořené aplikace

Ize díky knihovně REST SDK se syntaxí C++ 11 provozovat na Windows, Linux nebo Mac OS bez nutnosti velkého úsilí a práce na přizpůsobení kódu. Výhodou je vysoká opakovaná využitelnost komponent, snadná konfigurace na všech operačních systémech a nízká míra vazby mezi konzumentem a poskytovatelem [4].

**Node JS** je velmi populární platforma pro tvorbu mikroservisně založených aplikací. Node JS je postaven na prostředí V8, u něhož jsou velmi dobré výkonové charakteristiky při provozu mikroservisů. Využívá se především v situacích, kdy je kladen důraz na snížení ceny vývoje a zároveň na rychlou produktivitu a výkon systému [4][105].

**.NET** je platforma několika jazyků pro vývoj aplikací, servisů s vystaveným API. Poskytuje podporu pro vytváření image pro nasazení v kontejnerech nástroje Docker. Zároveň je využíván pro široké spektrum typů aplikací od desktopových, mobilní, webu, her a dalších. V **.NET Core** je možné propojit více technologií v celém systému, zapojit část servisů vystavěných Java nebo Node JS a je možné provozovat aplikaci na většině používaných cloud platformách [4][105].

V dalších podkapitolách této části je výběr a popis několika nástrojů a doplňku, které se uplatňují při konstrukci aplikací v mikroservisní architektuře a které budou i následně využity v praktické části.

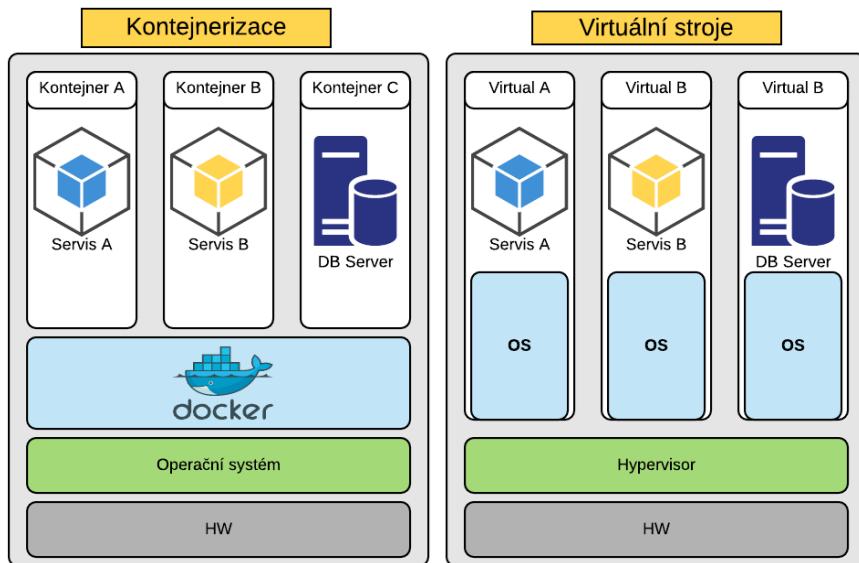
## 5.1 Docker

**Docker** je nástroj pomocí kterého lze podle konfiguračního souboru (Docker File) zabalit aplikaci se všemi potřebnými závislostmi a knihovnami do jednoho balíku (Docker Image). Ten je pak spuštěn ve virtualizačním prostředí Dockeru a provozován jako spravovatelný kontejner (Docker Container) [106].

**Kontejnerizace** byla zmíněna v kapitole o distribuovaném systému (4.3). Kontejner je objekt, s jehož pomocí je možné různorodá prostředí a technologie zaobalit do standardizovaného obalu, nakládat s nimi univerzálním způsobem, spouštět a provozovat ve standardizovaném prostředí Dockeru.

Díky kontejnerizaci je možné vzít libovolné technologie a nezávisle na podporovaném operačním systému je provozovat na libovolné platformě. Například je možné virtualizovat a provozovat Windows kontejner na platformě Linux a obráceně. Při výběru a volby technologie tedy nejsme limitováni hranicemi a možnostmi operačních systémů a můžeme se rozhodnout co je pro kterou část nejvhodnější nebo kde má společnost své technologické preference nebo nakoupené licence [106].

Velkým rozdílem oproti ostatním virtualizačním nástrojům je samotný provoz a vytváření kontejneru. Běžně je replikován celý operační systém v plném rozsahu a ten je virtualizován. Potom tedy na serveru běží několik plných instancí operačního systému, a to má samozřejmě adekvátně vysoké hardwarové nároky. V případě Dockeru je na serveru instalován pouze základní hostitelský operační systém a jednotlivé kontejnery jsou pouze deriváty a zmenšené konfigurace hostovaných operačních systémů. To je možné díky Docker Engine, který zajistí překlad požadavků z kontejneru na hostitelský systém. Díky tomu je možné replikovat jen to nejnuttnejší a ušetřit tak výpočetní výkon na provoz aplikace samotné. Tím je snížena i cena na pořízení hardware [106][22].



Obrázek 18 Kontejnerizace a Virtuální stroje Zdroj: [21], vlastní úprava

**Docker File** je textový konfigurační soubor obsahující příkazy, které jsou využity nástrojem Docker při spuštění komplikace servisu. Obsahuje cestu ke všem

potřebným souborům a závislostem, určuje technologii obsahu kontejneru a určení typu podle operačního systému, ve kterém bude kontejner provozován. Docker u kontejnerů rozlišuje dva typy, Linux nebo Windows. Více rozšířené je užití Linux kontejneru. Vytvoření na základě DockerFile pomocí příkazové řádky příkazem docker build [106][22].

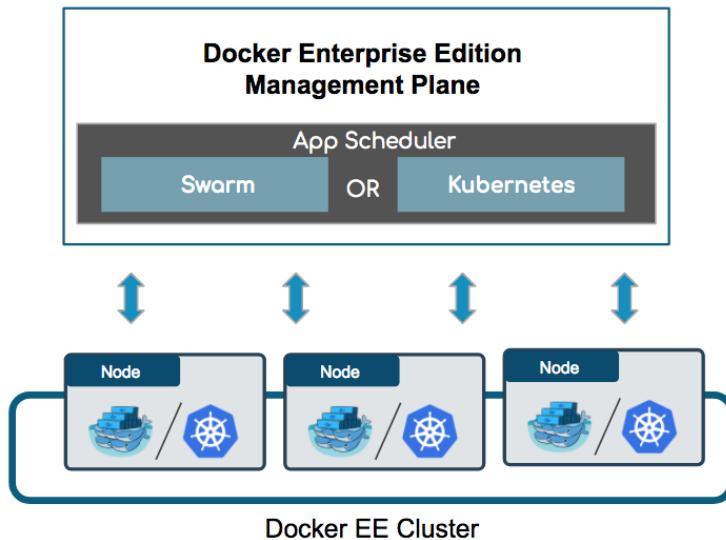
**Docker Image** je šablona pro kontejner, která lze opakovaně spouštět a přenášet mezi testovacím a produkčním prostředím. Je to základní stavební blok distribuovaného systému provozovanému nástrojem Docker. Prostřednictvím těchto souborů lze snadno a rychle vytvořit vývojové prostředí v případě příchodu nového člena týmu nebo reinstalace systému. Lze také spouštět jen ty části systému, které jsou nutné pro vývoj konkrétní servisů a které jsou společně s ní nutné k realizaci a modelování procesu [106][22].

Pro virtualizaci a současnemu spuštění více servisů se společnou konfigurací se používá **Docker Compose**. Je to konfigurační soubor sloužící pro nastavení vazeb a vlastností systému jako celku. Tento soubor ve struktuře YAML obsahuje seznam všech komponent, které budou postupně spuštěny v sestavě samostatných kontejnerů jedním příkazem. Do souboru se uvádí kromě servisů také nastavení sítě, cesty pro interní úložiště kontejneru, nastavení databázových a dalších komponent, message broker a další případné obslužné nástroje. V Docker compose se také určuje překlad z interních komunikačních portů aplikace na porty vystavené svému okolí. Dále se také nastaví názvy kontejnerů a synonyma pro vzájemnou orchestraci a komunikaci mezi servisy. Po spuštění kompozice jsou postupně připravovány docker image podle předpisu v DockerFile a následně spuštěny kontejnery pomocí předpisu DockerCompose [106].

**Docker Swarm** je o stupeň vyšší virtualizační vrstva než kontejner. Pomocí nástroje Docker swarm je vytvořena a spravována sestava několika instancí docker se svým virtualizovaným prostředím. Je to technika horizontálního navýšení výkonu a distribuce systému v případě rozsáhlých projektů. Takto vytvořená sestava serverů se označuje jako **Docker Cluster** [106].

## 5.2 Kubernetes

Kubernetes (K8s) je open-source nástroj od společnosti Google pro správu a orchestraci kontejnerů. Jeho prostřednictvím lze kontejnery nasazovat do produkčního prostředí, provádět škálování a load balancing a spravovat celkový výkon a vlastnosti provozovaného systému [19].



Obrázek 19 Schéma Kubernetes Zdroj: [25]

Podstatou tohoto nástroje kontejnerizace samotná, i když poskytuje více možností než Docker Swarm. K8s umožňuje po instalaci doplňků pro vytvoření clusteru, má lepší uživatelské rozhraní a podporu automatizace škálování procesů. Instalace je ale oproti Docker Swarm více komplikovaná. Proto je využívána vzájemná kombinace Dockeru pro virtualizaci a Kubernetes pro správu [106][25][101].

Sestava aplikacích kontejnerů vytvořená v Docker je využita jako jeden kontejner, v K8s označovaný jako Pods. S Podem je následně nakládáno obdobně jako s kontejnerem a je nad ním prováděna správa a škálování univerzálním způsobem podle potřeb systému [106][101].

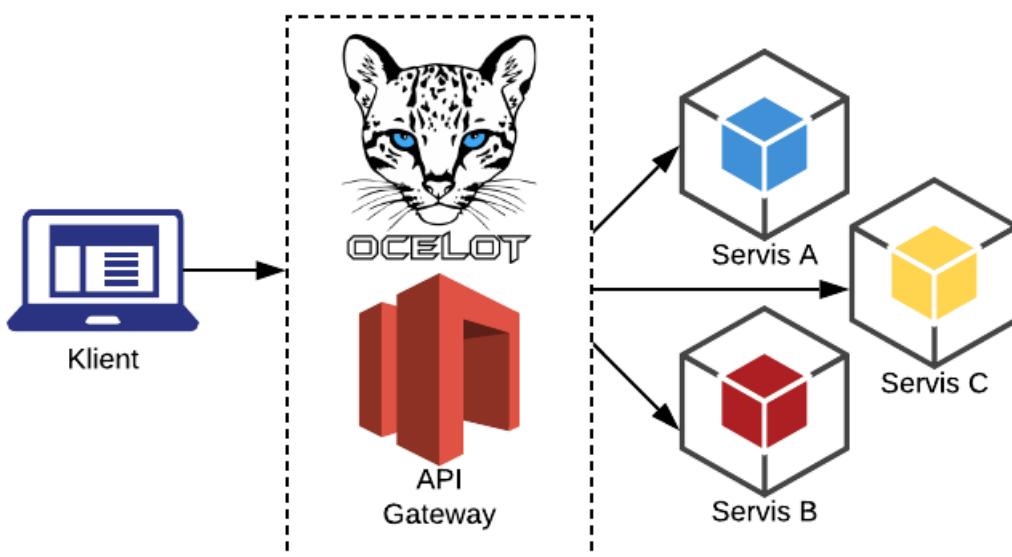
Kubernetes je pro potřeby vývoje ve své odlehčené verzi zakomponován přímo jako modul nástroje Docker [106][25].

### 5.3 Ocelot

Pomocí Ocelot je možné zapouzdřit systém pod jednotným chováním skrze API Gateway. Jednou z variant je možnost si i vytvořit vlastní Gateway na míru bez využití dalšího nástroje jako je například právě Ocelot. Vlastní řešení má ale větší nároky časové a návrhové nároky a v pozdější fázi může docházet ke komplikacím při řízení výkonu, kompatibilitou s dalšími nástroji a případné další neočekávané chyby. Pro rozsáhlejší projekty je lepší využít ověřené řešení, s možností napojení a podpory ze strany velkého množství dalších nástrojů například k orchestraci nebo řízení výkonu [33].

Ocelot je nástroj doporučovaný pro využití v aplikacích založených na .NET Core a s jeho pomocí jsou odstraněny případná rizika a nutnost komplexního vývoje vlastní Gateway.

Ocelot provádí předávání a přesměrování volání http podle pravidel nastavených v konfiguračním souboru. Konfigurační soubor obsahuje pro každou službu DownStream a UpStream sekci, která určuje, na které adresy bude přesměrováno volání z okolního prostředí do vnitřního prostředí za Gateway. Dále je možné upravit nastavení, aby spolupracovalo s orchestrátory jako jsou například Kubernetes nebo Consul. Ocelot také obsahuje velkou databázi rozšíření pro integraci monitoringu u většiny nejčastěji používaných nástrojů [33].



Obrázek 20 Schéma API Gateway Ocelot Zdroj: [33], vlastní úprava

## 5.4 Kafka

**Apache Kafka** je platforma navržená na zpracování velkého množství zpráv v reálném čase. Podporuje integraci klientů z technologií Java, .NET, PHP, Ruby nebo Python [30].

Poskytuje službu prostředníka zpráv (message broker) v distribuovaném systému. Poskytuje možnost uložení zpráv po dobu nutnou k jejich zpracování, jejich další ukládání, třídění a jejich další přeposílání. Zprávy řadí do témat a v tématu jsou rozděleny do jednotlivých proudů, které čekají ke konzumaci [62].

Výsledkem proudu je nastavení aktuálního stavu, ale právě díky uchování i historických zpráv je možné rekonstruovat stav v libovolném časovém intervalu a.

Sledování vývoje a historie entit je důležitým analytickým nástrojem pro podporu obchodu, optimalizace obchodu a reakce na aktuální vývoj situace.

Platforma má široké využití sloužící jako zpracovatel zpráv, sledování aktivit, sběr metrik, aplikační log, zpracování proudu, jako technika pro rozbití pevných vazeb mezi prvky systému nebo jako sběrný prvek pro nástroje zabývajícími se zpracováním a analýzou velkých dat jako Hadoop nebo Spark [30][62].

Apache Kafka je využíván například společnostmi Netflix, Uber nebo LinkedIn v různé míře využití právě pro sledování aktivit nebo komunikaci ve svém distribuovaném mikroservisním systému. Základními prvky Apache Kafka jsou **zprávy, dávky, schémata a téma**. V procesech zpracování těchto prvků pak vystupují role **producenta, konzumenta, poskytovatele** a správce [30].

**Zprávy (Message)** jsou základní zpracovávanou komoditou. Je to obdoba záznamu nebo řádku tabulky v běžně používaných databázích. Zpráva v Kafka je jednoduché bytové pole obsahující všechny potřebné informace, které jsou důležité pro poskytovatele a konzumenta. Samotný obsah zprávy Kafka neřeší, ale je možné přidat další metadata která jsou být využita jako klíče pro přesnější směrování a třídění [30].

**Dávky (Batch)** jsou souborem několika zpráv, který je vytvářen za účelem rychlejšího zpracování. Sdružuje zprávy náležící jednomu tématu nebo schématu.

Pokud je směřováno více zpráv na jedno téma, může Kafka na úkor mírně zvýšené odesvy správy sdružit v dívce, provést kompresi a poslat je v jednom kroku a ušetřit tak přenosovou kapacitu sítě [30].

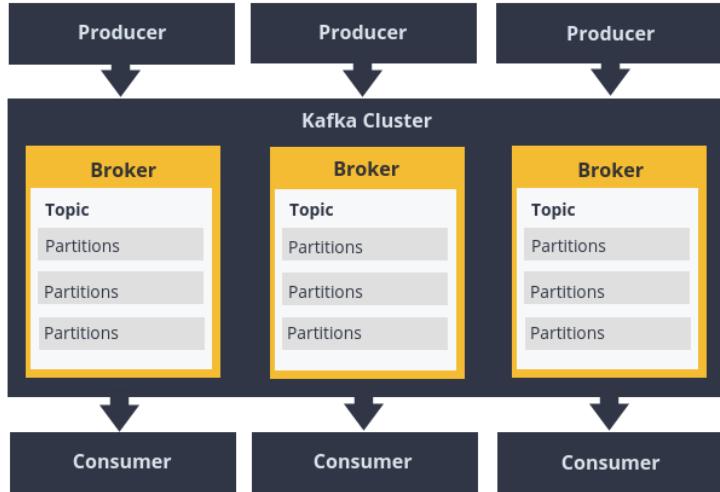
**Schéma (Scheme)** je šablona, která popisuje původně neprůhlednou zprávu. Je strukturováno ve formátu XML nebo JSON, čitelného a pochopitelného i člověkem a je možné za pomocí nástrojů díky této šabloně zasílané zprávy deserializovat. Pro konverze za pomoci schématu je využíváno pomocí rozšiřujícího frameworku Avro, který byl původně vyvinut pro práci s big data v Hadoop [30][62].

**Téma (Topic)** jsou kategorie pro vydávané právy. V analogii na databáze se jedná o tabulky nebo v souborovém systému adresáře. Témata se dále rozpadají do několika vláken tak, aby bylo umožněno rychlejší paralelní zpracování více zpráv v tématu. Celý datový tok skrze téma se označuje jako proud (stream). Pro zpracování proudů je opět další vydané rozšíření Kafka Stream, Apache Samza nebo Storm, které umožňují velice rychlé zpracování v reálném [30][62].

**Producent a Konzument (Producer, Consumer)** jsou role zákazníků v rámci procesu tvorby a zpracování zpráv. Umožňují tvorbu distribuovaného systému a asynchronní zpracování s volnou vazbou. Producenti vydávají zprávy, zasírají je do příslušných Témata a z nich pak jsou zasílány konzumentům, kteří se o téma zajímají [30].

**Poskytovatel (Broker)** je samotný Kafka server. Jeho hlavní rolí je zprávy přijímat, třídit, ukládat a poskytovat. Je navržen pro potřeby vysokého výkonu a zpracování milionů zpráv za minutu tak aby bylo možné ho škálovat, navyšovat kapacitu a umístit ho do shluku (clusteru) několika dalších Kafka serverů. V každém shluku vystupuje pak jeden server jako správce a vykonává nad celým shlukem kromě své vlastní činnosti a řídící operace [30][62].

Poskytovatel má také možnost nastavit limity velikosti, počtu nebo času po jakou dobu bude držet zprávy ve své paměti a poskytovat je s minimální odesvou. Například může držet informace o pohybu zásilky od jejího zahájení po doručení. Pak už nejsou tyto informace nezbytně nutné udržovat k okamžité konzumaci a mohou být odsunuta do logu drženého jiným Poskytovatelem s nastavením pro trvalé uložení [30][62].



Obrázek 21 Struktura MB – Kafka Zdroj: [30]

## 5.5 RabbitMQ

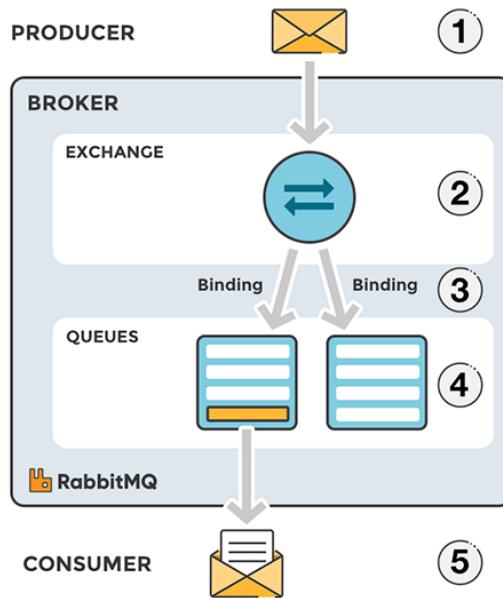
RabbitMQ byl vydaný v roce 2007 společností Pivotal. Od začátku roku 2020 se Pivotal začlenil pod společnost VMwave, která je součástí konsorcia Dell Technologies. Byl vyvinutý pro nasazení v SOA a stejně tak lze využít i v MSA. Vzájemnou komunikaci mezi servisy zajišťuje pomocí protokolů MQTT, AMQP a STOMP. Je možné ho využít v aplikacích na platformě Java, .NET, Ruby, Node.js a je podporován v mnoha dalších nástrojích [103].

Asynchronní přenos správ je zajištěn obdobně jako u Kafka. Hlavními částmi jsou **Fronta** (Queue), **Výměník** (Exchange), **Témata** (Topic) a role Producenta a Konzumenta zpráv.

Součástí nástroje je přehledné uživatelské rozhraní detailní informace o provozu systému, počtu předávaných zpráv a správy prostředků [103].

Je několik způsobů nastavení a vzorů, jak vystavět mechanismus předávání zpráv.

RabbitMQ je softwarový nástroj pro předávání zpráv a poskytování služeb message brokera. Servisy si díky tomu mohou předávat zprávy [103].



Obrázek 22 Struktura MB – RabbitMQ Zdroj: [103]

## 5.6 Swagger

Servisy komunikují mezi sebou a okolím prostřednictvím API. Jeho podobu je nutné popsat standardizovaným způsobem a vystavit ho na známé místo. Na základě takového popisu lze vytvořit v klientské aplikaci napojení a metody servisu začít využívat. Pro tyto účely slouží nástroj **Swagger**, známý také jako OpenApi. Lze jej, jako rozšiřující balíček, zakomponovat přímo do třídy API v servisu. Po nakonfigurování pak servis může svůj popis ve formátu YAML nebo JSON vystavit ke konzumaci. Formát je nezávislý na stylu nebo technologii, ve kterém je API servisů vytvořeno. Poskytuje také podporu formou vlastního přehledného uživatelského rozhraní. Díky uživatelskému rozhraní je možné metody i testovat, i když pro tyto účely je lepší využít nástroj **Postman**, který je k tomuto účelu přímo vytvořen a nabízí velké množství metod vhodných pro testování rozsáhlějších scénářů [92].

V případě využití technologií .NET Core je doporučeným doplňkem balíček **SwashBuckle.AspNetCore**, který je možné instalovat prostřednictvím knihovny Nuget.Org a který zprostředkovává funkce Swaggeru i v tomto prostředí. Dalším vhodným doplňkem je **NSwag**. Prostřednictvím NSwag jsou na základě

poskytnutého popisu API generovány třídy HTTP klienta, v C# nebo TypeScript, pro připojení a konzumaci aplikace k servisu, včetně obsažených modelů [92].

## 5.7 HealthCheck

Monitoring a kontrola byla v této práci zmíněna jako důležitá součást systému MSA. Logování a kontroly obsahují do jisté míry všechny zmíněné nástroje. U Docker je možné na úrovni kontejneru sledovat hlášení a využití zdrojů. Kubernetes, Consul i RabbitMQ mají své mechanismy na kontrolu stavu a výkonu. V .NET Core jsou knihovny na podporu logování a monitoringu, ale slouží spíše jako podpora pro využití externími nástroji [38].

Základní knihovnu v .NET Core AspNetCore.Diagnostics.HealthChecks lze rozšířit o další velké množství nástrojů. Tyto knihovny se iniciují při startu servisu a poskytují metriky o stavu prostřednictvím formátu JSON, který je dále zpracováván a předáván do grafických rozhraní. Hlášení je možné agregovat na úrovni API Gateway a vystavit a vizualizovat v jako celek v přehledném dashboardu. Tak je možné kromě automatizovaných testů monitorovat aktuální aktivitu, plánovat testování a sledovat vývoj vzniklých závad [33][38].

## 5.8 Uživatelské rozhraní

Poslední částí v kapitole technologií, která je nejdůležitější z pohledu klienta je uživatelské rozhraní sloužící jako prezentační vrstva celého projektu. Je to špička ledovce na práci, která je vykonaná na backendu. Rozsah funkcí, které lze na frontendu využít je definován vystaveným rozhraním na API Gateway. To poskytuje pomocí Swagger rozhraní, které lze konzumovat a mapuje metody z frontendu do backendu. Pro každý druh klientské aplikace s různým uživatelským rozhraním může být poskytnut jiný soubor funkcí podle principu BFF [48].

Klientské aplikace mohou být konstruovány na principu tenkého klienta, tedy webu nebo tlustého klienta, to znamená klasické aplikace například ve Windows forms.

Opět lze i na front endu volit mezi monolitickou nebo servisně orientovanou architekturou podporující modularitu.

Prvním způsobem je tedy použití klasického návrhu frontendu monolitickým způsobem. Například využít platformy .NET Core Web applications a pomocí návrhového vzoru MVC rozdělit celek na část pro model vztázený k datům, pohled jako prezentační vrstvu a kontroler pro část logiky. Na základě získaného popisu vystavenému na Gateway pak z vytvořených kontrolerů volat jednotlivé metody a se získanými daty nakládat vlastní způsobem na úrovni prezentační vrstvy [19].

Při správném návrhu a dodržení postupů pro BFF je možné po získání konfiguračních souborů a pomocí nástroje NSwag vygenerovat potřebné třídy pro vytvoření Http klienta a obsahujícího i modely a parametry, které se mají předávat prostřednictvím metod. Doménový přístup je tedy dodržen i na úrovni správy zdrojů klientské aplikace.

Vytvořené třídy pro přístup k metodám backendu pak lze využít v příslušných kontrolerech, které lze stále tvořit ve struktuře a organizaci doménového návrhu. Problém může nastat, pokud vznikne potřeba vytváření složitějších pohledů, které kombinují dvě a více domén.

Jednou z možností je tedy metody kombinovat, vytvořit tím vzájemnou vazbu a přijít o část autonomie jednotlivých domén a tím soustředit i případný vznik chyby. Na úrovni front endu jsou tímto snižovány výhody izolace chybosti a je na této úrovni vytvářen do určité míry distribuovaný monolit. V případě výpadku jednoho servisu budou nedostupné i další, které s ním spolupracují na konkrétním kontroleru a metodě, přestože jsou stále na backendu funkční.

Možnosti, jak reagovat je provedení opětovné analýzy potřeb zadavatele projektu, přejít opět k návrhu doménového rozdělení, zvolit jednu z těchto kombinovaných domén a vytvořit další servisu která bude poskytovat agregovaný pohled z požadovaných servisů. Případně je možné servisy opět spojit a poskytovat je jako jeden celek v rámci domény.

V případě dobrého doménového návrhu a vhodné členitosti servis je možné této komplikaci zabránit, případně omezit na minimální možnou míru a jen

na nejnutnější prvky, případně zapojit další kontrolní mechanizmy a chybovost tak izolovat jiným způsobem. Například při využití technologie tvorby stránek s rozhraním Razor View Engine rozdělit jednotlivé prvky do komponentů, které mají vlastní kontroler ve kterém je možné zachytávat a reagovat na případné výjimky [49].

Rozdělení do komponent je pak ideální situace pro tvorbu kompozitního uživatelského rozhraní složeného ze samostatných miniatr aplikací a při technikách tvorby aplikací jedné stránky (single page application, SPA) ve webovém prostředí [48]**Chyba! Nenalezen zdroj odkazů.**[16][97].

SPA jsou tedy aplikace, které se jeví uživateli jako jedna stránka, která se jako při klasickém způsobu zpracování neobnovuje, ale pouze dynamicky mění své části podle požadavku. Díky tomu je možné zvýšit přehlednost a pohodlí práce se stránkou a aplikovat další grafické prvky. Díky univerzálnosti technologií pro tvorbu těchto aplikací je možné tento způsob aplikovat na libovolný typ backendu a umožnit tak tedy multiplatformní použití v různých zařízeních **Chyba! Nenalezen zdroj odkazů.**[32]**Chyba! Nenalezen zdroj odkazů.**[16].

Hlavními aktéry při tvorbě SPA jsou javascriptové frameworky, z nichž je neznámější a aktuálně nepoužívanější Google **Angular** a Facebook **React** **Chyba! Nenalezen zdroj odkazů.**[42][19][21][97].

**Angular** je Javascriptový a později TypeScriptový framework pro tvorbu dynamických webových single page aplikací, za kterým aktuálně stojí firma Google. Technologie je do současnosti i pro svou populáritu stále rozvíjena a je už několik generací a verzí. Aktuální verzi je 9 z ledna 2020 [21][50].

Princip tvorby SPA pomocí Angularu spočívá ve využití jazyka JavaScript nebo TypeScript ve vyšších verzích jako kompletního nástroje při tvorbě všech komponent klientské aplikace, která je zpracována především na straně klienta bez nutnosti využití serverových prvků. Usnadňuje modularizaci stránek a tím může stavět na výhodách izolace chybovosti a samostatnosti poskytované z mikroservisního backendu [16][21][60].

V první verzi AngularJs byla technologie založena na obohacení HTML kódu stránky speciálními atributy, které byli říkali javascriptovému frameworku Angularu, jaké funkce má na daný html tag namapovat a jak s ním zacházet [60][71].

AngularJS je také rozdělen do kontrolerů a specializovaných directiv, které jsou obdobou výše zmíněných komponentů a mají své vlastní rozhraní jako malá aplikace. Díky atributům a direktivám pak lze opakovaně na stránce vytvářet malé samostatně fungující funkční celky. AngularJs pracuje s interaktivním dvoucestným nebo jednocestným bindováním prvků na atributy modelu právě přidáním speciálního atributu do standardní editační komponenty poskytované v html. V případě potřeby samostatného zobrazení pomocí speciálních značek **Chyba! Nenalezen zdroj odkazů.**[60].

Dalším používaným javascriptovým frameworkem je React. Od AngularuJS a dalších obdobných frameworku se od svého počátku liší v tom, že při otevření stránky nepotřebuje přečíst, zpracovat a obohatit celý HTML DOM stránky, ale rovnou vytváří své malé vlastní komponenty s přidělenou funkcionalitou, do kterých vkládá výsledek se vším potřebným k činnosti. Přístup je obdobný jako zpracování způsobem Hypermedia driven.

Pomocí **React** vytvoříme například komponentu <CommentBox>, která se ve stránce zobrazí jako text-area s vlastní předdefinovanou validací, modelem a automatickým potvrzením formuláře po stisku klávesy. Tak lze výrazně zjednodušit, zpřehlednit a zrychlit zápis kódu Pomocí modulu Node.js lze také React generovat přímo na serveru nebo pomocí ReactNative vytvářet aplikace pro mobilní zařízení [73][88].

Zmíněné technologie .Net Core web applications, AngularJs nebo React mají své odlišnosti, výhody a nevýhody. Pro to, jaký z nich zvolit závisí čistě na požadavcích zadavatele [21][50]**Chyba! Nenalezen zdroj odkazů..**

Platí obdobné pravidlo jako u volby architektury. Pokud je požadavkem rychlé úvodní vytvoření webové aplikace nebo využití ve firemním prostředí, je nejrychlejším způsobem technologie .NET Core Web Applications. Tvorba SPA v AngularJS a React vyžaduje především v úvodní fázi více času a klade větší nároky na týmovou spolupráci a postavení dobrého základu. V pokročilejších

částech projektu pak ale plynou výhody z udržení distribuovaného systému. Rozhodujícím faktorem je kromě času a komplexnosti návrhu také velikost. Pokud je potřeba vytvořit rozsáhlou firemní aplikaci, je prvním krokem opět vhodné využít .NET Core a později jednotlivé části na jeho základě modularizovat pomocí technik SPA.

React je například využíván velkými společnostmi jako je Facebook nebo Instagram. Angular je vhodnějším pro aplikace menšího rozsahu pro svou v porovnání větší jednoduchostí k pochopení, přehledností [21][71][88].

V rámci firemního prostředí pak můžou stránky na principu SPA nahrazovat robustnější a hůře udržovatelné tlusté klienty a pro uživatele poskytovat přehledné a nematoucí uživatelské rozhraní **Chyba! Nenalezen zdroj odkazů.**[16].

## 6 Porovnání

### 6.1 SOA vs. MSA

**Granularita** je hlavním rozdílem mezi SOA a MSA. V MSA jsou komponenty mnohem drobněji strukturované a slouží pro jeden účel a správu čistě jednoho ohraničeného zdroje nebo oblasti. V SOA je jedna komponenta zaměřena na poskytování různorodých funkcí nad různými entitami. Svými funkcemi může reprezentovat velkou část a o rozměrech větší aplikace ve formě subsystému [75][63][98][3].

**Sdílení komponent** je v případě SOA základní a podporovanou technikou. Tam kde jsou v případě SOA vytvářeny vzájemné vazby, tam se snaží MSA naopak vazby minimalizovat a zachovat si svou samostatnost. V případě SOA tak je v případě změny procesu nutné analýza vazeb a úprava všech závislých funkcionalit v kódu. V případě MSA lze díky principu samostatnosti vytvořit novou komponentu nebo funkcionalitu a pro potřeby zpětného využití jinými komponenty jsou původní funkce zachovány a vystaveny tak jak jsou [75][63][98][72][34].

**Komunikační rozhraní** je využito u obou architektur využito různým způsobem. V SOA je využito v komunikaci API pro vzájemné volání metod a výměna zpráv je řízena ESB, které obstarává správu nad toky vyměňovaných zpráv. V MSA využívají servisy ve vzájemné komunikaci minimálně pouze v opodstatněných případech. Vzájemnou komunikaci zajišťují publikováním informace ve formě události o akci, kterou vykonaly a v případě zájmu si na ní mohou další servisy komunikovat samostatně [75][63][98][72].

**Protokoly** jsou využívány v obou architekturách jako pravidla a určení způsobu komunikace. V SOA jsou využívány obecné i komplexnější komunikační protokoly. Pro zpracování zpráv využívá AMQP, MSMQ a pro komunikaci standardizovaný protokol SOAP. V MSA je především využíván obecný komunikační standard HTTP REST, který je univerzálním pro drtivou většinu všech technologických platform. Pro zasílání zpráv využívá také protokolů MSMQ nebo JMS [75][98][72].

**Podpora různorodých technologií** je vlastnost SOA i MSA. V SOA je tato podpora zajištěna skrze protokoly zasílání zpráv. V MSA jsou velké možnosti pro zapojení různorodých technologií díky univerzálnímu komunikačnímu rozhraní. To oproti SOA usnadňuje volbu kombinace technologií a podporující různé protokoly. Pokud je tedy systém vytvářen a konstruován podle předem vybraných několika protokolů lze využít vlastnosti SOA. Pokud je volba jednoho univerzálnějšího protokolu s volnou možností vazby, je lepší využití MSA [75][98][72].

**Velikost systému** a jeho účel je důležitým kritériem pro zvolení architektury. U velmi rozsáhlých systému je u systému stále SOA používána. MSA je naopak oproti SOA vhodnější využít u aplikací menšího rozsahu díky možnosti drobně dělit jednotlivé prvky, dobré podpoře webového rozhraní a univerzálnosti při využití stávajících schopností vývojářů. Systémy v MSA lze dále rozšiřovat a při nárůstu velikosti systému je možnost je dále rozvíjet podle potřeby. Záleží na způsobu a záměru poskytovaných služeb a jakým způsobem je jejich poskytování distribuováno. Například Netflix a jeho platforma je vhodnou ukázkou fungování mikroservisního přístupu v budování velmi rozsáhlých systémů [3][75][98][72][34].

Porovnání výhod a nevýhod u obou architektur bylo provedeno v kapitole 4.11 (Obrázek 12 Struktura MA, SOA a MSA, Zdroj: vlastní tvorba). V této tabulce jsou shrnutы hlavní charakteristiky na základě provedeného porovnání [75][98][72][34].

SOA	MSA
Záměrné <b>vytváření vzájemných vazeb</b> a sdílení zdrojů	<b>Minimalizace vzájemných vazeb</b> a sdílení zdrojů
<b>Vysoký důraz na opakovatelnost</b> komponent a funkcionalit. Jejich neohraničené užití v celém systému.	<b>Vysoký důraz na rozdělení a samostatnost.</b> Ohraničená odpovědnost za zdroj a oblast
<b>Společné řízení</b> a standardizace	Zaměřeno na <b>volnost a kreativitu</b> jednotlivých částí s možností spolupráce
<b>Komunikace</b> řízena pomocí centrálních prvků a ESB	<b>Komunikace</b> prostřednictvím nezávislé publikace a konzumace zpráv
Podpora <b>více specializovaných protokolů</b>	<b>Univerzální obecný protokol</b> pro komunikaci
<b>Souběžné více vláknové zpracování</b> vstupů a výstupů	Jednocestné <b>postupné zpracování zprávy</b> jako vstupního a výstupního zdroje
Snaha o <b>opakované využívání vazeb</b>	Zaměřeno na <b>zachování volných vazeb</b> a samostatnosti
Užití tradičních <b>relačních databázových struktur</b> a práce s daty	Užití <b>více typů databází</b> a způsobů práce s daty. <b>Udržování událostní konzistence</b>
Změny velkých komponentů znamenají <b>úpravu monolitického prvku</b>	<b>Vytváření nových prvků</b> a preference přidání nové funkcionality před úpravou staré
Řízení provozu a udržitelnost vývoje je trendem, ale není prioritním směrem.	Velký důraz na řízení provozu a nepřetržitého rozvoje

Tabulka 2 Porovnání SOA a MSA

## 6.2 MSA vs. Monolit

Po porovnání servisně zaměřených architektur, které fungují na obdobném principu a jen s odlišnou myšlenkou a přístupem je dále možné porovnat mikroservisní s monolitickou architekturou. Tady už je hlavní rozdíl v zásadních oblastech.

Hlavní rozdílem je, že nemá smysl porovnávat granularitu systému, sdílení komponent, vnitřní komunikační rozhraní nebo podporu různých technologií [63][15].

Takové porovnání by bylo možné v případě systému, který by byl vytvořen jako distribuovaný monolit. Tedy aplikace, která je původně tvořena buď jako SOA nebo MSA, ale má tak vysokou úroveň svázanosti a vazeb, že nejde komponenty samostatně a ani opakovaně využít [24][15].

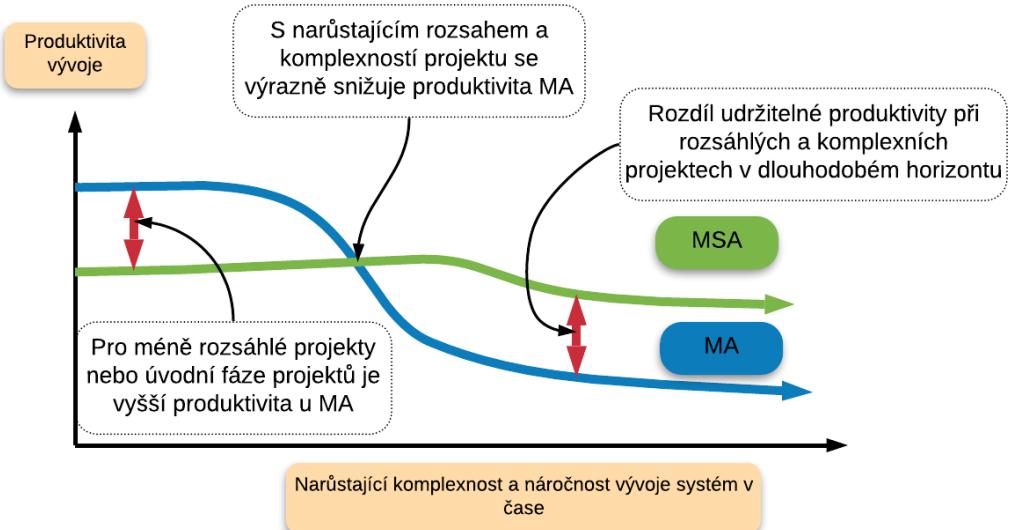
Systém vytvořený v MA je jako jeden celek postaven na jedné technologii a vnitřní komunikace je prostřednictvím přímého volání metod bez nutnosti využití komunikačních prostředků. Srovnání lze provést v oblasti vývoje, údržby, spolehlivosti, škálovatelnosti a ceny ve smyslu využití zdrojů a času[7].

V oblasti vývoje systému je monolitický systém vytvářen v jedné technologii a jako jeden celek. Jeho návrh není tak komplexní, a proto je v počátcích mnohem efektivnější a rychlejší vývoj než v případě MSA. V MA lze v rámci jednoho systému rozdělit funkcionalitu vnitřně do modulů, které můžou také do určité míry kopírovat rozdělení podle doménového návrhu s tím rozdílem, že používají vysokou míru vzájemných vazeb a sdílené datové zdroje [7][24][63].

Jsou doporučení, že vývoj systému v MSA by měl využít této vlastnosti a na začátku vývoje postupovat jako vývoj MA. Po vytvoření základu pak monolit rozdělit na jednotlivé služby a pokračovat dále čistě mikroservisní cestou. Tento postup ale narází na následnou potřebu zpětné analýzy před rozdělením, která získaný čas opět ztratit. Nelze odhadnout správný moment, kdy rozdělení provést. Systém je sice rozdělen do modulů, ale ty nelze kvůli vzájemným vazbám tak snadno rozdělit [7][24].

Jaká z architektur je vhodnější pro plánovaný projekt, závisí na několika faktorech. Hlavním faktorem je její plánovaný rozsah a doba využití. Vývoj systému v MSA obnáší především na začátku vývoje komplikace díky nutnosti provést komplexní návrh, vybudování infrastruktury a nastavení organizace činností. Pro správnou funkci systému je nutné implementovat způsoby pro nasazení aplikace do produkce, nastavení monitoringu, reakce na selhání, vytvoření a řízení událostní konzistence a mechanismů obnovy systému. Zároveň může být pro menší projekty zbytečně nákladné pořízení specializovaných nástrojů. Především tedy na začátku je přístup MA mnohem produktivnější. Pokud se ale plánuje další její rozvoj, tak s narůstající složitostí a rozsahem systému je údržba monolitického systému stále náročnější a zpomaluje celkový vývoj [63][24][15].

Tento vývoje je naznačen na Obrázek 23 Komplexita vývoje MSA a MA Zdroj: [24]



Obrázek 23 Komplexita vývoje MSA a MA Zdroj: [24], vlastní úprava

**Provoz** není v případě MA komplikovaný. Po provedených změnách je nutnou celou aplikaci sestavit a připravit soubory pro nasazení do produkčního prostředí. Tam je vyměněna celá provozovaná verze za novou a provedeno testování. V závislosti na velikosti souborů a aplikace to obnáší větší datový přenos, plánování změn a upgradu aplikace a nutnost dočasně odstávky mezi přepínáním na vyšší verzi. Oproti tomu v MSA je možné nasazení automatizovat a efektivně vyměnit pouze ty části, které byly změněny. U nasazovaných servisů se vytvoří produkční image, který se dá k dispozici virtualizačnímu nástroji, který provede validaci a testování a v momentě nízké zátěže může servis spustit. Díky verzování a zpětné kompatibilitě je zajištěna funkce všech běžících komponentů [7][24].

V rámci provozu je také rozdílem způsob škálovatelnosti, tedy reakce na potřeby zvýšeného nároku na výkon systému. V MA je nutné navýšovat výkon vertikálně, tedy přidávat výkon na provozovaném zařízení. Tento způsob má své technické a finanční limity. V případě horizontálního škálování je potřeba replikovat vždy celou aplikaci a tím se také navýšují výrazně náklady. V případě MSA je možné navýšovat výkon jak vertikálně na hostitelském zařízení, tak horizontálně. Velký rozdíl je, že díky virtualizaci je možné provést navýšení jen u té ohraničené části systému, u které je to nezbytně nutné a tím výrazně ušetřit zdroje [7].

**Údržba** monolitické je vázána publikování aplikace jako celku. Nelze okamžitě reagovat a provádět časté úpravy. Zároveň v závislosti na rozsahu a velikosti aplikace nebo v případě výměny vývojářského týmu se stává údržba aplikace časově náročnou a komplikovanou. Je potřeba dostudovat strukturu z vytvořené dokumentace nebo aplikaci zpětně analyzovat a pochopit pro případné zhodnocení dopadu prováděných změn. V případě špatné analýzy, může dojít k narušení funkčnosti a prodlužuje se tak čas a náročnost vývoje. Oproti MSA je ale možné aplikaci snáze a rychleji testovat a případné chyby identifikovat [7].

**Chybovost a její izolace** je faktor ve prospěch MSA. V MSA je systém distribuovaný a obsahuje mnoho pohyblivých částí, které lze hůře monitorovat než v případě MA. Od zahájení vývoje je projekt v MSA koncipován s ohledem na celkovou odolnost, samostatnost a izolaci chybovosti a v případě dobře navrženého testování a monitoringu je odhalena včas většina závad včas. U systému v MA pak neodhalení chyby nebo slabého místa systému způsobit díky závadu celého systému a odepření všech poskytovaných služeb [7][15].

Při volbě vhodné architektury pro projekt je nutné zvážit několik bodů.

**MA** je vhodné zvolit v případě malého vývojového týmu (2-4 osoby), malého rozsahu funkcionalit v řádu jednotek, omezeného rozpočtu času nebo v případě, že je zaručené, že vytvořená aplikace se nebude dále rozvíjet [24].

**MSA** lze uvažovat v momentě, kdy na začátku projektu není časová tíseň pro rychlé vyprodukování výsledku a je dostatek prostoru na komplexnější návrh a přípravu infrastruktury. Dále pokud je k dispozici dostatek personálních zdrojů na vývoj systému nebo jsou týmy rozděleny do několika různých oblastí ať logicky nebo fyzicky na jiných lokalitách. Zvolit MSA je vhodné, pokud je během úvodní analýzy předpoklad problémů s poskytováním dostatečného výkonu a pokud bude vyžadováno jeho řízení nebo v případě kdy se očekává další a častý vývoj nového systému. Případně lze uvažovat o projektu v MSA jako alternativně za stávající systém v MA, který má právě hlavní problémy s řízením výkonu nebo náročnou udržitelností rozvoje. V takovém případě je jednou možností vytvoření nové

aplikace nebo postupné rozdělení již provozované aplikace do jednotlivých servisů [7][24].

### 6.3 MSA a metodiky

Pro vývoj systému v MSA je vzhledem k jejím charakteristikám nevhodnější volba agilních metodik. Agilní metodiky využívají vlastnosti MSA jako je rychlá reakce na změny potřeb, adaptace vývojového procesu, časté dodávky funkčního kódu a možnost automatizace testování a provozu. Největší předností MSA pro agilní metodiky je především udržení volných vazeb a rozdělení prvků podle doménového návrhu na malé části. To umožňuje jasné vymezení odpovědností a více možností, jak tvořit a organizovat vývojové týmy, vydávat pravidelně funkční části a dosahovat rychle efektivních výsledků. Využitích klasických metodik jako například vodopádový model není vhodné z několika důvodů. V prvním případě je dlouhá prodleva mezi jednotlivými produkčními cykly, pomalá reakce na změnu a velké riziko tvorby pevných vazeb mezi komponentami, na jejich odstranění je potřeba další úsilí a ztráta času [61][57][67].

## 7 Aplikace v MSA

Projekt aplikace Kancelář má za účel využít principů mikroservisní architektury a zapojení zmíněných technologií z teoretické části práce. Vývoj aplikace je zasazen do prostředí imaginární organizace, která má svůj vlastní vývojový tým a zájem vybudovat svůj vlastní systém řízení a plánování personálních zdrojů a s tím spojených procesů a aktivit. Hlavní zaměření je tedy na podporu podnikových cílů z perspektivy rozvoje a podpory interních procesů.

V rámci organizace bude aplikace využívána zaměstnanci na všech úrovních organizace s plánovaným počtem 2000 uživatelských účtu, rozdělených do organizačních jednotek a využívající různých rolí z pohledu přidělených oprávnění. Záměrem je využití stávající databáze organizační struktury, napojení na infrastrukturu pro kontrolu vstupů a pohybu po objektech a poskytování strukturovaných podkladů pro existující systém pro výpočet mzdy. Aplikace by dále měla podporovat řízení a správu pracovních a vzdělávajících aktivit.

### 7.1 Role

Pro realizaci projektu podle metodiky SCRUM bude vytvořeno několik vývojových týmů, které budou slouženy z 5 až 6 členů. Součástí týmu bude vždy osoba z odborné oblasti domény v roli **Product Owner**, vedoucí vývojového týmu v roli **Scrum Master** a členové týmu v roli **Developer**. Pro každou doménu a fázi sprintu určí Product Owner podle svého uvážení zaměstnance z produkční oblasti do role **Public** za účelem otestování dema a vytvoření zpětné vazby.

Před zahájením projektu proběhne v každé oblasti analýza hlavních cílů s obecným popisem uvedeným v produkční backlogu, prioritizace úkolů, odhad náročnosti a bodové ohodnocení. Po provedení bodového ohodnocení bude v každé oblasti na rozhodnutí Scrum Mastera a hlavního vedoucího projektu, které cíle se budou realizovat. Z důvodu snahy o co nejkratší zahajovací fázi budou nejdříve realizovány

projekty s vysokou prioritou a rychlým odhadem realizace. Obecné složení týmu a popis rolí v tabulce č.

Struktura týmů		
Role	Osoby	Činnost
Product owner	1 - 2	Vlastník domény, definice procesů, struktury entit a požadavků na funkcionality, konzultace k odbornému zaměření domény
Scrum master	1	Analýza požadavků, konzultace k definovaným cílům, návrh technického zpracování, Spolupráce na kompozici podřízených prvků s dalšími Scrum mastery
Developer	2	Realizace zadání
Public	1	Vytváření zpětné vazby a zpětná vazba po předvedení dema

Tabulka 3 Struktura Scrum týmu

## 7.2 Doménově řízený návrh

Po analýze úvodního záměru projektu a před konkrétní specifikací cílů a vytvoření produkčního backlogu se provede doménově orientovaný návrh a rozdelení jednotlivých částí projektu na domény a subdomény. Pro vytvořené domény a subdomény určené podle odborného zaměření pak budou určení odborní garanti v roli Product Ownera, vytvořeny jednotlivé vývojové týmy a ty provedou úvodní analýzu konkrétních cílů a funkcionality za účelem tvorba Produkčního backlogu.

**Finanční doména** je zaměřena na sběr, přípravu a zpracování dat jako podkladu pro výpočet mzdy zaměstnanců. Pro zpracování návrh domény, subdomén a zahrnutých cílů jsou určeny týmy obsahující osobu vlastníka procesu z oblasti finančních oddělení v organizaci.

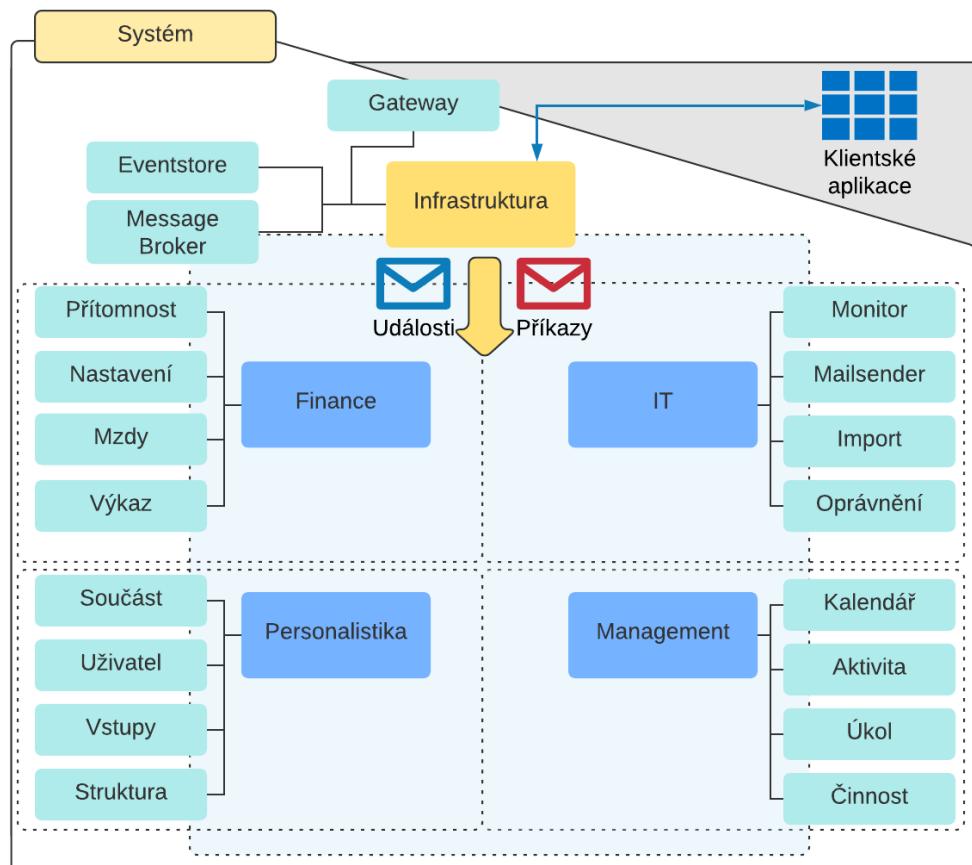
**Personální doména** se zabývá organizováním přiřazením osob z personální databáze organizace k uživatelským účtům v rámci aplikace, jejich organizování do struktur stejných jako v organizaci, případně do specializovaných týmů podle potřeby vedoucích nebo pro potřeby činnosti na úkolech. Tím je umožněno organizovat strukturu cíleně pro zpracování aktuálního úkolu nebo podle aktuálního stavu procesů uvnitř organizační součásti.

**Informační technologie a Infrastruktura** jsou domény zajišťující provoz celého systému. Tato doména je zaměřena na návrh a realizaci jednotlivých komponent

a způsobu realizace funkcionalit zajišťující ostatní cíle. Je plně ve správě vývojového týmu. V realizaci cílů je to prioritní oblast cílů. Návrh realizace je koordinován s časovými a finančními zdroji vyčleněnými pro projekt. Možnosti rozsahu a využití jsou závislé na rozhodnutí hlavního zadavatele projektu v rámci organizace.

Garantem domény je vedoucí pracovník za oblast informatiky za celou organizaci. Jako konzultační odborná pomoc je určen pověřená osoba z oblasti bezpečnosti a ve fázi návrhu osoby odpovědné za provoz informačního systému organizace a infrastruktury produkčního prostředí.

**Management** je doménou sloužící pro poskytování přehledu řídícím pracovníkům o personálních zdrojích a poskytuje možnost řízení aktivit. Prostřednictvím funkcionalit v této doméně mohou úkoly vytvářet, směřovat aktivity k jednotlivým zaměstnancům, monitorovat a vyhodnocovat jejich plnění. Garantem domény je pověřená osoba z oblasti managementu a personalistiky. Je nutné zakomponovat pravidla a možnosti, které jsou zaměstnancům a managementu k dispozici.



Obrázek 24 Doménově řízený návrh Zdroj: vlastní tvorba

## 7.3 Scrum

### 7.3.1 Produktový backlog

Součástí úvodní fáze projektu, je kromě vytvoření doménově orientovaného návrhu, vypracovat Produkční backlog, obsahující podrobnější specifikace cílů než úvodní záměr.

Při vytváření backlogu už jsou vyjasněné odborné hranice a podle rozdelených subdomén se dají určit jednotlivé funkcionality podpory podnikových procesů. Účelem je vyjasnit priority a význam jednotlivých funkcionalit a získat metriky, podle kterých se bude při tvorbě systému postupovat. U každého cíle je nutné přiřadit prioritu z pohledu dopadu a důležitosti konkrétní funkcionality na funkčnost celku a podporu podnikových procesů. Dále je odhadnuta náročnost konkrétního řešení a proveden výpočet bodového ohodnocení.

Priorita cílů		
Hodnota	Název	Popis
5	Kritická	Kritický význam pro funkctionalitu celého systému.
4	Prioritní	Nutné pro zajištění minimálních požadavků na plnění procesu
3	Potřebná	Nutné pro zajištění zamýšlených požadavků na podporu procesu
2	Vhodná	Nadstandardní funkce pro ideální plnění podpory procesů
1	Doplňková	Doplňková funkctionalita

Tabulka 4 Scrum - priorita cílů

Po určení priorit je určena relativní náročnost každého cíle. Nejsložitější úkol dostane nejvyšší bodové ohodnocení, a naopak ten nejsnadnější to nejnižší. Všechny ostatní cíle určují svou relativní náročnost v rozmezí těchto hranic.

Relativní náročnost		
Typ	Hodnota	Příklad cíle
MAX	10	Testování
AVG	5	Návrh Eventů
MIN	1	Popis rozhraní

Tabulka 5 Scrum - Škála relativní náročnosti

Po provedení prioritizace a ohodnocení náročnosti je vypočtené bodové ohodnocení, které je součin hodnoty priority a náročnosti.

Po získání celkového ohodnocení všech cílů, lze odhadnout bodové velikosti pro každý sprint a rozlišit rychlosť zpracovania cíle do tří kategorií. Pokud je vyplňené referenční ID, je aktuální cíl závislý na realizaci jiného cíle.

Rychlosť dosaženia užitku z cíle	
Hodnota	Název
4	Rychle získaný užitek (quick win)
0	Vyrovnáný užitek (balanced win)
-5	Pomalu získaný užitek (longterm win)

Tabuľka 6 Scrum - rychlosť dosaženia cíľu

V tabuľke č. je príklad záznamu v backlogu s obsahujúcimi vlastnými identifikátormi, názvom, prioritou, náročnosťou, bodovým ohodnocením, odhadom rychlosťou splnenia a referenčnou ID. Kompletná tabuľka je súčasťou prílohy A.

Oblast	ID	P	Název	Náročnost	Body	Rychlosť	Ref ID
Pers.	1	3	Schválení ÚPD osoby	3	9	0	8

Tabuľka 7 Záznam produkčného backlogu

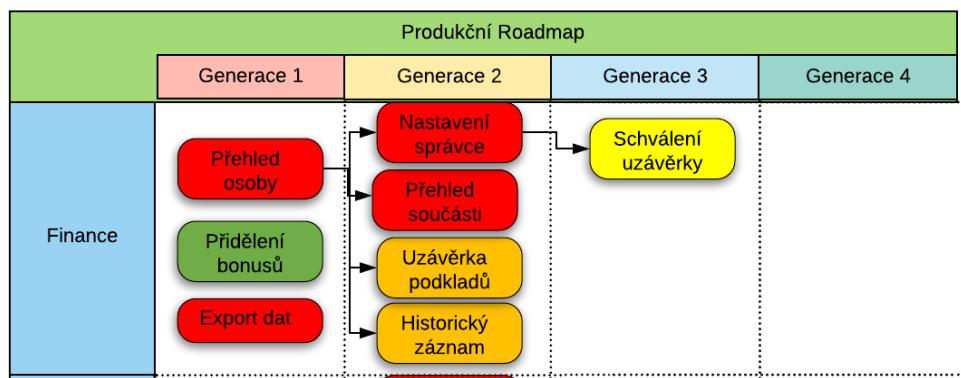
Současťí záznamu v backlogu je také popisné Story, ktoré je vypracované vlastníkom dané oblasti a cíle a ktoré obecně popisuje zamýšľenou funkcionality. Záznam obsahuje, kdo bude užívateľom, co má bude cílem a za jakým účelom tuto funkciu požaduje. Príklad Story (Tabuľka 8 Backlog Story) a kompletný prehľad je súčasťou Příloha A SCRUM.

Oblast	ID	P	Název	Ref	Kdo	Co	Proč
Pers.	1	3	Schválení ÚPD osoby	8	Ved.	Schválení <b>dočasné úpravy pracovnej doby</b> zaměstnance oproti standardní pracovní době. Dočasná úprava pro případ služební cesty nebo jiných mimořádných událostí.	Přehled o nestandardní pracovní době. Rozhodnutí o udělení nebo neudělení výjimky.

Tabuľka 8 Backlog Story

Vyhodnocení hodnot z produkčného backlogu obsahuje Příloha A SCRUM – Produkční Backlog. Na základe vyhodnocení lze odhadnout dĺžku a náročnosť sprintov, do ktorých sa pak rozdelí skupiny uživatelských story pre zpracovanie. Odhadovaný plán sprintov je opäť v príloze A tabuľce č.4

Závěrem přípravné faze je vypracována odhadovaná cesta vývoje jednotlivých cílů v rámci celého projektu tzv. Produkční Roadmap. Znázorňuje logickou návaznost jednotlivých cílů a rozděluje process vývoje do generací. Úplný Road Map je součástí přílohy A Scrum Obrázek 42 Produkční Road Map.



Obrázek 25 Produkční Roadmap – ukázka Zdroj: vlastní tvorba

### 7.3.2 Sprint

Po úvodní fázi projektu, kdy byla vytvořena struktura týmů, doménový návrh a na jeho základě zkonstruovaný produktový backlog obsahující popis jednotlivých cílů je vytvořen plán prvních sprintů (Příloha B SCRUM - sprinty).

Základní požadavky na jeden cyklus sprintu jsou, aby splnil minimální počet stanovených bodů a realizoval cíle podle priority a rychlosti jejich dosažení.

V každém dvoutýdenním sprintu se opakují fáze plánování, kontroly stavu, projekce dema pro získání zpětné vazby a vyhodnocení činnosti týmu.

**Plánování** probíhá jako první fáze cyklu. Na společném jednání se sejde projektový tým složený ze všech rolí určených podle domény, jsou vybrány jednotlivé cíle a zpracován Sprint backlog, kde se hlavní cíl rozpadá na jednotlivé detailněji pospané úkoly, které jsou realizovány.

**Kontrola stavu** se provádí průběžně na denní bázi v podobě konzultací v rámci projektového týmu. Díky tomu má práce na úkolu okamžitou zpětnou vazbu od vlastníka cíle a vývojový tým může rychle a efektivně provádět změny a další vývoj.

**Projekce dema** se provádí v konečné fázi sprintu, kdy je představena funkční část aplikace jako výsledek aktuálního sprintu. V této fázi se získá zpětná vazba od vlastníka cíle a od běžného uživatele v roli Public.

Na základě zpětné vazby je splnění cíle vyhodnoceno a v případě, že vyhovuje požadavkům, je zaznamenán do produktového backlogu jako splněný. Pokud jsou nutné úpravy a další zpracování, je požadovaným změnám přiřazena nová priorita, odhadnuta nová náročnost, rychlosť zpracování cíle a je zařazen do backlogu jako další cíl. Takto zařazený cíle je pak možné plánovat v dalších sprintech.

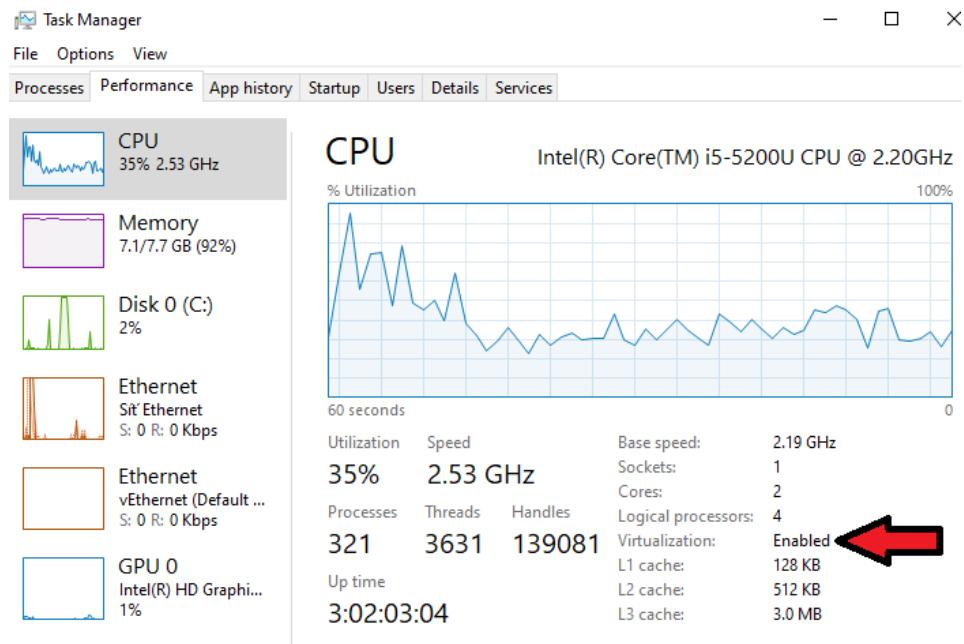
Poslední fází cyklu je **vyhodnocení činnosti projektového týmu** z pohledu dosažených cílů, efektivity práce a dalších možností zlepšení do dalšího sprintu.

## 7.4 Vývojové prostředí

Před zahájením vývoje aplikace je nutné nastavit vhodné vývojové prostředí podporující zmíněné technologie Docker a tvorbu jednotlivých servis. Pro potřeby tohoto projektu je vývoj realizován s využitím **Windows 10 Pro 64bit**, **Microsoft Visual Studio Community 2019** a **Docker Desktop**. Použité zařízení je laptop s operační pamětí 16 GB, procesorem Intel Core I7 9750H Coffe Lage a diskem SSD 256GB. Na zařízení bylo možné díky nástroji Docker virtualizovat 20 malých linuxových kontejnerů a samostatný kontejner pro MS SQL server a RabbitMQ.

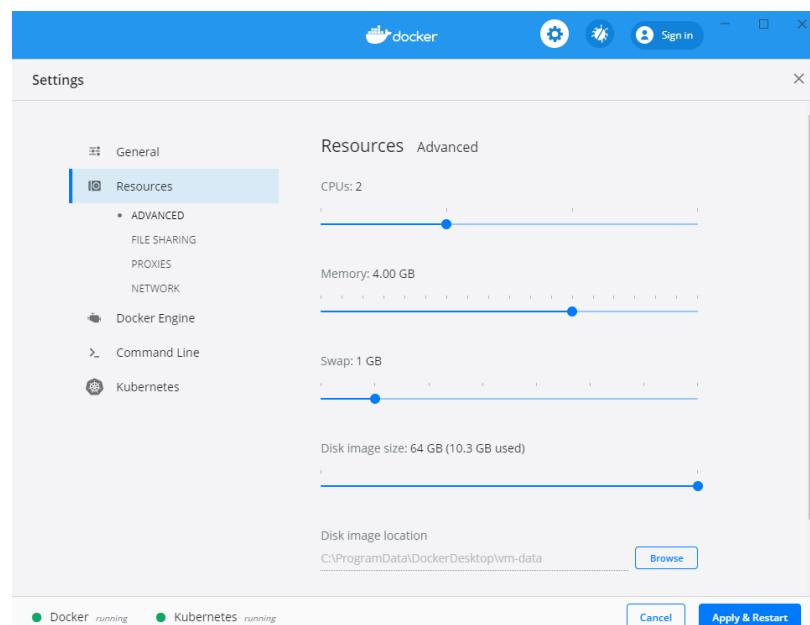
Druhým zařízením byl laptop Lenovo Thinkpad s operační pamětí 8GB, procesorem Intel core i5 5200U 2.20Ghz. Na tomto zařízení byl v případě spuštění všech mikroserverů problém s nedostatkem zdrojů. Pokud by například bylo využito virtuálních zařízení nástrojem VMware, bylo by spuštění vůbec možné.

**Instalace Docker desktop** vyžaduje verzi 64 bitovou verzi Windows Pro nebo Enterprise z důvodu využití funkce Client Hyper-V pro provedení virtualizace operačního systému v kontejnerech. Tuto funkci je nutné před instalací Dockeru povolit v nastavení Biosu. Kontrola funkce, zda aktivní, je možná v Task manager v sekci výkon a parametru virtualizace (Obrázek 26 Stav nastavení Hyper-V).



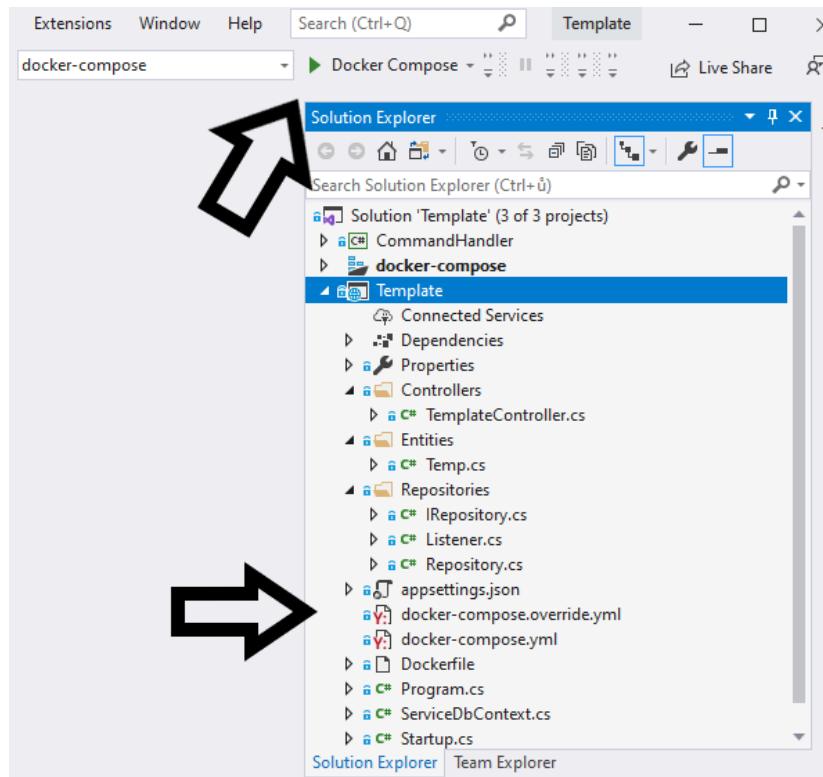
Obrázek 26 Stav nastavení Hyper-V Zdroj: vlastní tvorba

Po aktivaci a restartu systému je možné Docker Desktop nainstalovat. Po úspěšné instalaci je nutné nastavení výkonu a prostředků, které bude nástroj využívat. Pro využití integrovaného nástroje **Kubernetes** pro potřeby service discovery a orchestrace je nutné přepnout Docker do podpory **Linux kontejnerů**. Dále je pak nutné nastavit sdílení disků a následně nastavení využitelného rozsahu paměti a prostoru na disku.



Obrázek 27 Nastavení zdrojů Docker desktop Zdroj: vlastní tvorba

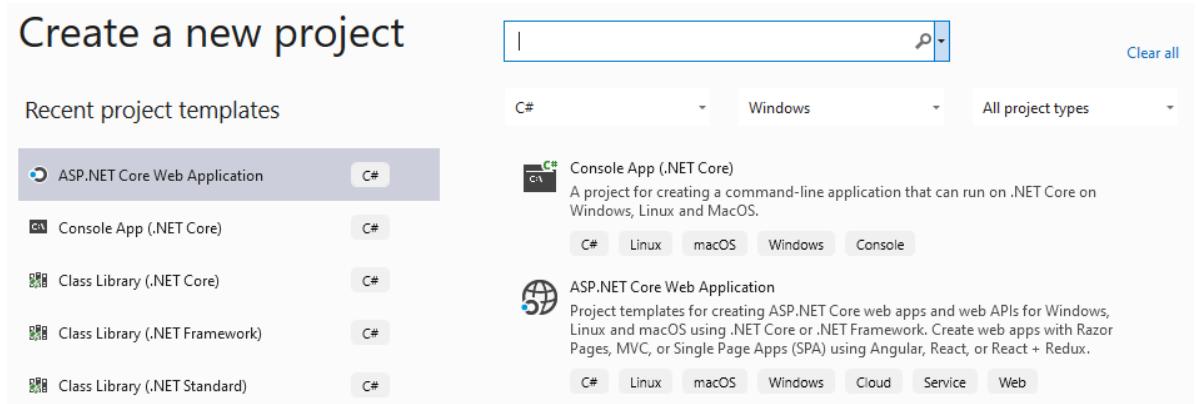
Pro vývoj servis na platformě .Net Core byl zvolen vývojový nástroj Visual studio 2019, který obsahuje podporu spolupráce s nástrojem Docker a je možné přímo z nich provádět spuštění kontejneru servisy nebo provedení kompozice více servisů projektu. Visual studio přímo podporuje vygenerování konfiguračního souboru Docker pro jednotlivý servis Docker-compose.yml více servisů. Spuštění kompozice lze pak provést přímo nebo s využitím příkazové řádky za pomoci příkazů docker buď opět ze studia nebo z řádky powershellu v adresáři, kde jsou konfigurační soubory umístěny.



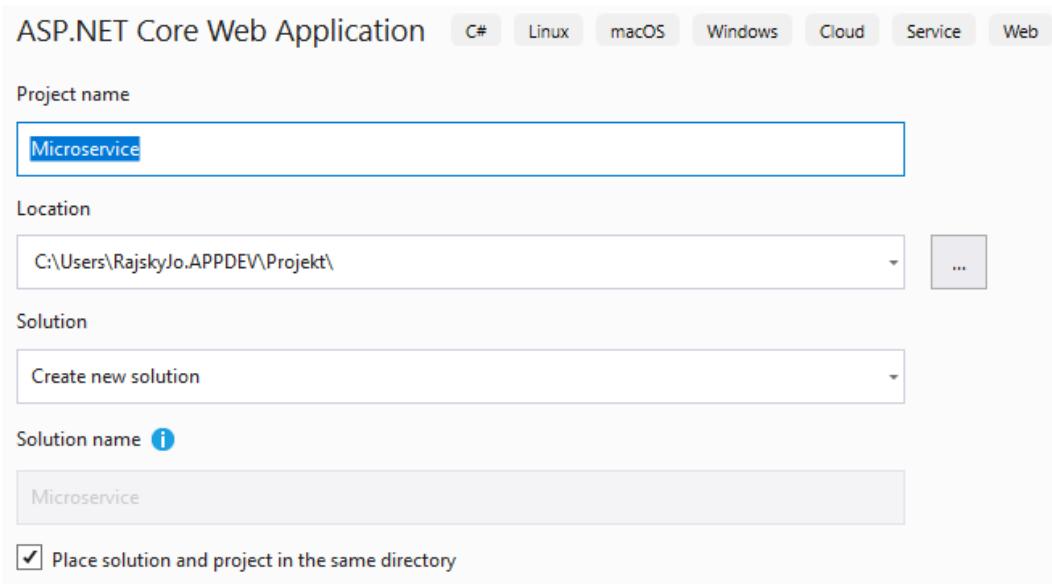
Obrázek 28 Visual Studio podpora Docker desktop Zdroj: vlastní tvorba

## 7.5 Založení mikroservisu

Každý servis je malý a samostatný projekt. Jeho vytvoření. Možnost je vybrat buď Console app pro tvorbu servis vykonávající služby samostatně nebo Web Aplikaci podporující rozhraní API (Obrázek 29 Výběr typu servisu). Dalším krokem je nastavení názvu servisu, adresáře a je vhodné pro potřeby docker-compose uložit celkovou Solution do stejného adresáře s projektem (Obrázek 30 Založení servisu).



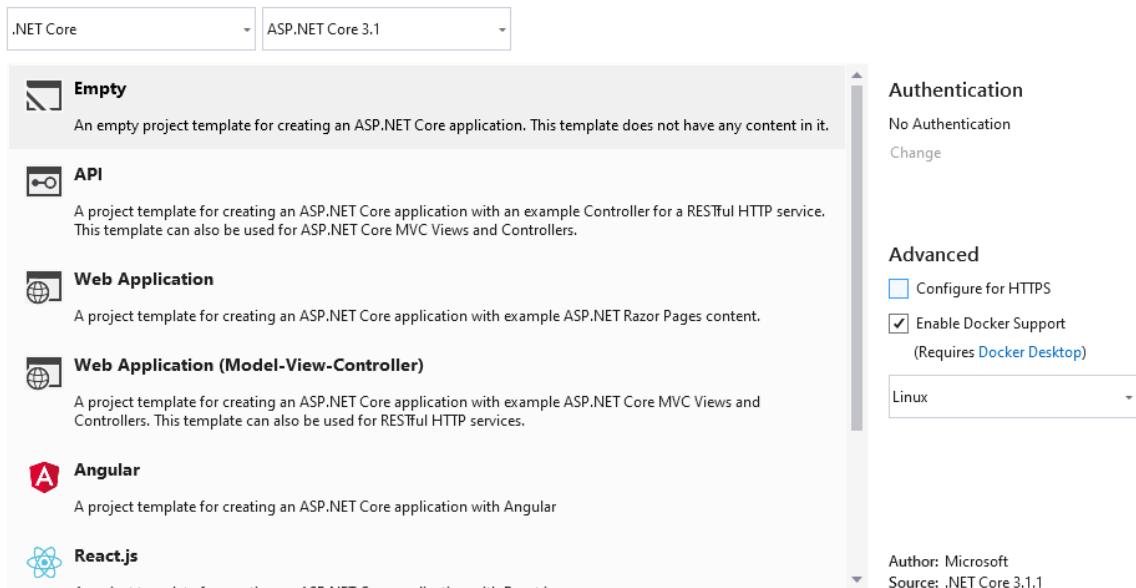
Obrázek 29 Výběr typu servisu Zdroj: vlastní tvorba



Obrázek 30 Založení servisu Zdroj: vlastní tvorba

Posledním krokem je vybrání nového prázdného projektu. V pravé části pak vybrat podporu kontejnerů Dockeru a vybrat typ. Na základě tohoto nastavení se vytvoří základní konfigurační soubor docker, který je součásti projektu.

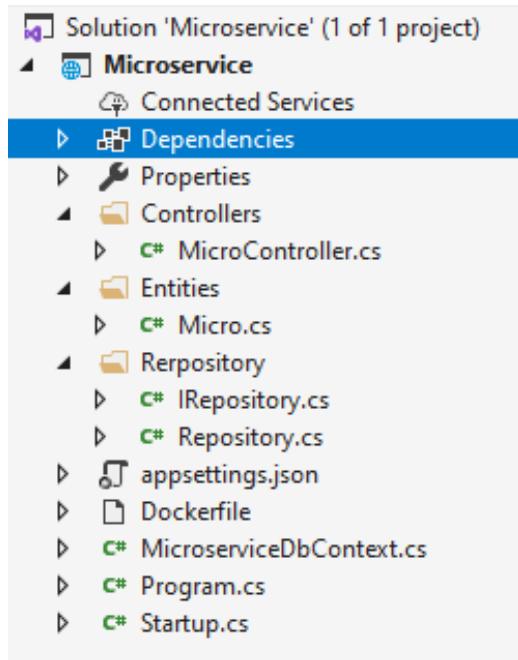
Závěrečným krokem je vybrání nového prázdného projektu. V pravé části je pak nutné vybrat podporu kontejnerů Docker a zvolit typ. V tomto projektu jsou všechny kontejnery tvořeny na platformě Linux. Na základě této definice a informací o projektu se založí v projektu předpis Dockerfile (Obrázek 31 Nastavení šablony a vlastností servisu).



Obrázek 31 Nastavení šablony a vlastností servisu Zdroj: vlastní tvorba

Do nově vytvořeného projektu Microservice se pak přidá třída pro Controller obsahující metody API a repositář obsahující metody pro práci s daty. Pro datovou vrstvu se vytvoří třída databázového kontextu a příslušná entita.

Do nově vytvořeného prázdného projektu se následně přidá adresář a třída pro Controller, Entitu, Repositář, Posluchač (Listener) a DbContext (Obrázek 32 Struktura servisu).

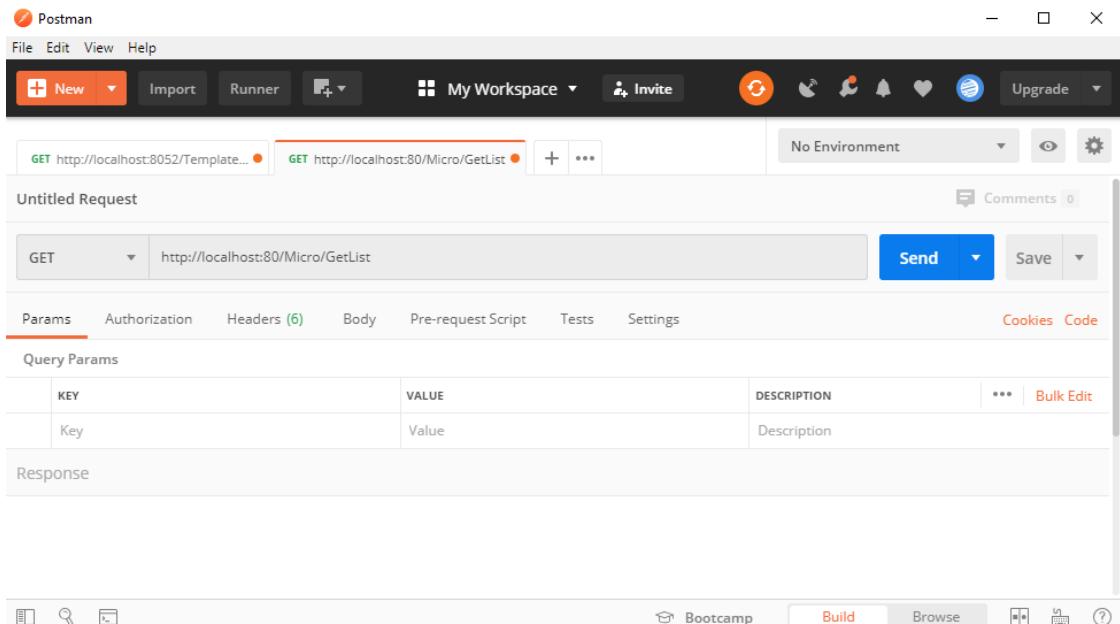


Obrázek 32 Struktura servisu Zdroj: vlastní tvorba

Třída Controller obsahuje metody vystavené jako API servisu, Entita popisuje zpracovávaný objekt, který je předáván v díky DbContextu do vytvořené databáze pomocí EntityFramework Core.Listener obsahuje metody reakce servisu na konzumované zprávy získané z Message Brokera, třídí je a předává dále pro zpracování do Repositáře.

Repositář obsahuje hlavní část logiky celého servisu. Zde jsou prováděny reakce a metody na příkazy získané z Controlleru a reakce na události z Listeneru. Výsledky jsou dále publikovány a ukládány do databáze díky připojenému DbContextu.

Každý servis lze spustit samostatně v kontejneru a lze ho i samostatně testovat. Možnostmi jsou využití aplikační konzole, loggeru na úrovni kontejneru nebo využití externích nástrojů pro práci s API jako je Postman nebo Swagger. V případě projektu byl využíván nástroj Postman (Obrázek 33 Uživatelské rozhraní Postman).



Obrázek 33 Uživatelské rozhraní Postman

Aby bylo možné servis testovat, je nejdříve nutné ho spustit. Po vytvoření a buildu projektu se nejdříve vytvoří na základě předpisu Dockerfile jeho Image, který je následně spuštěn v Docker.

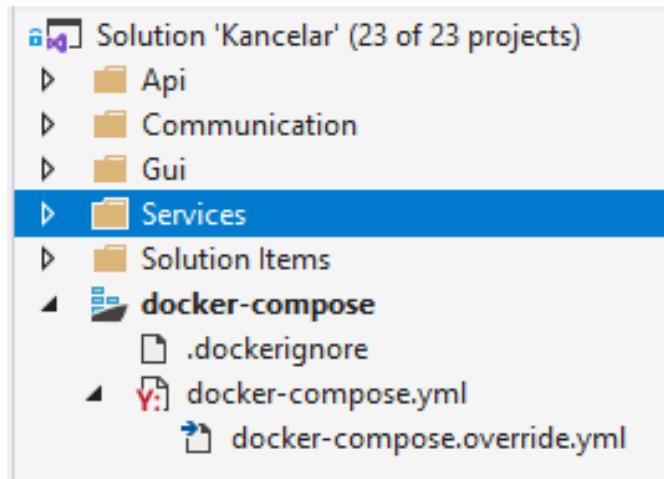
První možností je využít podpory prostředí Visual Studio a spustit celý proces stisknutím jednoho tlačítka (Obrázek 28 Visual Studio podpora Docker desktop). Mimo Visual Studio lze nástroj Docker ovládat pomocí sady příkazů, které lze zadávat buď do příkazové řádky i v rámci Visual studia, případně jiného editoru, nebo ze standardní příkazové řádky nebo Powershellu. Kompletní seznam příkazů je součástí dokumentace na stránkách produktu Docker.

Samostatný servis je tedy možné spustit přímo ze studia, které využívá obdobny příkazu docker-run s využitím předpisů a příkazů obsažených v Dockerfile.

Pro spuštění komplexnějšího projektu s využitím více servis v jednom balíku, například spuštění testovacího webu, message brokera, sql serveru, evenstore, monitoringu a samotného servisu je potřeba využít jiný přístup.

Do Solution, kde je projekt přidán je nutné zařadit podporu orchestrace docker. Tato operace se provede po kliknutí pravým tlačíkem na projekt a vybráním

v nabídce „Add“. Tím se v projektu vygenerují soubory předpisu docker-compose (Obrázek 34 Visual Studio a docker-compose).



Obrázek 34 Visual Studio a docker-compose Zdroj: vlastní tvorba

Předpis Docker compose je nadstavbou nad Dockerfile (Obrázek 35 Nastavení docker-compose.yml). Slouží pro souborné spuštění více servis ve vzájemné orchestraci. Každá sekce označená aliasem servisu nabízí velkou škálu parametrů pro nastavení vytvářeného kontejneru jako například způsob vytvoření image, název kontejneru, činnost po výpadku, přihlašovací informace, nastavení veřejných a vnitřních portů pro komunikaci, úložiště lokálních dat, cestu k souboru docker file nebo určení vzájemné závislosti.

```
version: '3.4'
services:
  sqlserver:
    image: ${DOCKER_REGISTRY-}mcr.microsoft.com/mssql/server:2017-latest-ubuntu
    hostname: 'sqlserver'
    restart: always
    container_name: 'sqlserver'
    environment:
      ACCEPT_EULA: Y
      SA_PASSWORD: "Password123"
    ports:
      - '1433:1433'
    expose:
      - 1433
    volumes:
      - /var/container_data/mysql:/var/lib/mysql mysql_image smi
  gateway:
    image: ${DOCKER_REGISTRY-}webapi
    hostname: "gateway"
    restart: always
    ports:
      - "8030:80"
    build:
      context: .
      dockerfile: Api/WebApi/Dockerfile
```

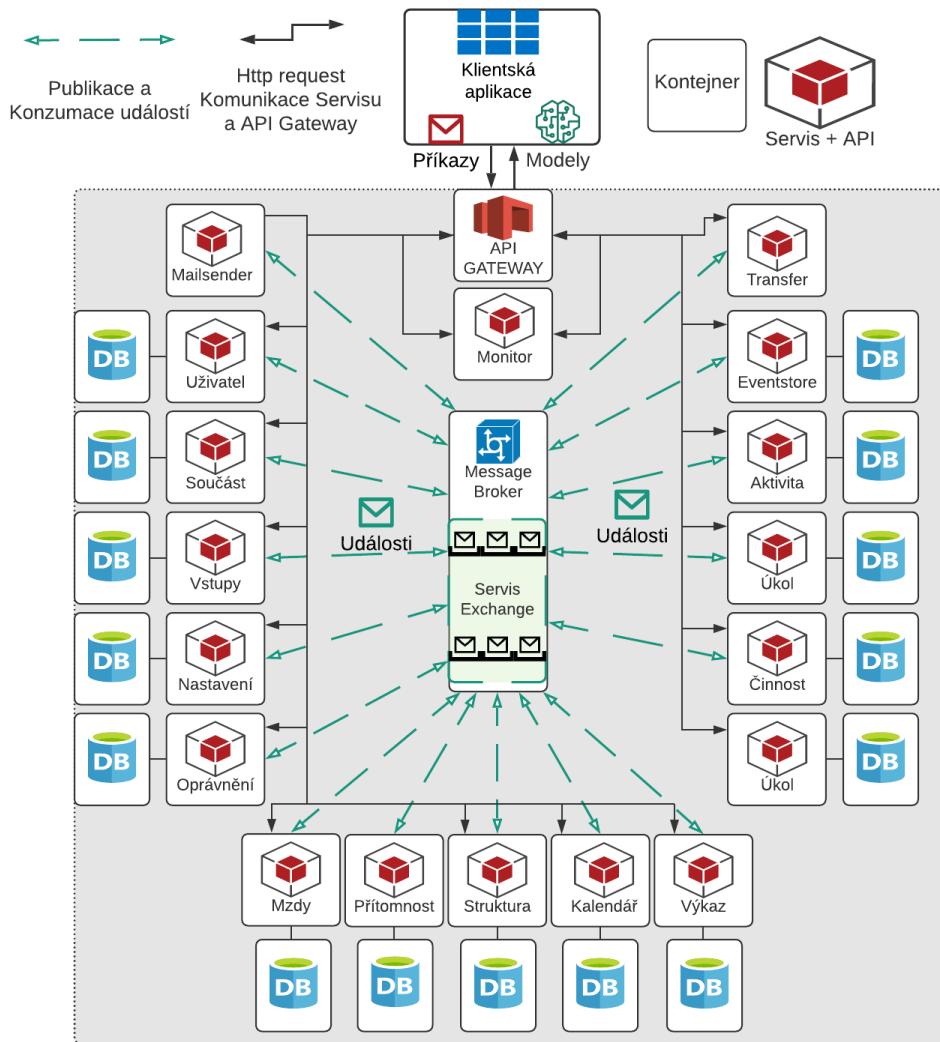
Obrázek 35 Nastavení docker-compose.yml Zdroj: vlastní tvorba

Docker-Compose je využíván na správu a spuštění celého systému v prostředí docker, ale také na umožnění samostatného vývoje každé části zvlášť.

V případě práce na jednom servisu můžou být týmu poskytnuty již ověřené a schválené funkční image dalších servisů, které chtějí využít a ověřit s nimi vzájemnou spolupráci.

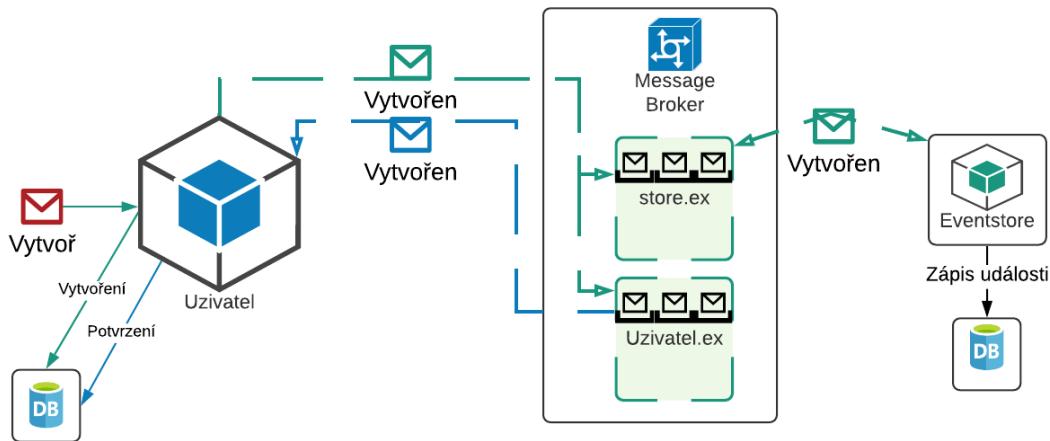
## 7.6 Struktura a komunikace

Struktura aplikace a počet servis je převzatá z doménově orientovaného návrhu (Obrázek 36 Prvky a struktura systému v MSA). Každé definované suboméně je vyhrazen jeden kontejner obsahující servis. V návrhu je pro každý kontejner servisu vyhrazen jeden kontejner obsahující databázový server s databází čistě pro konkrétní servis, aby byla dodržena samostatnost a izolace chybovosti. V rámci vývoje a úspory zdrojů je využíván jeden společný kontejner s databázovým serverem a databází pro každý servis. Klientská aplikace díky popsanému rozhraní aplikace na API získá informaci o struktuře příkazů (Commands) a podobě modelů, které budou vráceny na volání metod Get API Gateway tyto http požadavky překládá a volá pomocí http API servisů. Servisy vzájemně prostřednictvím http nekomunikují. Výjimkou je servis Monitor, který hlídá stav a zdraví systému. Pro realizaci plánované architektury bude zapojeny vybrané technologie zmíněné v teoretické části této práce.



Obrázek 36 Prvky a struktura systému v MSA Zdroj: vlastní tvorba

Servisy přijímají zasílané příkazy, odpovídají na metody typu Get a na základě prováděných akcí publikují zprávy s událostmi (Events), které publikují na svůj příslušný exchange a frontu v Message Brokeru. Zároveň poslouchají aktivitu na svém exchange a i dalších zájmových. V případě, že na zájmové exchange přijde zpráva s událostí, která je zajímá, provedou na jejím základě svou akci a vydají zprávu. Některé servisy převážně publikují a jiné mohou například pouze konzumovat a poskytovat klientské části pouze metody pro čtení. Tím je možné realizovat princip odděleného zápisu a čtení CQRS (Obrázek 37 Komunikace mikroservisu).



Obrázek 37 Komunikace mikroservisu Zdroj: vlastní tvorba

Virtualizaci a orchestraci bude zajištěna pomocí Docker desktop a vývojovu verzí Kubernetes. Všechny kontejnery tak budou provozovány s operačním systémem Linux.

Webová aplikace je plánována s využitím technologie Asp.NET Core 3.1 a programovacího jazyka C#. Další varianta webu ve verzi SPA bude realizována za pomoci AngularJS nebo jeho vyšších verzí.

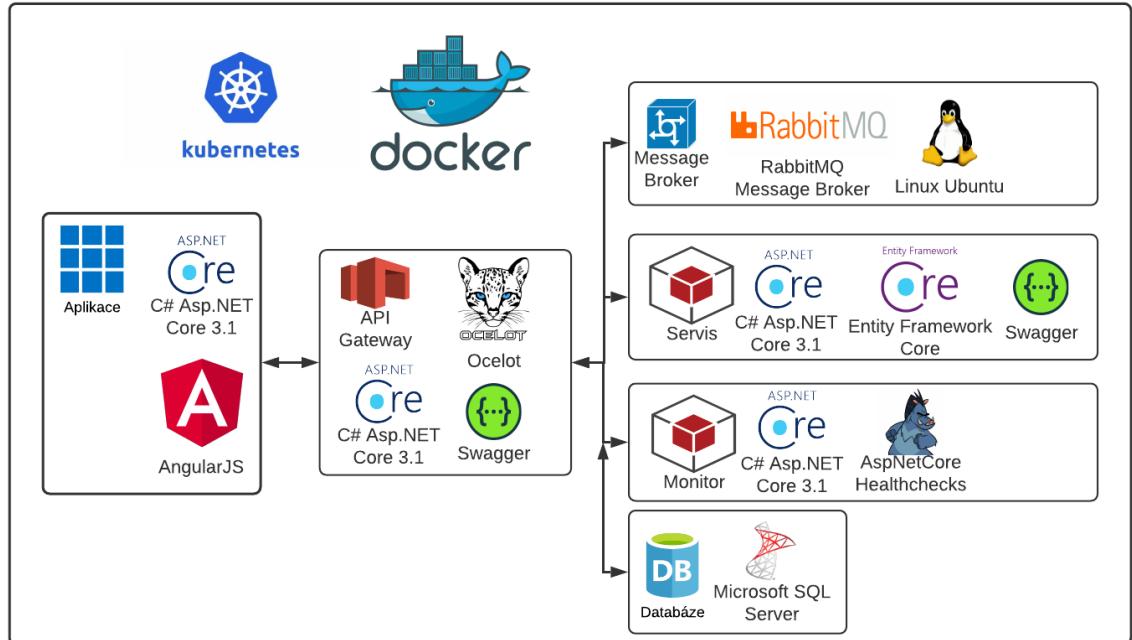
Servis zajišťující funkciálnitu API Gateway bude opět v .NET Core s využitím příslušné verze balíčku Ocelot pro zajištění překladu požadavků a řízení výkonu. Pro jednotný popis rozhraní bude využit modul Swagger, který je doplněn o nugetový balíček Swashbuckle Core.

Jako message broker bude nasazen RabbitMQ prostřednictvím virtualizovaného kontejneru v doker běžícím na Linux Ubuntu.

Obecné servisy budou opět v .NET Core s využitím Entity framework Core pro databázové perace. Pro popis rozhraní API konzumované na úrovni API gateway bude opět využit modul Swagger s doplnkem Swashbuckle.

Monitoring bude zajišťován specializovanou službou s využitím prostředků poskytnutých v .NET core a doinstalovaným nuget balíčkem AspNetCore.Healthchecks.

Pro databázový server bude nasazen Microsoft SQL server, který má pro potřeby dockeru podporovanou verzi běžící na Linux  
(Obrázek 38 Technologie ve struktuře MSA).



Obrázek 38 Technologie ve struktuře MSA Zdroj: vlastní tvorba

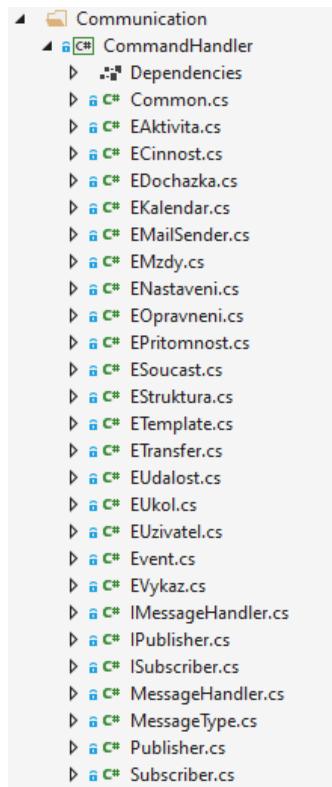
## 7.7 Události

Komunikaci mezi servisy v rámci systému je nutné vytvořit množinu událostí ke každému servisu obsahující informace o objektech podle jejich domény. Díky informacím, které události nesou, je pak možné provádět všechny potřebné operace, udržovat stav o konkrétních entitách nebo skládat pohledy na různé entity podle potřeby. Zároveň je nutné definovat objekty příkazů, které systému vydávají úvodní impuls k činnosti.

Pro tyty účely a také z potřeby zajistit jednotného přístupu k tvorbě událostí je vytvořena sdílená knihovna Command Handler (Obrázek 39 Složení knihovny událostí). Knihovna je přidána jako doplněk ke každému projektu využívající zasílání zpráv. Obsahuje třídy pro každou událost a příkaz ke každému servisu, jednotný číselník typu všech zpráv MessageType a třídy Publisher a Subscriber pro využití metod publikace a konzumace. Třída MessageHandler pak předpis pro jednotné

zpracování všech zpráv předávaných na Message Broker. Tato knihovna není určena k virtualizaci, je vždy pouze nakopírováná k projektu servisu a nevytváří tak žádné externí reference.

V případě provedení změn ve struktuře událostí je potřeba zavést úroveň verzování zpráv tak, aby starší verze, které jsou distribuované do již provozovaných služeb, byly stále kompatibilní jak na publikaci, tak při jejich zpracování.



Obrázek 39 Složení knihovny událostí Zdroj: vlastní tvorba

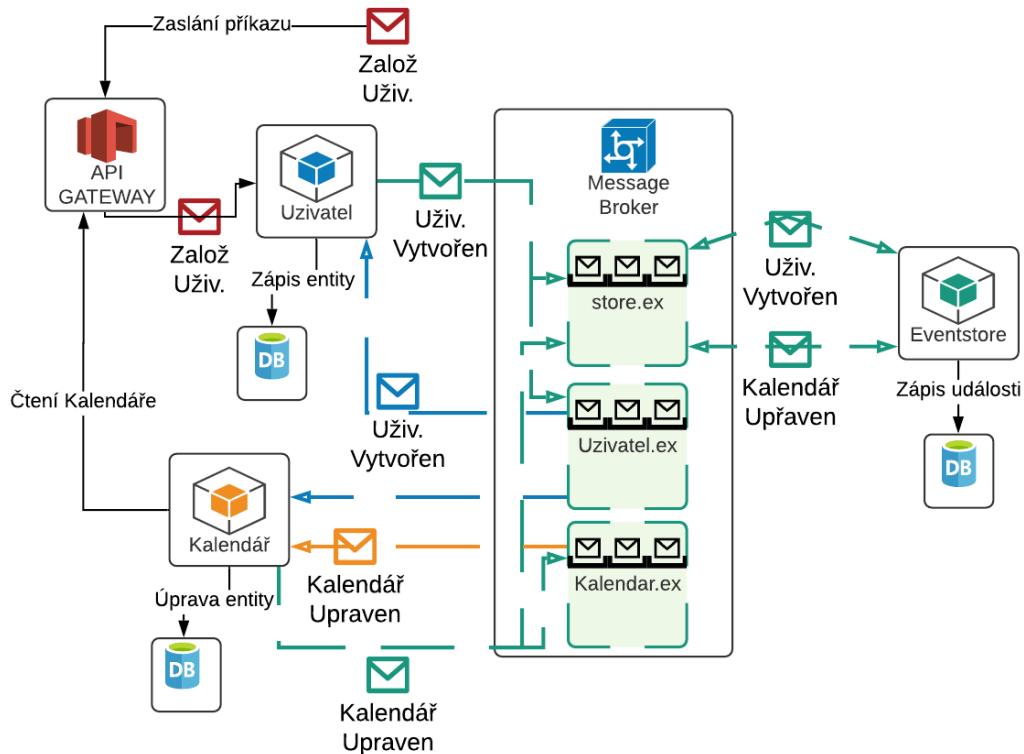
Popis třídy Subscriber, Publisher, MessageHandler, MessageType a EUzivatel jsou součástí přílohy D – Události.

### 7.7.1 Událostní řízení

Komunikace a reakce servis tedy závisí na publikování a konzumaci událostí. Jsou tedy jsou událostně řízeny. Zasláním jediného příkazu může být v systému spuštěn řetězec událostí, který vede ke změně jeho celkového stavu. Všechny události jsou zaznamenávány v eventstore. Na obrázku (Obrázek 40 Událostní řízení) je naznačen příklad kdy, je odeslán příkaz založení uživatele. Uživatel je založen, zapsán do databáze a událost odelsána do evenstore. V případě, že uložení do evenstore

proběhlo v pořádku, je publikována stejná událost na exchange Uživatele. Tuto událost konzumuje jak uživatel tak servis Kalendář. Uživatel na základě události nově založeného uživatele upraví a potvrdí tak událostní konzistenci.

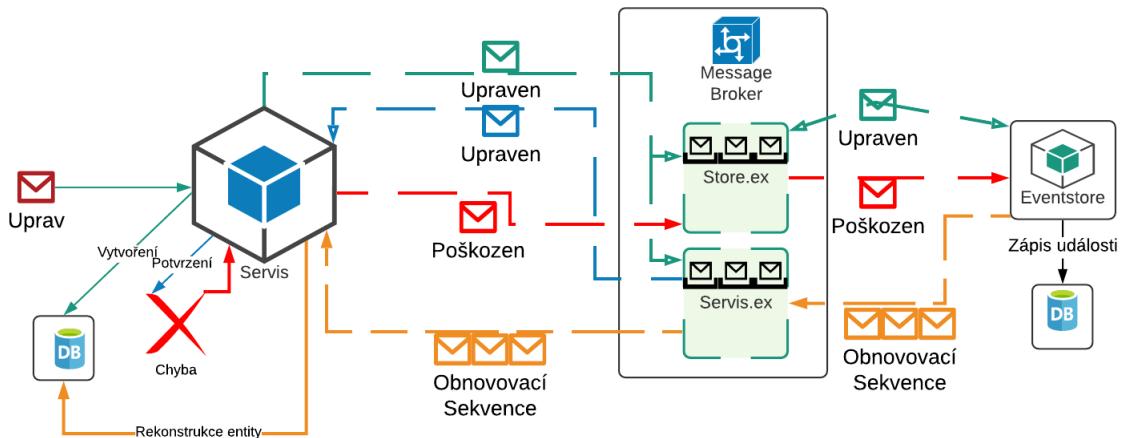
Servis kalendář pouze konzumuje událost, provede úpravy ve svých záznamech a dále je poskytuje pouze pro čtení. O své akci také vydá zprávu, která je uložena do eventstore.



Obrázek 40 Událostní řízení Zdroj: vlastní tvorba

### 7.7.2 Událostní Konzistence

Výhodou událostního řízení je událostní konzistence. Schéma (Obrázek 41 Komunikace a událostní konzistence) popisuje průběh operace provedení úpravy záznamu v servisu. Na servis přijde příkaz „uprav“, ten je zpracován a proveden zápis údajů do databáze s indikací generace a identifikátorem vytvořené události.



Obrázek 41 Komunikace a událostní konzistence Zdroj: vlastní tvorba

Událost „Upraven“ je předána message brokeru ke zpracování. Ten formou pseudotransakce zařadí zprávu do eventstore a zařadí ji do příslušného exchange servisu pro možnost konzumace dalšími službami. Tím je zajištěna pomyslná pseudotransakce. Pokud na úrovni servisy dojde k vyjímcce při publikaci, zápis do databáze se nepovede.

Po úspěšné publikaci servis poslouchá svůj exchange, konzumuje událost „Upraven“, provede kontrolu zapsaného záznamu a porovná zapsaný identifikátor a generaci s přijatou událostí. Pokud je vše v pořáku, zvedne generaci záznamu a proces dál nepokračuje.

V případě, že servis zjistí nesrovnalost mezi záznamem a událostí, například pokud mezikoupky než stihl konzumovat událost „upraven“ proběhl další update, tak odešle směrem ke službě eventstore novou událost indikující nekonzistentní stav. Služba evenstore tuto zprávu konzumuje a na základě v ní obdržených informací publikuje sekvenci všech zpráv k dané entitě nebo případně všem záznamům k dané subdoméně.

Servis konzumuje tento opravný proud a na jeho základě materializuje od začátku požadované záznamy do aktuálního stavu, který je udržován a organizován v evenstore.

Tento mechanismus slouží k udržení konzistence stavu entit a je také využit k automatickému obdovení databáze servisy v případě její ztráty nebo migrace

na jinou technologii. Po restartu si může každý servis odeslat žádost o získání aktuálního stavu své oblasti a ten promítнуть do své databáze.

Systém je tedy událostně konzistentní a testo stav je udržován ve službě eventstore.

Mechanismus pro publikování události o nekonzistentním stavu a způsob zpracování je součástí.

## 7.8 Servis Gateway

V rámci prvního sprintu bylo pro funkčnost dema a celého budoucí aplikace implementace servisu API Gateway. Byla doinstalována komponenta Ocelot, která zajišťuje překlad veřejných adres na vnitřní a zároveň provádí mapování popisu API jednotlivých servisů, které následně vystaví v uživatelském rozhraní přístupném na Gateway. Díky vystaveném rozhraní, které popisuje podporované metody, zpracovávaný model a příkazy lze vytvořit v klientské aplikaci třídy http klienta pomocí nástroje Nswag. Ukázka konfigurace servisu a použitých modulů je součástí Příloha I Servis – Gateway.

## 7.9 Message Broker

Jedním z centrálních a klíčových prvků celého systému je Message broker využívající technologii RabbitMQ.

RabbitMQ je instalován jako samostatný kontejner pomocí konfigurace v docker-compose.yml.

Pro připojení servis k Message Brokeru je v projektu vytvořena sdílená knihovna CommandHandler, která poskytuje předpis všech událostí, které lze v systému využívat a třídy pro vytvoření objektu Publisher a Subscriber, které mají metody pro publikaci a přihlášení se k odběru zpráv. V každém servisu ve třídě Startup je pak vložena metoda pro navázání spojení. Konfigurace každého servisu obsahuje nastavení vlastního Exchange pro výměnu vlastních událostí, Exchange eventstore pro zaslání žádosti o obnovu dat. Dále obsahuje seznam zájmových Exchange, od kterých chce servis odebírat jejich události. Popis implementace je součástí řílohy

popisující servis Uživatel Api (Příloha F Servis – Uživatel API) a z pohledu Message brokera v popisu knihovny CommandHandler (Příloha H Příkazy a události, Publikace a Odběr).

## 7.10 Servis Monitor

Další důležitou komponentou v mikroservisním systému je servis pro účely monitoringu. V rámci servisu je využito mechnismů pro kontrolu stavu a zdraví, které je obsaženo v .NET Core a je možné připojit další rozšíření. Jako rozšíření byl zvolen doporučený balíček `aspnetcore.healthchecks` který podporuje všechny využité technologie v budované aplikaci.

Jednotlivé kontrolní body jsou implementovány při startu každého servisu a monitor je pak při svém startu pomocí nastavené adresy skrze alias z orchestrátoru spojí do jednoho uživatelského rozhraní. V pravidelných interval pak odesílá požadavek na zaslání hlášení. V uživatelském rozhraní pak umožní kontrolu, zda jsou servisy bez závad a zda mají vytvořené spojení s message brokerem a databází. V dalších fázích projektu je pak možnost navázat na automatizované akce na některé ze stavů, které mohou u servisů nastat. Příklad implementace monitoru obsahuje Příloha J Servis – Monitor.

## 7.11 Servis Event Store

Servis eventstore slouží jako hlavní komponenta, která v sobě ukládá všechny publikované události a drží tak konzistenci stavu v celém systému. V rámci publikace se všechny vytvářené události zasílají na specializovaný Message Broker Exchange, ke kterému je Even Store přihlášen. Při publikaci Message Broker také umožnuje funkcionality pesudotransakce, kdy je odeslání na Exchange servisu a eventstoru uzavřen v do jednoho bloku kódu u kterého je vyžadováno ověření, zda byly operace provedeny v pořádku. Evenstore má také vytvořeny své speciální události, které detekují nekonzistentní stav servisu. V případě že servis eventstore tyto zprávy zachytí, reaguje na ně podle údajů ve zrávě a odešle z logu požadovaný

obdnovovací proud událostí, případně poskytne požadovaný časový interval pro požadovaný historický pohled.

Příklad základní implementace servisu Event store je obsažen v příloze (Příloha K Servis – EventStore).

## 7.12 Servis Kalendář

Servis Kalendář je komponenta tvořená v půběhu sprintu č. 3. Účelem servisu je generovat kalendář obsahující aktivity, úkoly a činnosti, které uživatel zaznamenal. Kalendář je určen jako servis, sloužící čistě jen ke konzumaci událostí. Poskytuje podporu pro princip CQRS. Například servis Aktivita provádí především zápis a úpravy všech aktivit, které jsou následně pro potřeby klientské aplikace vyčítány prostřednictvím servisu Kalendář. Kalendář konzumuje události zasílané servisem uživatel, aktivity, úkoly, činnosti atp. V případě, že založíme uživatele, kalendář tuto informaci detekuje a automaticky vytvoří příslušný záznam v databází, obsahující serializovaný kalendář ve formátu JSON. Pokud uživatel zadá aktivitu, servis opět událost konzumuje, podle uživatelského identifikátoru a roku najde příslušný kalendář, provede deserializaci do objektu třídy kalendář, přidá údaje o aktivitě do příslušného datumu, kalendář opět serializuje a uloží. Příklad implementace servisu Kalendář je dále rozveden v příloze (Příloha L Servis Kalendář).

## 7.13 Prototyp

V rámci praktického části této práce byla využita metodika Scrum za účelem plánování a návrhu aplikace založené na mikroservisní architektuře. Byl vytvořen doménový model a struktura projektu, byla odhadnuta podoba prvních sprintů a provedena realizace jednotlivých cílů. V rámci projektu byly ověřeny teoretické poznatky z oblasti MSA a využití agilních metodik. Výstupem je pak prototyp plánované aplikace, který završuje zahajovací fázi projektu, který lze dále dorvájet podle vize naznačené v produkční backlogu. Prototyp je dostupný k nahlédnutí na adresu <https://github.com/JosefRajský/Kancelar>

## Závěr

Cílem této diplomové práce bylo porovnání mikroservisní architektury využívající pro komunikaci událostního řízení s dalšími používanými architekturami. V praktické části práce bylo cílem využít jednu z vybraných metodik pro návrh a realizaci projektu, resp. vytvoření aplikace využívající událostní řízení založené na v součanosti velmi populární mikroservisní architektuře.

V rámci porovnání byly zmíněny hlavní znaky, výhody a nevýhody každé z architektur. Byly uvedeny důvody jejich vzniku, jejich vzájemná logická provázanost a způsob uspokojení nároků na tvorbu moderních aplikací. V kapitole věnované mikroservisní architektuře byly postupně uvedeny a popsány jednotlivé prvky, ze kterých jsou mikroservisní aplikace složeny a také technologie, které jsou pro tyto prvky využívány. Samotným technologiím se věnovala zvláštní část práce, kde bylo vyhodnoceno možné využití poznatků a principů mikroservis pro tvorbu uživatelského prostředí na klientské straně a bylo provedeno porovnání vhodných technologií pro účely realizace.

V průběhu praktické části, byl na příkladu fiktivní organizace zpracován návrh projektu aplikace pro podporu podnikových procesů pro řízení aktivit, personálních zdrojů a finančních zdrojů. V rámci projektu bylo využito agilního přístupu, konkrétně metodiky Scrum, a zpracováno prvních několik částí aplikace. Účelem bylo potvrdit a ověřit vhodnost postupů agilní metodiky pro podporu vývoje aplikace v mikroservisní architektuře.

Vzniklý prototyp aplikace je složen z komponentů a mikroservisů, které jsou organizovány podle doménově orientovaného návrhu. Aplikace využívá událostního řízení pro komunikaci a je schopná udržovat událostně konzistentní stav. V rámci realizace byla potvrzena očekávání ze vzájemného porovnání architektur, a to především v oblasti zvýšených nároků na úvodní fáze vývoje. Zajímavá byla realizace technik událostního řízení, které jsou oproti standardu odlišné především v přístupu k uchování stavu objektů a v oblasti práce s daty.

Praktickou část práce lze považovat za dokončenou úvodní fázi projektu. Projekt jako celek je připraven pro další rozšíření a zlepšení.

Stanovené cíle považuji za splněné jak v teoretické, tak v praktické části práce.

## **Seznam bibliografických odkazů**

- [1] Agile Aliance [online]. Corryton, Tennessee: Agile Aliance, 2001 [cit. 2020-02-29]. Dostupné z: <https://www.agilealliance.org/>
- [2] ARMSTRONG, Chris. *Modeling Web Services Modeling Web Services with UML: OMG Web Services Workshop 2002*. OMG Web Services Workshop 2002 [online]. Roseville, MN: ATC Enterprises, 2002 [cit. 2020-02-29]. Dostupné z:  
[http://petros.omg.org/news/meetings/workshops/presentations/WebServices\\_2002/03-2\\_Armstrong-ModelingWebServices\\_with\\_UML.pdf](http://petros.omg.org/news/meetings/workshops/presentations/WebServices_2002/03-2_Armstrong-ModelingWebServices_with_UML.pdf)
- [3] ARSANJANI, Ali. *Service-oriented modeling and architecture*. Ibm.com [online]. Armonk, New York: ibm.com, 2004, 9.2.2004 [cit. 2020-02-29]. Dostupné z:  
<https://www.ibm.com/developerworks/library/ws-soa-design1/index.html>
- [4] ATHRI, Vinugay. *5 Best Technologies To Build Microservices Architecture*. Clariontech [online]. clariontech.com: Clarion technologies, 2017, 21.9.2017 [cit. 2020-04-26]. Dostupné z:  
<https://www.clariontech.com/blog/5-best-technologies-to-build-microservices-architecture>
- [5] BABAR, Muhammad, Alan BROWN a Ivan MISTRIK. *Agile software architecture: aligning agile processes and software architectures*. Waltham, MA: Elsevier, 2014. ISBN 978-0-12-407772-0. [5]
- [6] BADRAL, Sanlig. *Web Services*. BDIT Haase [online]. BDIT Haase: BDIT Haase, 2015 [cit. 2020-02-29]. Dostupné z:  
<http://www.badaa.mngl.net/content/webservices.pdf>
- [7] BARASHKOV, Alex. *Microservices vs. Monolith Architecture. Dev* [online]. Dv: dev.to, 2018, 4.12.2018 [cit. 2020-04-26]. Dostupné z:  
[https://dev.to/alex\\_barashkov/microservices-vs-monolith-architecture-4l1m](https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m)

- [8] BECK, Kent. *Extrémní programování*. Praha: Grada, 2002. Moderní programování. ISBN 80-247-0300-9.
- [9] BRDA, Jiří. *9 nejnovějších webdesignových trendů pro rok 2017*. Jiribrda.cz [online]. jiribrda.cz: jiribrda.cz, 2017, 9.1.2017 [cit. 2020-04-29]. Dostupné z: <http://www.jiribrda.cz/9-nejnovějších-webdesignových-trendů-pro-rok-2017.html>
- [10] BROWN, Kyle. *Beyond buzzwords: A brief history of microservices patterns*. Ibm.com [online]. Armonk, New York: IBM, 2018, 10.10.2018 [cit. 2020-03-24]. Dostupné z: <https://developer.ibm.com/technologies/microservices/articles/cl-evolution-microservices-patterns/>
- [11] BRUCKNER, Tomáš. *Tvorba informačních systémů: principy, metodiky, architektury*. Praha: Grada, 2012. Management v informační společnosti. ISBN 978-80-247-4153-6.
- [12] BRUNO, Eric J. *SOA, Web Services, and RESTful Systems*. Drdobbs.com [online]. drdobbs.com: drdobbs.com, 2007, 8.6.2007 [cit. 2020-02-29]. Dostupné z: <https://www.drdobbs.com/web-development/soa-web-services-and-restful-systems/199902676>
- [13] BUREŠ, Michal. *Přínosy SOA pro integraci ERP systémů*. In: Systemonline [online]. 2014 [cit. 2019-11-5]. Dostupné z: <https://www.systemonline.cz/erp/prinosy-soa-pro-integraci-erp-systemu.htm>
- [14] CALÇADO, Phil. *The Back-end for Front-end Pattern (BFF)*. Philcalcado.com [online]. New York City: philcalcado.com, 2015, 18.8.2015 [cit. 2020-03-24]. Dostupné z: [https://philcalcado.com/2015/09/18/the\\_back\\_end\\_for\\_front\\_end\\_pattern\\_bff.html](https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html)
- [15] CAREY, Scott. *What are microservices?* Computerworld [online]. computerworld.com: computerworld.com, 2018, 10.10.2018 [cit. 2020-04-26]. Dostupné z: <https://www.computerworld.com/article/3427107/what-are-microservices-.html>

- [16] ČÁPKA, David. *Úvod do Single Page Application* v *ASP.NET*. Itnetwork.cz [online]. itnetwork.cz: itnetwork.cz, 2014 [cit. 2020-04-29]. Dostupné z: <https://www.itnetwork.cz/csharp/asp-net/single-page-application/tutorial-uvod-do-asp-net-single-page-application>
- [17] DANIEL, Requejo a Villanueva FERNANDO. *To go or not to go micro: the pros and cons of microservices*. Medium [online]. medium.com: Medium, 2018, 16.9.2018 [cit. 2020-04-26]. Dostupné z: <https://medium.com/@goodrebels/to-go-or-not-to-go-micro-the-pros-and-cons-of-microservices-7967418ff06>
- [18] DE LA TORRE, Cesar, Bill WAGNER a Mike ROUSOS. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Docs [online]. Redmond, Washington: Microsoft Developer Division, 2020 [cit. 2020-02-29]. Dostupné z: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/>
- [19] *Developer Survey Results 2019* [online]. stackoverflow.com: stackoverflow.com, 2019 [cit. 2020-03-24]. Dostupné z: <https://insights.stackoverflow.com/survey/2019#technology>
- [20] *Docker Alternatives | Top 8 Docker Alternatives with Pros and Cons* [online]. Mumbai, India: eduCBA, 2020 [cit. 2020-03-24]. Dostupné z: <https://www.educba.com/docker-alternatives/>
- [21] DONOVAN, Ryan. *The Top 10 Frameworks and What Tech Recruiters Need to Know About Them*. Stackoverflow [online]. New York: stackoverflow, 2008, 2019 [cit. 2020-03-24]. Dostupné z: <https://stackoverflow.blog/2019/12/17/the-top-10-frameworks-and-what-tech-recruiters-need-to-know-about-them/> [21]
- [22] ESCHWEILER, Sebastian. *Docker – Beginner's Guide – Part 1: Images & Containers*. Codingthesmartway [online]. Codingthesmartway.com: Codingthesmartway, 2019, 24.11.2019 [cit. 2020-04-26]. Dostupné z: <https://codingthesmartway.com/docker-beginners-guide-part-1-images-containers/>

- [23] *Extreme Programming (XP) vs Scrum*. Visual-paradigm [online]. visual-paradigm: visual-paradigm, 2019, 15.1.2019 [cit. 2020-04-26]. Dostupné z: <https://www.visual-paradigm.com/scrum/extreme-programming-vs-scrum/>
- [24] FAWCETT, Amanda. *Microservices Architecture Tutorial: all you need to get started*. Educative.io [online]. <https://www.educative.io>: educative.io, 2020, 17.3.2020 [cit. 2020-04-26]. Dostupné z: <https://www.educative.io/blog/microservices-architecture-tutorial-all-you-need-to-get-started>
- [25] FONG, Jenny. *Integrating Kubernetes with Docker Enterprise Edition 2.0 – Top 10 Questions from the Docker Virtual Event*. Docker [online]. Docker.com: Docker, 2018, 8.5.2018 [cit. 2020-04-26]. Dostupné z: <https://www.docker.com/blog/integrating-kubernetes-docker-enterprise-edition-2-0-top-10-questions-docker-virtual-event/>
- [26] FOWLER, Martin. *Event Sourcing*. Martinfowler.com [online]. Chicago, IL: martinfowler.com, 2005, 12.12.2005 [cit. 2020-03-24]. Dostupné z: <https://martinfowler.com/eaaDev/EventSourcing.html>
- [27] FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, 2003. ISBN 978-0321127426
- [28] FOWLER, Susan J. *Production-ready Microservices: building standardized systems across an engineering organization*. 2017. Sebastopol, CA: O'Reilly, [2017]. ISBN 978-1-491-96597-9
- [29] FRUHLINGER, Josh. *What are microservices? Your next software architecture*. Infoworld.com [online]. infoworld.com: infoworld.com, 2019, 10.10.2019 [cit. 2020-04-26]. Dostupné z: <https://www.infoworld.com/article/3445043/what-are-microservices-your-next-software-architecture.html>
- [30] GARG, Nishant. *Apache Kafka*. Birmingham: Packt Publishing, 2013. ISBN 978-1-78216-793-8.

- [31] GARG, Sarthag. *Service-Oriented Architecture*. Geeksforgeeks [online]. geeksforgeeks.org: geeksforgeeks.org, 2018, 25.3.2018 [cit. 2020-04-26]. Dostupné z: <https://www.geeksforgeeks.org/service-oriented-architecture/>
- [32] GERTNER, Matthew. *After Decades of Neglect, Functional Programming is Finally Going Mainstream. Why Now?* salsitasoft.com [online]. blog.salsitasoft.com/, 2016 [cit. 2020-04-29]. Dostupné z: <http://blog.salsitasoft.com/why-now/>
- [33] GitHub - ThreeMammals/Ocelot: .NET core API Gateway. *The world's leading software development platform* [online]. San Francisco, CA: GitHub, 2020 [cit. 2020-03-24]. Dostupné z: <https://github.com/ThreeMammals/Ocelot>
- [34] GOELBELGECKER, Eric. *Service Oriented Architecture: A Dead Simple Explanation*. Dzone [online]. dzone.com: dzon, 2019, 10.1.2019 [cit. 2020-04-26]. Dostupné z: <https://dzone.com/articles/service-oriented-architecture-a-dead-simple-explan>
- [35] GRØNBÆK, Finn. *Web Services and Service Oriented Architecture*. Citeseerx.ist.psu.edu [online]. IBM Software Group Nordics: IBM Software Group, 2004, 21.9.2004 [cit. 2020-02-29]. Dostupné z: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.200.9148&rep=rep1&type=pdf>
- [36] HAAS, Hugo a Allen BROWN. *Web Services Glossary*. W3C [online]. Cambridge, MA: W3C, 2004, 11.2.2004 [cit. 2020-02-29]. Dostupné z: <https://www.w3.org/TR/ws-gloss/>
- [37] HAYWOOD, Dan. *Domain-driven design using naked objects*. 2009. Raleigh, N.C.: Pragmatic Bookshelf, c2009. Pragmatic programmers. ISBN 1-934356-44-1.
- [38] *Health checks in ASP.NET Core | Microsoft Docs*. [online]. Microsoft: Microsoft, 2020 [cit. 2020-03-24]. Dostupné z: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-3.1>

- [39] HIFI, Lidan. *Building Event-Driven Microservices with Event Sourcing and CQRS*. In: Youtube [online]. 15.8.2018 [cit. 2019-11-5]. Dostupné z: <https://youtu.be/XWTrcBqXi6s>. Kanál uživatele NDC Conferences.
- [40] HUGHEY, Douglas. *The Traditional Waterfall Approach*. umsl.edu [online]. umsl.edu: umsl.edu, 2009, 1.5.2009 [cit. 2020-04-26]. Dostupné z: <https://www.umsl.edu/~hugheyd/is6840/waterfall.html>
- [41] HURWITZ, Judith a Judith HURWITZ. *Service oriented architecture for dummies*. 2007. Hoboken, NJ: Wiley Publishing, c2007. ISBN 978-0-470-05435-2.
- [42] HYBÁŠEK, Michal (2009). *Webové MVC rámce na platformě Java* [online]. [cit. 2020-02-29]. Brno: Vysoké učení technické v Brně. Fakulta informačních technologií. Dostupné z: <https://dspace.vutbr.cz/xmlui/handle/11012/53802>
- [43] CHINNICI, Roberto, Jean-Jacques MOREAU, Arthur RYMAN a Sanjiva WEERAWARANA. *Web Services Description Language (WSDL)*. W3C [online]. Cambridge, MA: W3C, 2004, 26.6.2007 [cit. 2020-02-29]. Dostupné z: <https://www.w3.org/TR/wsdl/>
- [44] IBM Knowledge Center, *Developing web service applications* [online]. New York: IBM Corporation, 2016 [cit. 2020-02-29]. Dostupné z: <https://www.ibm.com/support/knowledgecenter/>
- [45] JONÁK, Stanislav. *Jak se měnil přístup k vývoji SW během 20. století?* middleware.cz [online]. 2009 [cit. 2016-12-01]. Dostupné z: <http://www.middleware.cz/projektove-rizeni0/23-jak-se-menil-pristup-k-vyvoji-SW-behem-20-stoleti>
- [46] JOSUTTIS, Nicolai M. *SOA in practice*. Sebastopol: O'Reilly, 2007. ISBN 0-596-52955-4.
- [47] KADLEC, Václav. *98 % zakázek neúspěšných? Ještě že máme softwarové inženýrství!* zive.cz [online]. 2002 [cit. 2016-12-01]. Dostupné z:

- <http://www.zive.cz/clanky/98--zakazek-neuspesnych-jeste-ze-mame-softwarove-inzenyrstvi/sc-3-a-105343/default.aspx>
- [48] KADLEC, Václav. *Agilní programování: metodiky efektivního vývoje softwaru*. Brno: Computer Press, 2004. ISBN 80-251-0342-0.
- [49] KANJILAL, Joydip. *ASP.NET Web API: Build RESTful web applications and services on the .NET framework*. 2013. Birmingham, UK.: Packt Publishing, 2013. ISBN 978-1-84968-974-8.
- [50] KOROTYA, Eugeniya (2017). *5 Best JavaScript Frameworks in 2017* [online]. [cit. 2020-02-29]. Dostupné z: <https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282>
- [51] KRUCHTEN, Philippe. *The rational unified process: an introduction*. 3rd ed. Boston: Addison-Wesley, c2004. ISBN 0321197704.
- [52] LOWE, Steven. *Get your feet wet with domain-driven design: 3 guiding principles*. Techbeacon [online]. techbeacon.com: techbeacon.com, 2019, 2.4.2019 [cit. 2020-04-26]. Dostupné z: <https://techbeacon.com/app-dev-testing/get-your-feet-wet-domain-driven-design-3-guiding-principles>
- [53] LUMETTA, Jake. *5 guiding principles you should know before you design a microservice*. Opensource.com [online]. Raleigh, NC: Red Hat, 2018, 19.5.2018 [cit. 2020-03-24]. Dostupné z: <https://opensource.com/article/18/4/guide-design-microservices>
- [54] MACKENZIE, Matthew a Ken LASKEY. *Reference Model for Service Oriented 3 Architecture*. Oasis-open [online]. www.oasis-open.org/: oasis-open, 2006, 7.2.2006 [cit. 2020-04-26]. Dostupné z: <https://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf>
- [55] MAHMOUD, Qusay H. *Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)*. Oracle.com [online]. 1.5.2005: oracle, 2005 [cit. 2020-02-29]. Dostupné z: <https://www.oracle.com/technical-resources/articles/javase/soa.html>
- [56] MARKS, Eric A. *Service-oriented architecture governance for the services driven enterprise*. 2008. Hoboken, N.J.: Wiley, c2008. ISBN 978-0-470-17125-7. [

- [57] MCLARTY, Matt. *Microservice architecture is agile software architecture*. Infoworld [online]. infoworld.com: Infoworld, 2016, 2016 [cit. 2020-04-26]. Dostupné z: <https://www.infoworld.com/article/3075880/microservice-architecture-is-agile-software-architecture.html>
- [58] *Microservices.io* [online]. microservices.io: microservices.io, 2020 [cit. 2020-02-29]. Dostupné z: <https://microservices.io/>
- [59] MILLETT, Scott. *Patterns, principles, and practices of domain-driven design*. 2015. Indianapolis, IN: wrox, a Wiley Brand, [2015]. ISBN 978-1-118-71470-6.
- [60] MULLER, Eelco. *Angular 2 vs. Angular 1: Key Differences*. Dzone.com [online]. dzone.com: dzone.com, 2015, 11.9.2015 [cit. 2020-04-29]. Dostupné z: <https://dzone.com/articles/typed-front-end-with-angular-2>
- [61] NADAREISHVILI, Irakli. *Microservice architecture: aligning principles, practices, and culture*. Sebastopol, CA: O'Reilly Media, 2016. ISBN 978-1-491-95625-0.
- [62] NARKHEDE, Neha, Gwen SHAPIRA a Todd PALINO. *Kafka: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1-491-99065-0.
- [63] NEWMAN, Sam. *Building microservices: designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1-491-95035-7.
- [64] NITIN, Kumar. *Enterprise Benefits on Service Oriented Architecture – SO*. Javacodegeeks.com [online]. avacodegeeks.com: avacodegeeks.com, 2013, 28.3.2013 [cit. 2020-03-24]. Dostupné z: <https://www.javacodegeeks.com/2013/03/enterprise-benefits-on-service-oriented-architecture-soa.html>
- [65] PAPAZOGLOU, Mike P. a Willem-Jan VAN DEN HEUVEL. *Service oriented architectures: approaches, technologies and research issues*. The VLDB Journal [online]. 2007, 16(3), 389-415 [cit. 2020-02-29]. DOI: 10.1007/s00778-

007-0044-3. ISSN 1066-8888. Dostupné z:

<http://link.springer.com/10.1007/s00778-007-0044-3>

- [66] PATNI, Sanjay. *Pro RESTful APIs: design, build and integrate with REST, JSON, XML and JAX-RS*. Berkeley, California: Apress, 2017. Books for professionals by professionals. ISBN 978-1-48422664-3.
- [67] PETTEY, Christy. *4 Steps to Design Microservices for Agile Architecture*. Gartner [online]. Gartner.com: Gartner, 2018, 7.10.2018 [cit. 2020-04-26]. Dostupné z: <https://www.gartner.com/smarterwithgartner/4-steps-to-design-microservices-for-agile-architecture/>
- [68] POWELL-MORSE, Andrew. *Rational Unified Process: What Is It And How Do You Use It?* Airbrake.io [online]. San Francisco, CA: airbrake.io, 2017, 14.3.2017 [cit. 2020-02-29]. Dostupné z: <https://airbrake.io/blog/sdlc/rational-unified-process>
- [69] POWELL-MORSE, Andrew. *Waterfall Model: What Is It and When Should You Use It?* Airbrake.io [online]. Airbrake.io: Airbrake.io, 2016, 8.12.2016 [cit. 2020-04-26]. Dostupné z: <https://airbrake.io/blog/sdlc/waterfall-model>
- [70] Proč Facebook React zabil jQuery. Zdrojak.cz [online]. zdrojak.cz: zdrojak.cz, 2014, 19.5.2014 [cit. 2020-04-29]. Dostupné z: <https://www.zdrojak.cz/clanky/proc-facebook-react-zabil-jquery/>
- [71] Ramos, Miguel (2017). *AngularJS Performance: A Survey Study* [online]. [cit. 2020-02-29]. Dostupné z: <https://arxiv.org/abs/1705.02506>
- [72] RAYCAD. *Monolithic vs Microservice Architecture*. Medium [online]. medium.com: Medium, 2018, 20.10.2018 [cit. 2020-04-26]. Dostupné z: <https://medium.com/@raycad.seedotech/monolithic-vs-microservice-architecture-e74bd951fc14>
- [73] React (2017). *A JavaScript library for building user interfaces* [online]. [cit. 2020-02-29]. Dostupné z: <https://facebook.github.io/react/>

- [74] REHMAN, Junaid. *Advantages and disadvantages of service oriented architecture (SOA)*. Itorelease.com [online]. itorelease.com: itorelease.com, 2018 [cit. 2020-03-24]. Dostupné z: <http://www.itorelease.com/2018/10/advantages-and-disadvantages-of-service-oriented-architecture-soa/>
- [75] RICHARDS, Mark. *Microservices vs. Service-Oriented Architecture*. 2015. Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1-491-95242-9.
- [76] RICHARDS, Mark. *Software Architecture Patterns*. 2015. Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1-491-92424-2.
- [77] RICHARDSON, Chris. *Event-Driven Data Management for Microservices*. Nginx [online]. Nginx: Nginx.com, 2015, 4.12.2015 [cit. 2020-04-26]. Dostupné z: <https://www.nginx.com/blog/event-driven-data-management-microservices/>
- [78] RICHARDSON, Chris. *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019. ISBN 978-1491950357.
- [79] RICHARDSON, Chris. *Microservices: From Design to Deployment*. <Https://www.nginx.com/> [online]. San Francisco, CA: Nginx, 2016, 2016 [cit. 2020-02-29]. Dostupné z: <https://www.nginx.com/resources/library/designing-deploying-microservices/>
- [80] RICHARDSON, Leonard a Michael AMUNDSEN. *RESTful Web APIs*. Beijing: O'Reilly, 2013. ISBN 978-1-449-35806-8.
- [81] ROSEN, Michael. *Applied SOA: service-oriented architecture and design strategies*. Indianapolis, Wiley Pub., 2008. ISBN 978-0470223659
- [82] ROUSE, Margaret. *UDDI (Universal Description, Discovery, and Integration)*. Whatis.techtarget.com [online]. Newton, MA: WhatIs.com, 2005, 1.8.2005 [cit. 2020-03-24]. Dostupné z: <https://whatis.techtarget.com/definition/UDDI-Universal-Description-Discovery-and-Integration>

- [83] RUBIN, Kenneth S. *Essential Scrum: a practical guide to the most popular agile process*. 2012. Upper Saddle River, NJ: Addison-Wesley, c2012. ISBN 978-0-13-704329-3.
- [84] SACOLICK, Isaac. What is agile methodology? Modern software development explained. *Infoworld* [online]. www.infoworld.com: www.infoworld, 2020, 25.2.2020 [cit. 2020-04-26]. Dostupné z: <https://www.infoworld.com/article/3237508/what-is-agile-methodology-modern-software-development-explained.html>
- [85] Scrum.org [online]. Lexington, MA: Scrum.org, 2020 [cit. 2020-04-26]. Dostupné z: <https://www.scrum.org/resources/what-is-scrum>
- [86] SCHOONDERWOERD, Ruud. *Process-oriented modeling for SOA, Part 1, A technique for process decomposition*. IBM Cloud Education [online]. Armonk, New York: IBM Cloud Education, 2019, 13.5.2019 [cit. 2020-02-29]. Dostupné z: <https://www.ibm.com/developerworks/library/ar-procmod1/index.html>
- [87] SCHWABOSKY, Jonathan. *Microservices Choreography vs Orchestration: The Benefits of Choreography*. Solace [online]. Solace.com: Solace, 2019, 26.11.2019 [cit. 2020-04-26]. Dostupné z: <https://solace.com/blog/microservices-choreography-vs-orchestration/>
- [88] SIMONS, Eric. *What exactly is React?* [Https://thinkster.io](https://thinkster.io) [online]. <https://thinkster.io>: thinkster.io, 2017 [cit. 2020-04-29]. Dostupné z: <https://thinkster.io/tutorials/what-exactly-is-react>
- [89] SMITH, Floyd. *Designing and Deploying Microservices*. In: Nginx [online]. 2016 [cit. 2019-11-5]. Dostupné z: <https://www.nginx.com/resources/library/designing-deploying-microservices/>
- [90] SOA (Service-Oriented Architecture). IBM Cloud Education [online]. Armonk, New York: IBM Cloud Education, 2019, 17.7.2019 [cit. 2020-02-29]. Dostupné z: <https://www.ibm.com/cloud/learn/soa>
- [91] SOMMERVILLE, Ian. *Software engineering*. 9th ed. Boston: Pearson, c2011. ISBN 978-0-13-703515-1.

- [92] *Swagger.io* [online]. Somerville, MA: Swagger, 2019 [cit. 2020-03-24]. Dostupné z: <https://swagger.io/docs/specification/2-0/what-is-swagger/>
- [93] SZALVAY, Victor. *An Introduction to Agile Software Development*. Danube.com [online]. Bellevue, WA: Danube Technologies, 2004, 5.11.2004 [cit. 2020-02-29]. Dostupné z: <http://danube.com/blog/>
- [94] TELMO, Subira Rodriguez. *Understanding Event-Driven Architectures (EDA): the paradigm of the future*. In: Medium [online]. 2019 [cit. 2019-11-5]. Dostupné z: <https://medium.com/drill/understanding-event-driven-architectures-eda-the-paradigm-of-the-future-7ae632f056bb>
- [95] TENBERG, Jan. *The Long History of Microservices*. Infoq.com [online]. infoq.com: C4Media, 2016 [cit. 2020-03-24]. Dostupné z: <https://www.infoq.com/news/2016/11/microservices-history/>
- [96] The Open Group [online]. Berkshire, UK: The Open Group, 2015 [cit. 2020-02-29]. Dostupné z: <https://collaboration.opengroup.org/projects/soa/>
- [97] *Top 10 Trending Technologies To Master In 2020* [online]. www.edureka.co: www.edureka.co, 2019 [cit. 2020-03-24]. Dostupné z: <https://www.edureka.co/blog/top-10-trending-technologies/>
- [98] TULLI, Samarpit. *Microservices vs SOA: What's the Difference?* Dzone [online]. dzone.com: dzone, 2018, 16.5.2018 [cit. 2020-04-26]. Dostupné z: <https://dzone.com/articles/microservices-vs-soa-whats-the-difference>
- [99] UL HAQ, Siraj. *Introduction to Monolithic Architecture and MicroServices Architecture*. Medium.com [online]. Internet: medium.com, 2018, 2.5.2018 [cit. 2020-02-29]. Dostupné z: <https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63>
- [100] VALLOTTI, Andrea. *Event Sourcing and CQRS in C#*. Andreavallotti [online]. Italy: andreavallotti.tech, 2013, 8.1.2018 [cit. 2020-03-24]. Dostupné z: <http://www.andreavallotti.tech/en/2018/01/event-sourcing-and-cqrs-in-c/>

- [101] VANDERVAART, Dan. *Design Trend of the Month: Single Page Websites*. Moveableonline.com [online]. moveableonline.com: moveableonline.com, 2015, 1.6.2015 [cit. 2020-04-29]. Dostupné z: <http://moveableonline.com/blog/2015/06/01/design-trend-of-the-month-single-page-websites/>
- [102] VOHRA, Deepak. *Kubernetes Microservices with Docker*. 2017. White Rock, British Columbia Canada: Apress, 2017. ISBN 978-1-4842-1907-2.
- [103] *What can RabbitMQ do for you?* [online]. Unterföhring, Germany: rabbitmq.com, 2007 [cit. 2020-03-24]. Dostupné z: <https://www.rabbitmq.com/features.html>
- [104] *What is a Sprint in Scrum?* Visual-paradigm [online]. visual-paradigm.com: visual-paradigm, 2018 [cit. 2020-04-26]. Dostupné z: <https://www.visual-paradigm.com/scrum/what-is-sprint-in-scrum/>
- [105] *What Is AngularJS?* [online]. docs.angularjs: angularjs, 2017 [cit. 2020-04-29]. Dostupné z: <https://docs.angularjs.org/guide/introduction> TechStacks [online]. TechStacks: TechStacks, 2019 [cit. 2020-03-24]. Dostupné z: <https://techstacks.io/top/>
- [106] *What is Docker Container? – Containerize Your Application Using Docker* [online]. www.edureka.co: www.edureka.co, 2019 [cit. 2020-03-24]. Dostupné z: <https://www.edureka.co/blog/top-10-trending-technologies/>
- [107] *What is event-driven architecture?* Redhat [online]. Redhat.com: Redhat.com, 2017, 2017 [cit. 2020-04-26]. Dostupné z: <https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture>
- [108] *What Is Extreme Programming? An Overview of XP Rules and Values*. Lucidchart.com [online]. lucidchart.com: lucidchart.com, 2018, 30.5.2018 [cit. 2020-04-26]. Dostupné z: <https://www.lucidchart.com/blog/what-is-extreme-programming>
- [109] *What Is Service-Oriented Architecture?* Medium [online]. medium.com: Medium, 2019, 13.1.2019 [cit. 2020-04-26]. Dostupné z:

<https://medium.com/@SoftwareDevelopmentCommunity/what-is-service-oriented-architecture-fa894d11a7ec>

- [110] WRIGHT, Eric. *A Brief History of Microservices*. Turbonomic.com [online]. Boston, MA: turbonomic, 2015, 24.8.2015 [cit. 2020-03-24]. Dostupné z: <https://blog.turbonomic.com/blog/on-technology/a-brief-history-of-microservices>
- [111] YOUNG, David. *Software Development Methodologies*. Researchgate [online]. Berlin, Germany: Researchgate, 2013, 1.8.2013 [cit. 2020-02-29]. Dostupné z: [https://www.researchgate.net/publication/255710396\\_Software\\_Development\\_Methodologies](https://www.researchgate.net/publication/255710396_Software_Development_Methodologies)

## **Seznam příloh**

PŘÍLOHA A	SCRUM – PRODUKČNÍ BACKLOG.....	136
PŘÍLOHA B	SCRUM - SPRINTY .....	147
PŘÍLOHA C	PRODUKČNÍ ROAD MAP .....	150
PŘÍLOHA D	REALIZACE SPRINTŮ 1-3.....	151
PŘÍLOHA E	ZÁKLADNÍ SERVIS.....	152
PŘÍLOHA F	SERVIS – UŽIVATEL API .....	155
PŘÍLOHA G	STRUKTURA PROJEKTU .....	164
PŘÍLOHA H	PŘÍKAZY A UDÁLOSTI, PUBLIKACE A ODBĚR .....	165
PŘÍLOHA I	SERVIS – GATEWAY.....	169
PŘÍLOHA J	SERVIS – MONITOR .....	172
PŘÍLOHA K	SERVIS – EVENTSTORE.....	176
PŘÍLOHA L	SERVIS KALENDÁŘ.....	179

## **Seznam tabulek**

Tabulka 1 Porovnání metodik vývoje software.....	25
Tabulka 2 Porovnání SOA a MSA.....	88
Tabulka 3 Struktura Scrum týmu .....	94
Tabulka 4 Scrum - priorita cílů .....	96
Tabulka 5 Scrum - Škála relativní náročnosti .....	96
Tabulka 6 Scrum - rychlosť dosažení cílů .....	97
Tabulka 7 Záznam produkčního backlogu.....	97
Tabulka 8 Backlog Story .....	97
Tabulka 9 Scrum - Produkční backlog .....	137
Tabulka 10 Produkční backlog – vyhodnocení.....	138
Tabulka 11 Produkční backlog - story .....	146
Tabulka 12 Plán sprintů.....	147
Tabulka 13 Sprint 1.....	147

Tabulka 14 Sprint 2.....	148
Tabulka 15 Sprint 3.....	149
Tabulka 16 Realizace sprintů 1-3 .....	151

## **Seznam obrázků**

Obrázek 1 Vodopádový model Zdroj: [40], vlastní úprava .....	13
Obrázek 2 Spirálový model Zdroj: [91], vlastní úprava .....	16
Obrázek 3 Schéma RUP [51], vlastní úprava .....	18
Obrázek 4 Schéma Scrum[100], vlastní úprava .....	21
Obrázek 5 Scrum – úkoly Zdroj: vlastní tvorba .....	22
Obrázek 6 Schéma XP Zdroj:[108], vlastní úprava .....	23
Obrázek 7 Monolitická architektura Zdroj: vlastní tvorba .....	26
Obrázek 8 MA na SOA, Zdroj: vlastní tvorba.....	31
Obrázek 9 Struktura MA, SOA, Zdroj: vlastní tvorba .....	32
Obrázek 10 Komunikace Servis-Gateway [46] , Zdroj: vlastní úprava .....	37
Obrázek 11 Porovnání MA a SOA, Zdroj: vlastní tvorba .....	41
Obrázek 12 Struktura MA, SOA a MSA, Zdroj: vlastní tvorba .....	44
Obrázek 13 Schéma doménově řízeného návrhu, Zdroj: vlastní tvorba .....	46
Obrázek 14 Orchestrace a Choreografie Zdroj: [87], vlastní úprava .....	55
Obrázek 15 Schéma Message broker Zdroj: vlastní tvorba .....	61
Obrázek 16 Schéma Backend For Frontend (BFF) Zdroj: [58], vlastní úprava .....	63
Obrázek 17 Výhody a nevýhody MA, SOA, MSA Zdroj: vlastní tvorba .....	70
Obrázek 18 Kontejnerizace a Virtuální stroje Zdroj: [21], vlastní úprava .....	73
Obrázek 19 Schéma Kubernetes Zdroj: [25] .....	75
Obrázek 20 Schéma Api Gateway Ocelot Zdroj: [33], vlastní úprava .....	76
Obrázek 21 Struktura MB – Kafka Zdroj: [30] .....	79
Obrázek 22 Struktura MB – RabbitMQ Zdroj: [103] .....	80
Obrázek 23 Komplexita vývoje MSA a MA Zdroj: [24], vlastní úprava .....	90
Obrázek 24 Doménově řízený návrh Zdroj: vlastní tvorba .....	95
Obrázek 25 Produkční Roadmap – ukázka Zdroj: vlastní tvorba .....	98
Obrázek 26 Stav nastavení Hyper-V Zdroj: vlastní tvorba .....	100

Obrázek 27 Nastavení zdrojů Docker desktop	Zdroj: vlastní tvorba .....	100
Obrázek 28 Visual Studio podpora Docker desktop	Zdroj: vlastní tvorba .....	101
Obrázek 29 Výběr typu servisu	Zdroj: vlastní tvorba .....	102
Obrázek 30 Založení servisu	Zdroj: vlastní tvorba .....	102
Obrázek 31 Nastavení šablony a vlastností servisu	Zdroj: vlastní tvorba .....	103
Obrázek 32 Struktura servisu	Zdroj: vlastní tvorba .....	104
Obrázek 33 Uživatelské rozhraní Postman	.....	105
Obrázek 34 Visual Studio a docker-compose	Zdroj: vlastní tvorba .....	106
Obrázek 35 Nastavení docker-compose.yml	Zdroj: vlastní tvorba .....	106
Obrázek 36 Prvky a struktura systému v MSA	Zdroj: vlastní tvorba .....	108
Obrázek 37 Komunikace mikroservisu	Zdroj: vlastní tvorba .....	109
Obrázek 38 Technologie ve struktuře MSA	Zdroj: vlastní tvorba .....	110
Obrázek 39 Složení knihovny událostí	Zdroj: vlastní tvorba .....	111
Obrázek 40 Událostní řízení	Zdroj: vlastní tvorba .....	112
Obrázek 41 Komunikace a událostní konzistence	Zdroj: vlastní tvorba .....	113
Obrázek 42 Produkční Road Map	Zdroj: vlastní tvorba .....	150
Obrázek 43 Struktura projektu	.....	164

## Příloha A SCRUM – Produkční Backlog

Produkční BackLog							
Oblast	ID	P	Název	Náročnost	Body	Rychlosť	Ref ID
Pers.	1	3	Schválení ÚPD osoby	3	9	0	8
Pers.	2	3	Fond PD osoby	4	12	-1	
Pers.	3	3	Fond PD skupiny	4	12	-1	
Pers.	4	4	Schválení úpravy I/O	3	12	1	6
Pers.	5	5	Záznam I/O osoby	3	15	2	
Pers.	6	5	Přehled I/O osoby	3	15	2	5
Pers.	7	5	Přehled I/O org. Součásti	3	15	2	5
Pers.	8	3	Nastavení ÚPD osoby	5	15	-2	10
Pers.	9	3	Kontrola přesčasů	5	15	-2	10
Pers.	10	4	Nastavení PD osoby	4	16	0	
Pers.	11	4	Nastavení PD skupiny	4	16	0	10
Pers.	12	4	Úprava záznamu I/O	4	16	0	5
Pers.	13	2	Statistika I/O	8	16	-6	9
Pers.	14	4	Schválení PD osoby	6	24	-2	10
Pers.	15	5	Napojení na ext. Systémy	8	40	-3	
Mng.	16	3	Schválení Aktivity osoby	3	9	0	17
Mng.	17	5	Zapsání aktivity osoby	2	10	3	
Mng.	18	5	Editace Aktivity osoby	2	10	3	17
Mng.	19	5	Číselník Aktivit	2	10	3	
Mng.	20	4	Přehled aktivity skupiny	3	12	1	17
Mng.	21	5	Přehled Aktivity osoby	4	20	1	2
Mng.	22	4	Automatické Aktivity	5	20	-1	17
Mng.	23	5	Zadání úkolu	2	10	3	
Mng.	24	5	Editace úkolu	2	10	3	23
Mng.	25	5	Přehled úkolů	2	10	3	23
Mng.	26	5	Zadání činnosti	2	10	3	23
Mng.	27	5	Editaci činnosti	2	10	3	23
Mng.	28	3	Generování aktivity	4	12	-1	26
Mng.	29	4	Přiřazení osoby	4	16	0	23
Mng.	30	4	Přiřazení součástí	4	16	0	23
Mng.	31	4	Přidělení činnosti	4	16	0	23
Mng.	32	3	Statistika součásti	6	18	-3	25
Mng.	33	3	Statistika osoby	6	18	-3	25
Mng.	34	3	Statistika úkolu	7	21	-4	25
IT	35	3	Mailsender	4	12	-1	
IT	36	5	Monitoring	5	25	0	
IT	37	4	Integrace uživ. Práv	8	32	-4	38
IT	38	5	Import org. Struktury	8	40	-3	47
IT	39	5	Řízení přístupu osob	9	45	-4	37

IT	40	1	Admin Aplikace	9	9	-8	37
IT	41	2	Moduly	6	12	-4	46
Mng.	42	4	Kalendář den	5	20	-1	
Mng.	43	5	Kalendář úkolů	4	20	1	23
Mng.	44	5	Kalendář rok	4	20	1	
Mng.	45	5	Kalendář měsíc	4	20	1	
IT	46	4	Web SPA	8	32	-4	48
IT	47	4	Organizační struktura	8	32	-4	
IT	48	5	Web standard	6	30	-1	
Infr.	49	4	Popis rozhraní	1	4	3	
Infr.	50	5	Event store	3	15	2	52
Infr.	51	5	Message Broker	3	15	2	
Infr.	52	5	Návrh Eventů	3	15	2	
Infr.	53	3	Snapshoting	6	18	-3	50
Infr.	54	4	Obnovení systému	6	24	-2	50
Infr.	55	5	Gateway	4	20	1	
Infr.	56	5	Testování	10	50	-5	
Fin.	57	1	Přidělení bonusů	5	5	-4	60
Fin.	58	3	Schválení uzávěrky	3	9	0	63
Fin.	59	5	Přehled osoby	3	15	2	
Fin.	60	5	Přehled součásti	3	15	2	
Fin.	61	4	Historický záznam	7	28	-3	59
Fin.	62	4	Uzávěrka podkladů	8	32	-4	59
Fin.	63	5	Nastavení správce	8	40	-3	59
Fin.	64	5	Export dat	9	45	-4	61

Tabulka 9 Scrum - Produkční backlog

Produkční BackLog - Hodnocení					
Popisky řádků	POČET Cíle	SUM Body	AVG Body	Avg Náročnost	Avg Rychlosť
<b>Fin.</b>	<b>8</b>	<b>189</b>	<b>23,6</b>	<b>5,8</b>	<b>-1,8</b>
0	1	9	9,0	3,0	0,0
2	2	30	15,0	3,0	2,0
-3	2	68	34,0	7,5	-3,0
-4	3	82	27,3	7,3	-4,0
<b>Infr.</b>	<b>8</b>	<b>161</b>	<b>20,1</b>	<b>4,5</b>	<b>0,0</b>
1	1	20	20,0	4,0	1,0
2	3	45	15,0	3,0	2,0
-2	1	24	24,0	6,0	-2,0
-3	1	18	18,0	6,0	-3,0
3	1	4	4,0	1,0	3,0
-5	1	50	50,0	10,0	-5,0
<b>IT</b>	<b>14</b>	<b>349</b>	<b>24,9</b>	<b>6,3</b>	<b>-2,2</b>
0	1	25	25,0	5,0	0,0
-1	3	62	20,7	5,0	-1,0
1	3	60	20,0	4,0	1,0
-3	1	40	40,0	8,0	-3,0
-4	5	153	30,6	7,8	-4,0
-8	1	9	9,0	9,0	-8,0
<b>Mng.</b>	<b>19</b>	<b>258</b>	<b>13,6</b>	<b>3,5</b>	<b>0,7</b>
0	4	57	14,3	3,8	0,0
-1	2	32	16,0	4,5	-1,0
1	2	32	16,0	3,5	1,0
-3	2	36	18,0	6,0	-3,0
3	8	80	10,0	2,0	3,0
-4	1	21	21,0	7,0	-4,0
<b>Pers.</b>	<b>15</b>	<b>248</b>	<b>16,5</b>	<b>4,5</b>	<b>-0,7</b>
0	4	57	14,3	3,8	0,0
-1	2	24	12,0	4,0	-1,0
1	1	12	12,0	3,0	1,0
2	3	45	15,0	3,0	2,0
-2	3	54	18,0	5,3	-2,0
-6	1	16	16,0	8,0	-6,0
-3	1	40	40,0	8,0	-3,0
<b>Celkem</b>	<b>64</b>	<b>1205</b>	<b>18,8</b>	<b>4,7</b>	<b>-0,6</b>

Tabuľka 10 Produkčný backlog – vyhodnocení

Produkční Story							
Oblast	ID	P	Název	Ref	Kdo	Co	Proč
Fin.	57	1	Přidělení bonusů	60	Ved.	Správa přidělení bonusů. Historický přehled proběhlých bonusů. Možnost rozdělit celkový nastavený objem mezi podřízené celky.	Správa finančních prostředků, motivace zaměstnanců.
Fin.	58	3	Schválení uzávěrky	63	Ved.	Schválení vytvořené uzávěrky aktivit vedoucím pracovníkem.	Řízení finančních prostředků na personální zdroje.
Fin.	59	5	Přehled osoby		Zam.	Přehled vytvořených podkladů pro mzdy u každého zaměstnance. Vybrané aktivity, které mají vliv na tvorbu mzdy.	Řízení uzávěrky a příprava podkladů.
Fin.	60	5	Přehled součásti		Ved.	Přehled vytvořených podkladů zaměstnanců za podřízené součásti.	Řízení uzávěrky a příprava podkladů.
Fin.	61	4	Historický záznam	59	PZam.	Možnost načtení historického záznamu pouze za konkrétní období a zjištění stavu v té době. Indikace pozdějších úprav.	Zpětná tvorba opravné uzávěrky.
Fin.	62	4	Uzávěrka podkladů	59	PZam.	Možnost vyvolání uzávěrky pro zápis a práci s aktivitami pro zaměstnance pro mzdové účely.	Příprava podkladů pro export.
Fin.	63	5	Nastavení správce	59	PZam.	Rozhraní pro nastavení defaultních hodnot pracovní doby, aktivit a pravidel pro vytváření mzdy. Možnost manipulovat s uzávěrkami aktivit a procházet historická data	Globální správa a udržení pravidel pro výpočet mzdy.
Fin.	64	5	Export dat	59	PZam.	Export vytvořených podkladů z uzávěrky do systému pro zpracování a odeslání platebních příkazů na mzdy.	Provedení plateb, využití provozovaného řešení.

Infr.	49	4	Popis rozhraní		App	Mechanismus popisu API všech konzumovaných servis ve strojově čitelném a uživatelsky čitelném rozhraní.	Podpora více aplikací a spolupráce vývojových týmů.
Infr.	50	5	Event store	52	App	Vytvoření úložiště vydaných událostí jako centrální místo o udržení stavu všech entit v aplikaci	Udržení událostní konsistence systému.
Infr.	51	5	Message Broker		App	Provoz poskytovatele a zpracovatele zpráv o událostech proběhlých v systému, zajištění komunikace mezi servisy.	Zajištění komunikace v rámci systému.
Infr.	52	5	Návrh Eventů		App	Vytvoření struktury zpráv o událostech proběhlých nad všemi zpracovávanými entitami v systému	Standardizovaný způsob komunikace a reakce na události
Infr.	53	3	Snapshoting	52	App	Provedení snapshotu eventstore a migrace historických dat do externího úložiště. Mechanismus zpětného obnovení a přidání historických dat zpět.	Efektivita a rychlosť systému
Infr.	54	4	Obnovení systému	61	App	Mechanismus obnovení systému v případě nesrovnalosti stavu entity nebo ztráty databáze některého servisu.	Zajištění důvěrnosti a integrity systému.
Infr.	55	5	Gateway		App	Vytvoření Api gateway pro klientské aplikace a jednotný přístup na api u všech servisů v backendu.	Jednotné rozhraní volání, vystavení informace o API rozhraní servis, možnosti řízení aplikace.
Infr.	56	5	Testování		App	Nastavení mechanismů automatického testování všech funkcionalit provozovaných servis.	Podpora automatického nasazení a rychlosti reakce na závady nebo změny.

IT	35	3	Mailsender		App	Napojení na existující službu mailsenderu a odesílání notifikací u vybraných událostí oprávněným uživatelům.	Zajištění informovanosti, rychlosť reakce na události.
IT	36	5	Monitoring	35	App	Vytvoření mechanizmů pro kontrolu stavu systému.	Zajištění stabilního provozu systému.
IT	37	4	Integrace uživ. Práv		App	Import struktury uživatelských oprávnění z externího zdroje.	Využití oprávnění vytvořených v rámci intranetu organizace.
IT	38	5	Import org. Struktury		App	Import organizační struktury z externího zdroje.	Využití definované organizační struktury a reakce na její změny.
IT	39	5	Řízení přístupu osob	35	App	Nastavení prostředí pro řízení uživatelských přístupů k datům v rámci aplikace na základě údajů získaných z active direktory.	Podpora bezpečnosti, Zajištění autenticity.
IT	40	1	Admin Aplikace		PZam.	Administrátorská aplikace pro pověřené zaměstnance a vedoucí pracovníku umožňující využívat pouze některé funkce z prostředí tlustého klienta.	Ulehčení řídících činností.
IT	41	2	Moduly	17	App	Vytvoření modulů a aplikací využitelných na různých místech aplikace a v jiných aplikacích intranetového systému.	Usnadnění vybraných činností a aktivit.
IT	46	4	Web SPA		App	Single page aplikace v AngularJS pro potřeby konzumace backendu a využití vlastností MSA. Možnost využití modularizaci do dalších částí.	Využití služeb, modularizace, přehlednost

IT	47	4	Organizační struktura		Ved.	Provádět úpravy nad rámec importované organizační struktury a organizovat v přehledu vlastní úkolová uskupení nebo projektové týmy. Možnost výběru ze všech členů organizace.	Projektové řízení.
IT	48	5	Web standard		App	Standardní webová aplikace v .NET core poskytující možnost konzumovat služby backendu.	Využití služeb.
Mng.	16	3	Schválení Aktivity osoby	17	Ved.	Schválení vybrané skupiny aktivit z číselníku pro zaměstnance z podřízených celků.	Řízení personálních zdrojů.
Mng.	17	5	Zapsání aktivity osoby	20	Zam.	Zapsání aktivity zaměstnance z definovaného číselníku. Necelodenní, celodenní aktivity.	Plánování personálních zdrojů a aktivit.
Mng.	18	5	Editace Aktivity osoby	17	Zam.	Editace zadaných aktivit zaměstnance.	Zpětná úprava pro potřeby doplnění informací.
Mng.	19	5	Číselník Aktivit		App	Vytvořený editovatelný číselník aktivit pro potřeby standardizovaného zadávání událostí. Indikace aktivit, které je nutné schválit vedoucím.	Pro nastavení schvalovacích procesů a jednotného přístupu.
Mng.	20	4	Přehled aktivity skupiny	17	Ved.	Přehled v tabulce o zadaných aktivitách podřízených celků.	Řízení personálních zdrojů.
Mng.	21	5	Přehled Aktivity osoby	2	Zam.	Přehled vlastních aktivit formou tabulky.	Zpětná vazba zaměstnance, další plánování aktivit.
Mng.	22	4	Automatické Aktivity	17	Zam.	Zápis automatické aktivit v závislosti na dalších událostech probíhajících v systému a propsaných automaticky do přehledu.	Ušetření času zaměstnance při vykonávání činnosti.

Mng.	23	5	Zadání úkolu		Ved.	Vytvoření úkolu a zadání potřebných informací ohraňující projekt.	Projektové řízení.
Mng.	24	5	Editace úkolu	23	Ved.	Zpětná úprava úkolu pro potřeby doplnění informací	Doplňení informací v reakci na situaci.
Mng.	25	5	Přehled úkolů	23	Ved.	Přehled zadaných, probíhajících a splněných nebo stornovaných úkolů pro vedoucího pracovníka. Zaměstnanec uvidí úkoly, na kterých se podílel.	Projektové řízení.
Mng.	26	5	Zadání činnosti	23	Zam.	Zadání vlastní činnosti zaměstnance na přiděleném úkolu. Popis aktivity a stráveného času. Možnost propsání do přehledu Aktivit.	Doklad o plnění aktivit a postupu splnění projektu / úkolu. Zpětná vazba, Dokumentace best practice.
Mng.	27	5	Editaci činnosti	23	Zam.	Úprava údajů u zadané činnosti zaměstnance.	Dodatečné doplnění informací a provedení zpětné vazby
Mng.	28	3	Generování aktivity	26	Zam.	Generování Aktivity do přehledu aktivit zaměstnance na základě probíhající činnosti na projektu.	Podpora činnosti zaměstnanců.
Mng.	29	4	Přiřazení osoby	23	Ved.	Přiřazení zaměstnance vedoucím pracovníkem k vytvořenému úkolu.	Správa personálních zdrojů a projektové řízení.
Mng.	30	4	Přiřazení součástí	23	Ved.	Přiřazení součásti (skupiny zaměstnanců) k vytvořenému úkolu.	Správa personálních zdrojů a projektové řízení.
Mng.	31	4	Přidělení činnosti	23	Ved.	Přidělení a nařízení činnosti zaměstnanci vedoucím pracovníkem. Indikace přijmutí a splnění úkolu.	Řízení personálních zdrojů.
Mng.	32	3	Statistika součásti	25	Ved.	Přehled činnosti na úkolech za podřízené celky. Soupis jejich vytíženosti a činností.	Efektivita činnosti a míra vytíženosti podřízené současti, Projektové řízení.

Mng.	33	3	Statistika osoby	25	Ved.	Přehled informací o plněných úkolech, činnostech a aktivitách za jednu osobu.	Efektivita činnosti a míra vytíženosti zaměstnance, Projektové řízení.
Mng.	34	3	Statistika úkolu	25	Ved.	Přehled informaci plněném úkolu nebo úkolech. Přehled postupu prací.	Přehled o využití personálních zdrojů za podřízené součásti a efektivity plnění úkolu.
Mng.	42	4	Kalendář den		Zam.	Přehled zadaných aktivit, činností a úkolů pro konkrétně vybraný den. Vytvoření denního rozvrhu.	Přehlednost pracovních aktivit, Plánování.
Mng.	43	5	Kalendář úkolů		Zam.	Možnost využít plánovač úkolů a činností v kalendáři nebo jiném přehledném rozhraní	Přehlednost pracovních aktivit, Plánování
Mng.	44	5	Kalendář rok		Zam.	Přehled zadaných aktivit, činností a úkolů za kalendářní rok v přehledném kalendáři	Přehlednost pracovních aktivit, Plánování
Mng.	45	5	Kalendář měsíc		Zam.	Přehled zadaných aktivit, činností a úkolů za měsíc v přehledném kalendáři. Možnost tvorby rozvrhu na konkrétní měsíc po týdnech.	Přehlednost pracovních aktivit, Plánování.
Pers.	1	3	Schválení ÚPD osoby	8	Ved.	Schválení <b>dočasné úpravy pracovní doby</b> zaměstnance oproti standardní pracovní době. Dočasná úprava pro případ služební cesty nebo jiných mimořádných událostí.	Přehled o nestandardní pracovní době. Rozhodnutí o udělení nebo neudělení výjimky.
Pers.	2	3	Fond PD osoby		Zam.	Přehled fondu pracovní doby zaměstnance za týden, měsíc a rok. Výpočet na základě nastavení organizačních pravidel.	Zpětná vazba o plnění pracovník povinností z fondu PD.
Pers.	3	3	Fond PD skupiny		Ved.	Přehled fondu pracovní doby u podřízené součásti nebo skupiny zaměstnanců pro vedoucího.	Zpětná vazba vedoucího o plnění povinností z fondu PD.

Pers.	4	4	Schválení úpravy I/O	6	Ved.	Schválení uživatelské editace editace automatického záznamu o příchodu a odchodu zaměstnance.	Řízení oprávněnosti a zdůvodnění úpravy.
Pers.	5	5	Záznam I/O osoby		Zam.	Záznam čas příchodu a odchodu zaměstnance z objektu.	Evidence a řízení zaměstnanců, výpočet Fondu PD.
Pers.	6	5	Přehled I/O osoby	5	Zam.,Ved.	Přehled času příchodu a odchodu zaměstnance z objektu.	Zpětná vazba zaměstnance.
Pers.	7	5	Přehled I/O org. Součásti	5	Ved.	Přehled o příchodu a odchodu u podřízených součástí, zaměstnanců.	Zpětná vazba Vedoucího, Řízení zaměstnanců.
Pers.	8	3	Nastavení ÚPD osoby	10	Zam.	Nastavení dočasné úpravy pracovní doby zaměstnance.	Potřeba zaměstnanců vykonávat práci v nestandardní době, Podklady pro přesčasy, Fond PD a výpočet mzdy.
Pers.	9	3	Kontrola přezčasů	10	Ved.	Přehled příchodů a odchodů s porovnáním nastavené pracovní doby.	Výpočet fondu PD, Řízení náhradního volna ze strany Vedoucího. Podklady pro výpočet mzdy.
Pers.	10	4	Nastavení PD osoby		Zam.	Nastavení základní pracovní doby zaměstnance, možnost využití defaultní hodnoty.	Plánování aktivit, Výpočet Fondu PD, Mzdové podklady.
Pers.	11	4	Nastavení PD skupiny	10	Ved.	Nastavení základní pracovní doby podřízeným celkům.	Správa zaměstnanců, Plánování činnosti podřízených celků.
Pers.	12	4	Úprava záznamu I/O	5	Zam.	Uživatelská editace automatického záznamu pracovní doby ze strany zaměstnance v případě nesrovnalostí nebo výpadku importu dat z bezp. Systému.	Zpětná úprava v případě nefunkčnosti docházkového systému nebo práci mimo objekt.
Pers.	13	2	Statistika I/O	9	Ved.	Přehled obsahující statistiku zaměstnance o jeho příchodech, odchodech a naplnění pracovní doby, přehled	Přehled o využití personálních zdrojů za podřízené součásti.

						za podřízené celky.
Pers.	14	4	Schválení PD osoby	10	Ved.	Schválení nedefaultní pracovní doby nastavené zaměstnancem.
Pers.	15	5	Napojení na ext. Systémy		App	Import údajů z provozovaného systému čteček fyzické bezpečnosti.

Tabulka 11 Produkční backlog - story

## Příloha B SCRUM - sprints

Plán sprintů						
ID	Počet týmů	Členů	Body / tým	Body	Počet Týden	Prac. Dny
1	3	5-6	30	90	2	10
2	3	5-6	30	90	2	10
3	3	5-6	35	105	2	10
4	3	5-6	40	120	2	10
5	3	5-6	45	135	2	10
6	3	5-6	50	150	2	10
7	3	5-6	55	165	2	10
8	3	5-6	60	180	2	10
9	3	5-6	60	180	2	10
<b>Celkem</b>			<b>405</b>	<b>1215</b>	<b>18</b>	<b>90</b>

Tabulka 12 Plán sprintů

Sprint 1					
Tým 1					
Doména	Id	Priorita	Název	Body	Total Body
IT	48	5	Web standard	10	30
Infr.	55	5	Gateway	20	20
Infr.	49	4	Popis rozhraní	4	4
Infr.	52	5	Návrh Eventů	15	15
				<b>Body</b>	<b>49</b>
Tým 2					
Doména	Id	Priorita	Název	Body	Total Body
Mng.	17	5	Zapsání aktivity osoby	10	10
Mng.	18	5	Editace Aktivity osoby	10	10
Mng.	19	5	Číselník Aktivit	10	10
Mng.	21	5	Přehled Aktivity osoby	20	20
				<b>Body</b>	<b>50</b>
Tým 3					
Doména	Id	Priorita	Název	Body	Total Body
Pers.	5	5	Záznam I/O osoby	15	15
Pers.	6	5	Přehled I/O osoby	15	15
Pers.	7	5	Přehled I/O org. Součásti	15	15
				<b>Body</b>	<b>45</b>
Minimum			<b>40</b>		
Plán			<b>144</b>		
Projekt			<b>144</b>		

Tabulka 13 Sprint 1

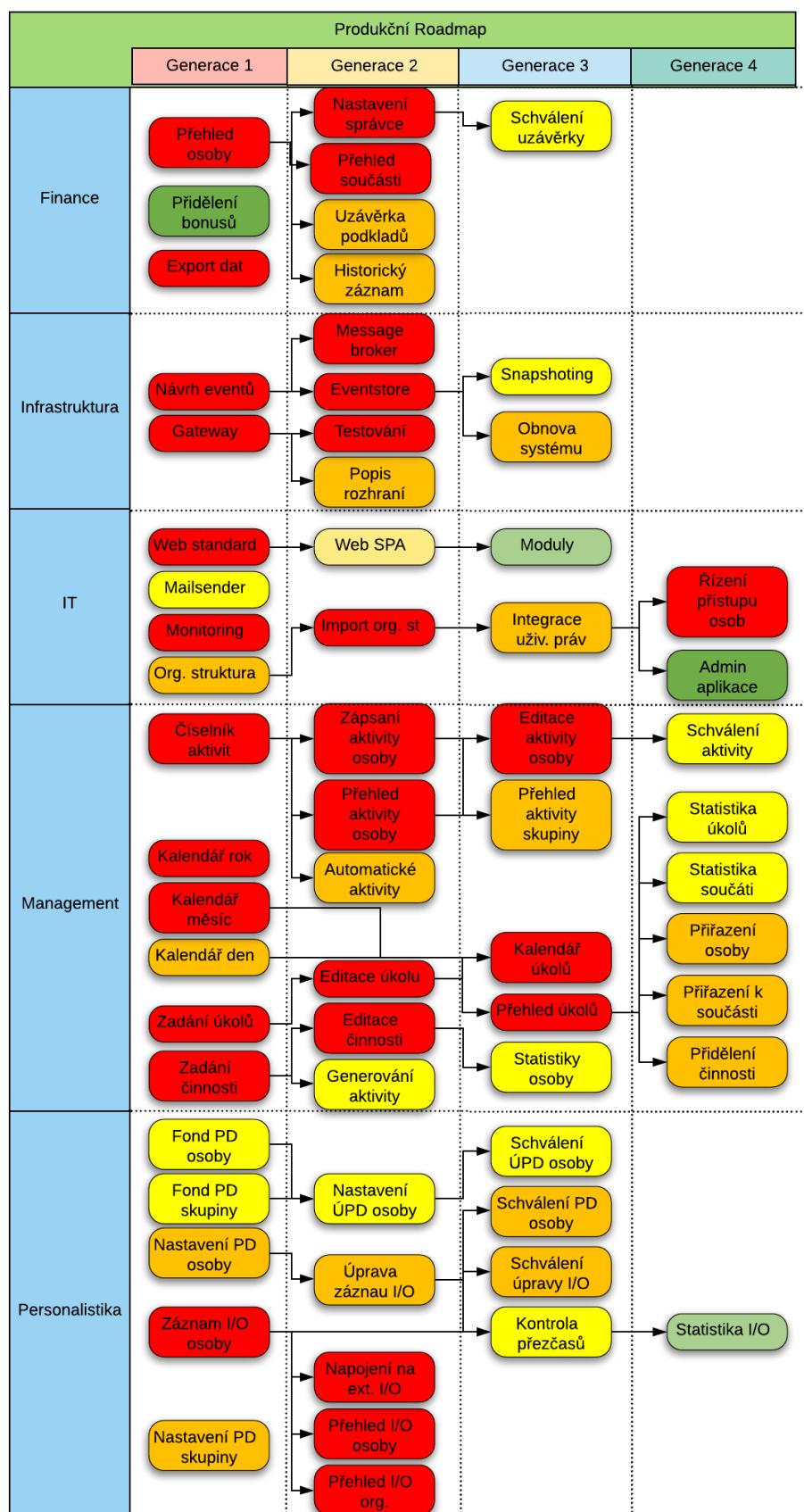
Sprint 2						
Tým 1						
Doména	Id	Priorita	Název	Body	Total Body	
IT	48	5	Web standard	10	30	
Infr.	51	5	Message Broker	15	4	
IT	36	5	Monitoring	25	25	
				Body	50	
Tým 2						
Doména	Id	Priorita	Název	Body	Total Body	
Mng.	21	5	Přehled Aktivity osoby	20	20	
Mng.	44	5	Kalendář rok	20	20	
Mng.	20	4	Přehled aktivity skupiny	12	12	
Mng.	45	5	Kalendář měsíc	10	20	
				Body	62	
Tým 3						
Doména	Id	Priorita	Název	Body	Total Body	
Pers.	7	5	Přehled I/O org. Součásti	15	15	
Pers.	12	4	Úprava záznamu I/O	12	12	
Pers.	4	4	Schválení úpravy I/O	12	12	
Pers.	10	4	Nastavení PD osoby	16	16	
				Body	55	
Minimum	<b>45</b>					
Plán	<b>167</b>					
Projekt	<b>311</b>					

Tabulka 14 Sprint 2

Tabulka č. 7 - Sprint 3					
Tým 1					
Doména	Id	Priorita	Název	Body	Total Body
Infr.	50	5	Event store	15	15
IT	38	5	Import org. Struktury	40	40
IT	46	4	Web SPA	10	32
Infr.	54	4	Obnovení systému	12	24
				Body	<b>77</b>
Tým 2					
Doména	Id	Priorita	Název	Body	Total Body
Mng.	45	5	Kalendář měsíc	10	20
Mng.	23	5	Zadání úkolu	10	10
Mng.	24	5	Editace úkolu	10	10
Mng.	25	5	Přehled úkolů	10	10
Mng.	43	5	Kalendář úkolů	20	
				Body	<b>60</b>
Tým 3					
Doména	Id	Priorita	Název	Body	Total Body
Pers.	11	4	Nastavení PD skupiny	16	16
Pers.	2	3	Fond PD osoby	12	12
Pers.	3	3	Fond PD skupiny	12	12
Fin.	59	5	Přehled osoby	15	15
				Body	<b>55</b>
Minimum					
Plán			<b>192</b>		
Projekt			<b>503</b>		

Tabulka 15 Sprint 3

## Příloha C Produkční Road Map



Obrázek 42 Produkční Road Map Zdroj: vlastní tvorba

## Příloha D Realizace sprintů 1-3

Produktový backlog - Realizace sprintu [1-3]					
ID	P	Název	Náročnost	Body	Rychlosť
50	5	Event store	3	15	2
51	5	Message Broker	3	15	2
52	5	Návrh Eventů	3	15	2
55	5	Gateway	4	20	1
36	5	Monitoring	5	25	0
48	5	Web standard	6	30	-1
38	5	Import org. Struktury	8	40	-3
17	5	Zapsání aktivity osoby	2	10	3
18	5	Editace Aktivity osoby	2	10	3
19	5	Číselník Aktivit	2	10	3
23	5	Zadání úkolu	2	10	3
24	5	Editace úkolu	2	10	3
25	5	Přehled úkolů	2	10	3
21	5	Přehled Aktivity osoby	4	20	1
43	5	Kalendář úkolů	4	20	1
44	5	Kalendář rok	4	20	1
45	5	Kalendář měsíc	4	20	1
5	5	Záznam I/O osoby	3	15	2
6	5	Přehled I/O osoby	3	15	2
7	5	Přehled I/O org. Součásti	3	15	2
49	4	Popis rozhraní	1	4	3
46	4	Web SPA	8	32	-4
20	4	Přehled aktivity skupiny	3	12	1
4	4	Schválení úpravy I/O	3	12	1
10	4	Nastavení PD osoby	4	16	0
11	4	Nastavení PD skupiny	4	16	0
12	4	Úprava záznamu I/O	4	16	0
2	3	Fond PD osoby	4	12	-1
3	3	Fond PD skupiny	4	12	-1

Tabulka 16 Realizace sprintů 1-3

## Příloha E Základní servis

### Konfigurace podpory (Dockerfile)

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim AS base
WORKDIR /app
EXPOSE 80
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build
WORKDIR /src
COPY ["Microservice.csproj", ""]
RUN dotnet restore "./Microservice.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "Microservice.csproj" -c Release -o /app/build
FROM build AS publish
RUN dotnet publish "Microservice.csproj" -c Release -o /app/publish
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Microservice.dll"]
```

### Zaváděcí procedura servisu (Startup.cs)

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddTransient< IRepository, Repository>();
        services.AddDbContext< MicroserviceDbContext >(
            opts => opts.UseSqlServer("Server=srv;Initial Catalog=MicroDb"));
        services.AddControllers();
    }
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller}/{action}/{id?}");
        });
    }
}
```

### API Kontroler (Controllers/MicroController.cs)

```
[Route("api/[controller]")]
```

```

[ApiController]
public class MicroController : ControllerBase
{
    private readonly IRepository _repository;
    public MicroController(IRepository repository)
    {
        _repository = repository;
    }
    [HttpGet]
    [Route("Get/{id?}")]
    public Micro Get(Guid id)
    {
        return _repository.Get(id);
    }
    [HttpPost]
    [Route("Add")]
    public void Add(Micro micro)
    {
        _repository.Add(micro);
    }
}

```

### Entita Micro (Entities/micro.cs)

```

public class Micro
{
    [Key]
    public Guid Id { get; set; }
    public string Value1 { get; set; }
    public int Value2 { get; set; }
}

```

### Databázový kontext (MicroserviceDbContext.cs)

```

public class MicroserviceDbContext : DbContext
{
    public MicroserviceDbContext(DbContextOptions options) : base(options)
    {
        this.Database.EnsureCreated();
    }
    public DbSet<Micro> Micros { get; set; }
}

```

### Repositář (Repository.cs)

```
public class Repository : IRepository
{
    private readonly MicroserviceDbContext db;
    public Repository(MicroserviceDbContext dbContext)
    {
        db = dbContext;
    }
    public void Add(Micro micro)
    {
        var newMicro = new Micro()
        {
            Id = Guid.NewGuid(),
            Value1 = micro.Value1,
            Value2 = micro.Value2,
        };
        db.Add(newMicro);
        db.SaveChanges();
    }
    public Micro Get(Guid id)
    {
        return db.Micros.FirstOrDefault(m => m.Id == id);
    }
}
```

## Příloha F Servis – Uživatel API

### Reference

```
☒ AspNetCore.HealthChecks (1.0.0)
☒ AspNetCore.HealthChecks.Rabbitmq (3.1.1)
☒ AspNetCore.HealthChecks.SqlServer (3.1.1)
☒ AspNetCore.HealthChecks.UI.Client (3.1.0)
☒ Microsoft.EntityFrameworkCore (3.1.3)
☒ Microsoft.EntityFrameworkCore.SqlServer (3.1.3)
☒ Microsoft.VisualStudio.Azure.Containers.Tools.Targets (1.9.10)
☒ Microsoft.VisualStudio.Web.CodeGeneration.Design (3.1.1)
☒ NSwag.AspNetCore (13.4.2)
☒ Polly (7.2.0)
☒ RabbitMQ.Client (5.1.2)
☒ Swashbuckle.AspNetCore (5.3.3)
```

### Zaváděcí procedura mikroservisu (Startup.cs)

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    private IConfiguration Configuration { get; }
    private IConnection Connection { get; set; }
    public void ConfigureServices(IServiceCollection services)
    {
        //Description: Přidání objektu obsluhy operací a databáze
        services.AddTransient< IRepository, Repository>();
        //Description: Vytvoření DbContextu a připojení do databáze.
        ConnectionString v appsettings.json
        services.AddDbContext<ServiceDbContext>(opts =>
        opts.UseSqlServer(Configuration["ConnectionString:DbConn"]));
        //Description: Generování popisu metod API. Název modulu v
        appsettings.json
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo { Title =
        Configuration["Modul:Name"], Version = "v1" });
        });
        //Description: Přidání souboru s popisem API
        services.AddSwaggerDocument();
        //Description: Vytvoření kontrolerů pro metody API
        services.AddControllers();
        //Description: Vytvoření kontroly stavu modulu a databáze a
        message brokeru
        services.AddHealthChecks()
            .AddCheck(Configuration["Modul:Name"], () =>
        HealthCheckResult.Healthy())
            .AddSqlServer(connectionString:
        Configuration["ConnectionString:DbConn"],
                    healthQuery: "SELECT 1;",
                    name: "DB",
                    failureStatus: HealthStatus.Degraded).AddRabbitMQ(sp =>
        Connection);
```

```

        //Description: Navání spojení s Message Broker
        MessageBrokerConnection(services);
    }
    public async void MessageBrokerConnection(IServiceCollection services)
    {
        //Description: Seznam zájmových Exchange ke konzumaci událostí
        var exchanges =
Configuration.GetSection("RbSetting:Subscription").Get<List<string>>();
        //Description: Nastavení připojení k MB
        var factory = new ConnectionFactory() { HostName =
Configuration["ConnectionString:RbConn"] };
        //Description: Nastavení cyklické kontroly stavu, automatického
        obnovení a intervalu
        factory.RequestedHeartbeat = 60;
        factory.AutomaticRecoveryEnabled = true;
        factory.NetworkRecoveryInterval = TimeSpan.FromSeconds(15);

        //Description: Vytvoření objektu pro Poskytnutí metod Publikace
        událostí
        //Description: Přidělení základního Exchange a fronty servisu.
        services.AddSingleton<Publisher>(s => new Publisher(factory,
Configuration["RbSetting:Exchange"], Configuration["RbSetting:Queue"]));

        //Description: Nastavení politiky reakce na výjimku při
        nedostupnosti
        //Description: v případě výjimky opakovat 5krát, každých 10 vteřin
        var retryPolicy = Policy
            .Handle<BrokerUnreachableException>()
            .WaitAndRetryAsync(5, i => TimeSpan.FromSeconds(10));
        await retryPolicy.ExecuteAsync(async () =>
{
    await Task.Run(() => {
        try
        {
            //Description: Navázání připojení s Message Broker
            Connection = factory.CreateConnection();
            //Description: Vytvoření kanálu na MB pro připojení
            servisu
            var _channel = Connection.CreateModel();
            //Description: Deklarace fronty pro servis
            var queueName = _channel.QueueDeclare().QueueName;
            //Description: Vytvoření objektu singleton Konzumaci
            naslouchání aktivity MB a Konzumace událostí
            var consumer = services.AddSingleton<ISubscriber>(s =>
new Subscriber(exchanges, Connection, _channel, queueName))

.BuildServiceProvider().GetService<ISubscriber>().Start();
            //Description: Návázání spojení na zájmové Exchange
            foreach (var ex in exchanges)
{
                _channel.QueueBind(queue: queueName,
                    exchange: ex,
                    routingKey: "");
}
            //Description: Vytvoření objektu s metodami reakce na
            konzumované události
            var listener = new
Listener(services.BuildServiceProvider().GetService< IRepository>());
            //Description: Zpracování konzumované zprávy z Message
            Broker
            consumer.Received += (model, ea) =>
{

```

```

        var body = ea.Body;
        var message = Encoding.UTF8.GetString(body);
        //Description: Předání zprávy pro nasměrování a
zpracování.
        listener.AddCommand(message);
    };
}
catch (Exception)
{
    //ToDo: Prostor pro logování výjimky a další reakce
}

});
});

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    //Description: Vystavení adresy pro kontrolu stavu
    app.UseHealthChecks("/hc");
    app.UseStaticFiles();
    //Description: Aktivace popisu API, Vystavení popisu na rozhraní
    app.UseOpenApi();
    app.UseSwaggerUi3();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json",
"${Configuration["Modul:Name"]} v1");
    });
    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();
    //Description: Vystavení výsledků kontroly stavu na rozhraní
    app.UseHealthChecks("/healthcheck", new HealthCheckOptions
    {
        Predicate = _ => true,
        ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
    });
    //Description: Předpis pro směrování příchozích požadavků na
Controller
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller}/{action}/{id?}");
    });
}
}

```

### Api Kontroler (UzivateľController.cs)

```

[ApiController]
[Route("[controller]")]
public class UzivateľController : ControllerBase
{
    private readonly IRepository _repository;

```

```

DB
    //Description: Přiřazení repositáře pro zpracování operací a připojení
    public UzivatelController(IRepository repository)
    {
        _repository = repository;

    }
    //Description: Metoda pro získání konkrétního uživatele podle ID
    [HttpGet]
    [Route("Get/{id?}")]
    public async Task<Uzivatel> Get(Guid id)
    {
        var response = await _repository.Get(id);
        return response;
    }
    //Description: Metoda získání seznamu všech uživatelů
    [HttpGet]
    [Route("GetList")]
    public async Task<List<Uzivatel>> GetList() {
        return await _repository.GetList();
    }
    //Description: Metoda přidání nového uživatele
    [HttpPost]
    [Route("Add")]
    public async Task Add(CommandUzivatelCreate cmd)
    {
        await _repository.Add(cmd);
    }
    //Description: Metoda odstranění uživatele
    [HttpDelete]
    [Route("Remove")]
    public async Task Delete(CommandUzivatelRemove cmd)
    {
        await _repository.Remove(cmd);
    }
    //Description: Metoda pro úpravu uživatele
    [HttpPost]
    [Route("Update")]
    public async Task Update(CommandUzivatelUpdate cmd)
    {
        await _repository.Update(cmd);
    }
}
}
}

```

### Definice rozhraní repositáře ( IRepository.cs )

```

public interface IRepository
{
    Task<List<Uzivatel>> GetList();
    Task<Uzivatel> Get(Guid id);
    Task Add(CommandUzivatelCreate cmd);
    Task Update(CommandUzivatelUpdate cmd);
    Task Remove(CommandUzivatelRemove cmd);
    Task LastEventCheck(Guid eventId, Guid entityId);
    Task ReplayEvents(List<string> msgstream, Guid? entityId);
    Task RequestEvents(Guid? entityId);
}

```

## Repositář (Repository.cs)

```
public class Repository : IRepository
{
    private readonly ServiceDbContext db;
    private readonly MessageHandler _handler;

    //Description: Vytvoření repositáře. Přiřazení databáze a objektu s metodami pro Publikaci
    public Repository(ServiceDbContext dbContext, Publisher publisher)
    {
        db = dbContext;
        _handler = new MessageHandler(publisher);
    }
    //Description: Ověření aktuálního stavu entity po uložení a publikaci
    public async Task LastEventCheck(Guid eventId, Guid entityId)
    {
        //Description: Nalezení záznamu
        var item = db.Uzivatele.FirstOrDefault(u => u.UzivatelId == entityId);
        if (item != null)
        {
            if (item.EventGuid != eventId)
            {
                //Description: Pokud nesouhlasí ID události, vyžádej obnovu entity
                await RequestEvents(entityId);
            }
            else {
                //Description: Potvrzení úravy entity
                item.Generation += 1;
                await db.SaveChangesAsync();
            }
        }
    }
    //Description: Metoda pro vyžádání obnovy u zájmových událostí
    public async Task RequestEvents(Guid? entityId)
    {
        var msgTypes = new List<MessageType>
        {
            MessageType.UzivatelCreated,
            MessageType.UzivatelUpdated,
            MessageType.UzivatelRemoved
        };
        //Description: Publikace požadavku na Obnovu. Určení na který exchange a kterou entitu podle ID
        await _handler.RequestReplay("uzivatel.ex", entityId, msgTypes);
    }
    //Description: Reakce na přijatou obnovovací sekvenci událostí. Může být určen jen pro entitu
    public async Task ReplayEvents(List<string> stream, Guid? entityId)
    {
        //Description: Deserializace všech zpráv z Json
        var messages = new List<Message>();
        foreach (var item in stream)
        {
            messages.Add(JsonConvert.DeserializeObject<Message>(item));
        }
        //Description: Provedení setřídění podle data vytvoření
        var replayOrderedStream = messages.OrderBy(d => d.Created);
        //Description: Postupné zpracování zpráv
    }
}
```

```

        foreach (var msg in replayOrderedStream)
    {
        //Description: Reakce na zprávy podle typu události
        switch (msg.MessageType)
        {
            case MessageType.UzivatelCreated:
                var create =
JsonConvert.DeserializeObject<EventUzivatelCreated>(msg.Event);
                var forCreate = db.Uzivatele.FirstOrDefault(u =>
u.UzivatelId == create.UzivatelId);
                if (forCreate == null)
                {
                    forCreate = Create(create);
                    db.Uzivatele.Add(forCreate);
                    db.SaveChanges();
                }
                break;
            case MessageType.UzivatelRemoved:
                var remove =
JsonConvert.DeserializeObject<EventUzivatelRemoved>(msg.Event);
                var forRemove = db.Uzivatele.FirstOrDefault(u =>
u.UzivatelId == remove.UzivatelId);
                if (forRemove != null) db.Uzivatele.Remove(forRemove);

                break;
            case MessageType.UzivatelUpdated:
                var update =
JsonConvert.DeserializeObject<EventUzivatelUpdated>(msg.Event);
                var forUpdate = db.Uzivatele.FirstOrDefault(u =>
u.UzivatelId == update.UzivatelId);
                if (forUpdate != null)
                {
                    forUpdate = Modify(update,forUpdate);
                    db.Uzivatele.Update(forUpdate);
                    db.SaveChanges();
                }
                break;
        }
        await db.SaveChangesAsync();
    }
//Description: pomocná metoda na vytvoření záznamu na základě události
private Uzivatel Create(EventUzivatelCreated evt)
{
    var model = new Uzivatel()
    {
        ImportedId = evt.ImportedId,
        UzivatelId = evt.UzivatelId,
        TitulPred = evt.TitulPred,
        Jmeno = evt.Jmeno,
        Prijmeni = evt.Prijmeni,
        TitulZa = evt.TitulZa,
        Pohlavi = evt.Pohlavi,
        DatumNarozeni = evt.DatumNarozeni,
        Email = evt.Email,
        Telefon = evt.Telefon,
        Generation = evt.Generation,
        EventGuid = evt.EventId
    };
    return model;
}
//Description: Pomocná metoda na úpravu záznamu na základě události

```

```

private Uzivatel Modify(EventUzivatelUpdated evt, Uzivatel item)
{
    item.TitulPred = evt.TitulPred;
    item.Jmeno = evt.Jmeno;
    item.Prijmeni = evt.Prijmeni;
    item.TitulZa = evt.TitulZa;
    item.Pohlavi = evt.Pohlavi;
    item.DatumNarozeni = evt.DatumNarozeni;
    item.Email = evt.Email;
    item.Telefon = evt.Telefon;
    item.EventGuid = evt.EventId;
    return item;
}
//Description: Načtení uživatele podle ID
public async Task<Uzivatel> Get(Guid id) => await Task.Run(() =>
db.Uzivatele.FirstOrDefault(b => b.UzivatelId == id));
//Description: Načtení seznamu všech uživatelů
public async Task<List<Uzivatel>> GetList() => await
db.Uzivatele.ToListAsync();
//Description: Přidání uživatele, příkaz
public async Task Add(CommandUzivatelCreate cmd)
{
    //Description: Zpracování události na základě obdrženého příkazu
    var ev = new EventUzivatelCreated()
    {
        EventId = Guid.NewGuid(),
        UzivatelId = Guid.NewGuid(),
        EventCreated = DateTime.Now,
        ImportedId = cmd.ImportedId,
        TitulPred = cmd.TitulPred,
        Jmeno = cmd.Jmeno,
        Prijmeni = cmd.Prijmeni,
        TitulZa = cmd.TitulZa,
        Pohlavi = cmd.Pohlavi,
        DatumNarozeni = cmd.DatumNarozeni,
        Email = cmd.Email,
        Telefon = cmd.Telefon,
        Generation = 0,
    };
    //Description: Vytvoření uživatele
    var item = Create(ev);
    //Description: Přidání uživatele
    db.Uzivatele.Add(item);
    //Description: Uložení uživatele
    await db.SaveChangesAsync();
    //Description: Přidání Id uživatele do události
    ev.UzivatelId = item.UzivatelId;
    //Description: Přidání generace záznamu do události, zvýšení o
stupeň
    //Description: Záznam v DB bude uveden do generace eventu po jeho
zpětné konzumaci
        ev.Generation += 1;
    //Description: Publikace události o vytvoření uživatele
        await _handler.PublishEvent(ev, MessageType.UzivatelCreated,
ev.EventId, null, ev.Generation, ev.UzivatelId);
    }
    public async Task Update(CommandUzivatelUpdate cmd)
    {
        var item = db.Uzivatele.FirstOrDefault(u => u.UzivatelId ==
cmd.UzivatelId);
        if (item != null) {
            var ev = new EventUzivatelUpdated()

```

```

        {
            EventId = Guid.NewGuid(),
            EventCreated = DateTime.Now,
            UzivatelId = item.UzivatelId,
            TitulPred = cmd.TitulPred,
            Jmeno = cmd.Jmeno,
            Prijmeni = cmd.Prijmeni,
            TitulZa = cmd.TitulZa,
            Pohlavi = cmd.Pohlavi,
            DatumNarozeni = cmd.DatumNarozeni,
            Email = cmd.Email,
            Telefon = cmd.Telefon,
        };
        item = Modify(ev, item);
        db.Uzivatele.Update(item);
        ev.Generation = item.Generation + 1;
        await _handler.PublishEvent(ev, MessageType.UzivatelUpdated,
ev.EventId, item.EventGuid, ev.Generation, item.UzivatelId);
        await db.SaveChangesAsync();
    }
}
public async Task Remove(CommandUzivatelRemove cmd)
{
    var remove = db.Uzivatele.FirstOrDefault(u => u.UzivatelId ==
cmd.UzivatelId);
    db.Uzivatele.Remove(remove);
    var ev = new EventUzivatelRemoved()
    {
        Generation = remove.Generation + 1,
        EventId = Guid.NewGuid(),
        UzivatelId = cmd.UzivatelId,
    };
    await _handler.PublishEvent(ev, MessageType.UzivatelRemoved,
ev.EventId, remove.EventGuid, remove.Generation, remove.UzivatelId);
    await db.SaveChangesAsync();
}
}
}

```

### Listener – Posluchač a směrovač zpráv

```

public class Listener
{
    private readonly IRepository _repository;
    //Description: Vytvoření posluchače a přidání repositáře
    public Listener(IRepository repository)
    {
        _repository = repository;
        //Description: Spuštění metody kontroly stavu
        CheckOnStartUp();
    }
    //Description: Kontrola stavu entit v DB
    public async void CheckOnStartUp()
    {
        //Description: Publikace o nekonzistentní stavu a vyžádání obnovy
        await _repository.RequestEvents(Guid.Empty);
    }
    //Description: Metoda zpracování konzumovaných zpráv
    public void AddCommand(string message)
    {
        //Description: Deserializace Json objektu na základní typ zprávy
        var envelope = JsonConvert.DeserializeObject<Message>(message);
    }
}

```

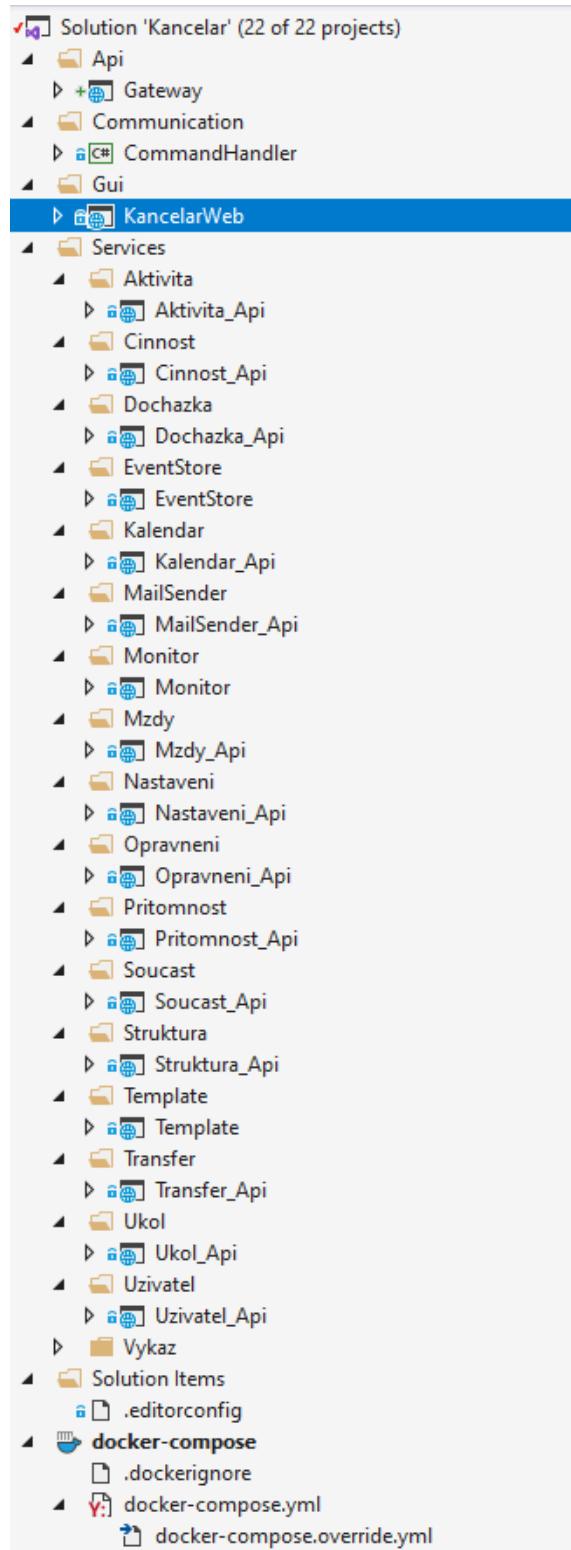
```

    //Description: Rozhodnutí o typu získazné zprávy. Typ vázaný na
Enum z knihovny
    switch (envelope.MessageType)
    {
        //Description: Indikace typu obnovovacího proudu zpráv
        case MessageType.HealingStreamProvided:
            //Description: Deserializace zprávy do správného typu a
odeslání k uložení do DB
            var ev =
JsonConvert.DeserializeObject<HealingStreamProvided>(envelope.Event);
            //Description: Předání obnovovacího proudu pro zpracování
            _repository.ReplayEvents(ev.MessageList,
envelope.EntityId);
            break;
        //Description: Reakce na vytvoření uživatele, předání ke
kontrole stavu
        case MessageType.UzivatelCreated:
            _repository.LastEventCheck(JsonConvert.DeserializeObject<EventUzivatelCreated>
(envelope.Event).EventId, envelope.EntityId);
            break;
        //Description: Reakce na upravení uživatele uživatele, předání
ke kontrole stavu
        case MessageType.UzivatelUpdated:
            _repository.LastEventCheck(JsonConvert.DeserializeObject<EventUzivatelUpdated>
(envelope.Event).EventId, envelope.EntityId);
            break;
    }
}

```

## Příloha G Struktura projektu

Struktura a rozdělení servisů v projektu Visual Studio 2019



Obrázek 43 Struktura projektu

## Příloha H Příkazy a události, Publikace a Odběr

### Události servisu Uživatel (knihovna CommandHandler)

```
public class CommandUzivatelCreate
{
    public string ImportedId { get; set; }
    public string TitulPred { get; set; }
    public string Jmeno { get; set; }
    public string Prijmeni { get; set; }
    public string TitulZa { get; set; }
    public string Pohlavi { get; set; }
    public DateTime DatumNarozeni { get; set; }
    public string Email { get; set; }
    public string Telefon { get; set; }
}
public class CommandUzivatelUpdate
{
    public Guid UzivatelId { get; set; }
    public string TitulPred { get; set; }
    public string Jmeno { get; set; }
    public string Prijmeni { get; set; }
    public string TitulZa { get; set; }
    public string Pohlavi { get; set; }
    public DateTime DatumNarozeni { get; set; }
    public string Email { get; set; }
    public string Telefon { get; set; }
}
public class CommandUzivatelRemove
{
    public Guid UzivatelId { get; set; }
}
public class EventUzivatelCreated
{
    public Guid EventId { get; set; }
    public DateTime EventCreated { get; set; }
    public int Generation { get; set; }
    public string ImportedId { get; set; }
    public Guid UzivatelId { get; set; }
    public string TitulPred { get; set; }
    public string Jmeno { get; set; }
    public string Prijmeni { get; set; }
    public string TitulZa { get; set; }
    public string Pohlavi { get; set; }
    public DateTime DatumNarozeni { get; set; }
    public string Email { get; set; }
    public string Telefon { get; set; }
}
public class EventUzivatelUpdated
{
    public Guid EventId { get; set; }
    public DateTime EventCreated { get; set; }
    public int Generation { get; set; }
    public Guid UzivatelId { get; set; }
    public string TitulPred { get; set; }
    public string Jmeno { get; set; }
    public string Prijmeni { get; set; }
    public string TitulZa { get; set; }
    public string Pohlavi { get; set; }
    public DateTime DatumNarozeni { get; set; }
}
```

```

        public string Email { get; set; }
        public string Telefon { get; set; }
    }
    public class EventUzivatelRemoved
    {
        public Guid EventId { get; set; }
        public int Generation { get; set; }
        public Guid UzivatelId { get; set; }
    }
}

```

### Příslušná část číselníku událostí (knihovna CommandHandler)

```

public enum MessageType
{
    [Description("Prikaz k obnove entity")]
    ProvideHealingStream = 1000,
    [Description("Event k obnove entity")]
    HealingStreamProvided = 1001,
    #region Uzivatel
    [Description("Command: Vytvoření nového uživatele")]
    UzivatelCreate = 12,
    [Description("Command: Odstranění uživatele")]
    UzivatelRemove = 13,
    [Description("Command: Update uživatele")]
    UzivatelUpdate = 14,
    [Description("Command: Uživatel byl vytvořen")]
    UzivatelCreated = 15,
    [Description("Command: Uživatel byl odstraněn")]
    UzivatelRemoved = 16,
    [Description("Command: Uživatel byl upraven")]
    UzivatelUpdated = 17,
    #endregion
}

```

### Publisher (knihovna CommandHandler)

```

public class Publisher : IPublisher
{
    private ConnectionFactory _factory { get; set; }
    private IConnection _connection { get; set; }
    private IModel _channel { get; set; }
    private string _exchange { get; set; }
    private string _queue { get; set; }
    //Description: Metoda pro zaslání zprávy na konkrétní Exchange
    public async Task PushToExchange(string exchange, string message)
    {
        await Task.Run(() =>
        {
            var body = Encoding.UTF8.GetBytes(message);
            _channel.BasicPublish(
                exchange: exchange,
                routingKey: "",
                basicProperties: null,
                body: body);
        });
    }
    //Description: Metoda Publikace události
    public async Task Push(string message) {
        await Task.Run(() =>

```

```

    {
        //Description: Rozložení zprávy
        var body = Encoding.UTF8.GetBytes(message);
        //Description: Zahájení publikovací pseudotransakce
        _channel.TxSelect();
        #region EventStore Exchange
        var args = new Dictionary<string, object>();
        //Description: Nastavení životnosti zprávy v Exchange
        args.Add("x-message-ttl", 432000);
        //Description: Ověření existence Exchange Eventstore a
        jeho případná deklarace
        _channel.ExchangeDeclare("eventstore.ex",
        ExchangeType.Fanout, false, false, args);
        //Description: Publikace zprávy s událostí do Eventstore
        Exchange
        _channel.BasicPublish(
            exchange: "eventstore.ex",
            routingKey: "",
            basicProperties: null,
            body: body);
        #endregion
        //Description: Publikace na exchange původce pro ověření
        _channel.BasicPublish(
            exchange: _exchange,
            routingKey: "",
            basicProperties: null,
            body: body);
        //Descriptiton: Ukončení, ověření a zpracování
        pseudotransakce
        _channel.TxCommit();
    });
}
//Description: Vytvoření připojení Publikace
public Publisher(ConnectionFactory connectionFactory, string
exchange, string queue)
{
    //Description: Parametry připojení servisy k publikaci
    this._exchange = exchange;
    this._queue = queue;
    this._factory = connectionFactory;
    this._factory.AutomaticRecoveryEnabled = true;
    this._factory.NetworkRecoveryInterval = TimeSpan.FromSeconds(5);
    try
    {
        //Description: Připojení k Message broker
        this._connection = _factory.CreateConnection();
    }
    catch (BrokerUnreachableException e)
    {
        //Description: V případě nedosažitelnosti MB, uloživ výjimku,
        počkat a připojiti znova.
        var exception = e;
        Thread.Sleep(5000);
        this._connection = _factory.CreateConnection();
    }
    this._channel = _connection.CreateModel();
    IBasicProperties props = _channel.CreateBasicProperties();
    //Description: nastavení životnosti zpráv v Exchange
    props.Expiration = "432000";
    var args = new Dictionary<string, object>();
    args.Add("x-message-ttl", 432000);
    //Description: Ověření existence exchange nebo jeho založení

```

```
        _channel.ExchangeDeclare(_exchange,
ExchangeType.Fanout,false,false,args); } }
```

### Subscriber (knihovna CommandHandler)

```
public class Subscriber : ISubscriber
{
    ConnectionFactory _factory { get; set; }
    IConnection _connection { get; set; }
    IModel _channel { get; set; }
    List<string> _exchange { get; set; }
    string _queueName { get; set; }
    //Description: Metoda zahájení a přihlášení posluchače ke konzumaci
    public EventingBasicConsumer Start()
    {
        //Description: Vytvoření / Ověření platnosti exchange
        foreach (var ex in _exchange)
        {
            var args = new Dictionary<string, object>();
            args.Add("x-message-ttl", 432000);
            _channel.ExchangeDeclare(exchange: ex, type:
ExchangeType.Fanout,false,false,args);
        }
        //Description: Nastavení kanálu pro konzumaci zpráv
        var consumer = new EventingBasicConsumer(_channel);
        //Description: Reakce na příchozí zprávu
        consumer.Received += (model, ea) =>
        {
            var body = ea.Body;
            var message = Encoding.UTF8.GetString(body);
        };
        //Description: Nastavení konzumačního kanálu na frontu servisu.
        _channel.BasicConsume(queue: _queueName,
                              autoAck: true,
                              consumer: consumer);
        return consumer;
    }
    //Description: Metoda pro zastavení konzumace
    public void Stop()
    {
        this._connection.Close();
    }
    //Description: Odběr zpráv z listu Exchange
    public Subscriber(List<string> exchange, IConnection conn, IModel
channel, string queue)
    {
        this._exchange = exchange;
        this._connection = conn;
        this._channel = channel;
        this._queueName = queue;
    }
    //Description: Odběr zpráv z konkrétního Exchange Message Brokera
    public Subscriber(string exchange, IConnection conn, IModel channel,
string queue)
    {
        var exchanges = new List<string>();
        exchanges.Add(exchange);
        this._exchange = exchanges;
        this._connection = conn;
        this._channel = channel;
        this._queueName = queue;
    }
}
```

## Příloha I Servis – Gateway

### Instalované balíčky

```
• MMlib.SwaggerForOcelot (1.10.5)
• NSwag.AspNetCore (13.2.3)
• Ocelot (14.0.11)
• Ocelot.Administration (14.0.11)
• Ocelot.Cache.CacheManager (14.0.11)
• OcelotSwagger (0.0.5)
• Swashbuckle.AspNetCore (5.0.0)
```

### Spuštění servisu

```
public class Startup
{
    private readonly IConfiguration _config;
    public Startup()
    {
        //Description: Připojení konfiguračních souborů pro aplikaci a
        modul Ocelot
        _config = new ConfigurationBuilder()
            .SetBasePath(ApplicationContext.BaseDirectory)
            .AddJsonFile("appsettings.json", false, true)
            .AddJsonFile("ocelot.json", false, true)
            .Build();
    }
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        //Description: Připojení Ocelot a jeho konfigurace
        services.AddOcelot(_config);
        //Description: Připojení podpory modulu Swagger do Ocelot
        services.AddSwaggerForOcelot(_config);
    }
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        //Description: Deserializace popisu api jednotlivých služeb do
        rozhraní Swagger v Ocelot
        app.UseSwaggerForOcelotUI(_config, opt =>
        {
            opt.DownstreamSwaggerHeaders = new[]
            {
                new KeyValuePair<string, string>("Key", "Value"),
                new KeyValuePair<string, string>("Key2", "Value2"),
            };
        }).UseOcelot().Wait();
        app.UseRouting();
        app.UseAuthorization();
    }
}
```

### Konfigurace Ocelot (Ocelot.json)

```
{  
    //Description: Konfigurace pro přesměrování  
    "ReRoutes": [  
        {  
            //Description: Vnitřní adresa služby  
            "DownstreamPathTemplate": "/uzivatel/{everything}",  
            "DownstreamScheme": "http",  
            "DownstreamHostAndPorts": [  
                {  
                    //Description: Alias z Docker-compose.yml  
                    "Host": "uzivatelapi/"  
                }  
            ],  
            //Description: Veřejná adresa pro volání služby  
            "UpstreamPathTemplate": "/api/uzivatel/{everything}",  
            //Description: Povolené metody  
            "UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE" ],  
            //Description: Alias pro konfiguraci swagger  
            "SwaggerKey": "uzivatel"  
        }  
    ],  
    //Description: Nastavení služby adres pro předávání konfigurace API  
    "SwaggerEndPoints": [  
        {  
            "Key": "uzivatel",  
            "Config": [  
                {  
                    "Name": "Uzivatel API",  
                    "Version": "v1",  
                    "Url": "http://uzivatelapi/swagger/v1/swagger.json"  
                }  
            ]  
        }  
    ]  
}
```

## Popis metod API servisů

### Uzivatel

<b>GET</b>	/api/uzivatel/Get/{id}
<b>GET</b>	/api/uzivatel/GetList
<b>POST</b>	/api/uzivatel/Add
<b>DELETE</b>	/api/uzivatel/Remove
<b>POST</b>	/api/uzivatel/Update

### Models

- Uzivatel >
- CommandUzivatelCreate >
- CommandUzivatelRemove >
- CommandUzivatelUpdate >

### Models

```
Uzivatel v {
    id*           string($guid)
    eventGuid     string($guid)
    generation*   integer($int32)
    uzivateliId*  string($guid)
    importedId    string
    titulPred     string
    jmeno          string
    prijmeni      string
    titulZa        string
    pohlavi       string
    datumNarozeni* string($date-time)
    email          string
    telefon        string
}

CommandUzivatelCreate v {
    importedId    string
    titulPred     string
    jmeno          string
    prijmeni      string
    titulZa        string
    pohlavi       string
    datumNarozeni* string($date-time)
    email          string
    telefon        string
}

CommandUzivatelRemove >

CommandUzivatelUpdate >
```

## Příloha J Servis – Monitor

### Instalované balíčky

```
ASP.NET Core Health Checks (1.0.0)
ASP.NET Core Health Checks.EventStore (3.0.0)
ASP.NET Core Health Checks.Network (3.0.1)
ASP.NET Core Health Checks.Rabbitmq (3.1.1)
ASP.NET Core Health Checks.SqlServer (3.1.1)
ASP.NET Core Health Checks.System (3.0.3)
ASP.NET Core Health Checks.UI (3.0.9)
ASP.NET Core Health Checks.UI.Client (3.1.0)
ASP.NET Core Health Checks.Uris (3.0.0)
Microsoft.EntityFrameworkCore.SqlServer (3.1.3)
Microsoft.VisualStudio.Azure.Containers.Tools.Targets (1.9.10)
NSwag.AspNetCore (13.4.2)
Polly (7.2.0)
RabbitMQ.Client (5.1.2)
Swashbuckle.AspNetCore (5.3.3)
```

### Spuštění a konfigurace servisu (Startup.cs)

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public IConfiguration Configuration { get; }
    public IConnection Connection { get; set; }
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        //Description: Nastavení stavu servisu Monitor
        services.AddHealthChecks().AddCheck("Monitor", () =>
HealthCheckResult.Healthy());
        services.AddHealthChecksUI(setupSettings: setup =>
        {
            //Description: Adresy na endpoint kontroly stavu servisů
            setup.AddHealthCheckEndpoint("Dochazka",
"http://dochazkaapi/healthcheck");
            setup.AddHealthCheckEndpoint("Uzivatel",
"http://uzivatelapi/healthcheck");
            setup.AddHealthCheckEndpoint("Eventstore",
"http://eventstore/healthcheck");
            setup.AddHealthCheckEndpoint("Kalendar",
"http://kalendarapi/healthcheck");
            setup.AddHealthCheckEndpoint("Pritomnost",
"http://pritomnostapi/healthcheck");
            setup.AddHealthCheckEndpoint("Aktivita",
"http://aktivitaapi/healthcheck");
            setup.AddHealthCheckEndpoint("Cinnost",
"http://cinnostapi/healthcheck");
            setup.AddHealthCheckEndpoint("MailSender",
"http://mailsenderapi/healthcheck");
        });
    }
}
```

```

                setup.AddHealthCheckEndpoint("Mzdy",
"http://mzdyapi/healthcheck");
                setup.AddHealthCheckEndpoint("Nastaveni",
"http://nastaveniapi/healthcheck");
                setup.AddHealthCheckEndpoint("Opravneni",
"http://opravneniapi/healthcheck");
                setup.AddHealthCheckEndpoint("Soucast",
"http://soucastapi/healthcheck");
                setup.AddHealthCheckEndpoint("Struktura",
"http://strukturaapi/healthcheck");
                setup.AddHealthCheckEndpoint("Ukol",
"http://ukolapi/healthcheck");
                setup.AddHealthCheckEndpoint("Vykaz",
"http://vykazapi/healthcheck");
                setup.AddHealthCheckEndpoint("Transfer",
"http://transferapi/healthcheck");
            });
            //Description: Kontrola stavu serveru RabbitMQ
            MessageBrokerConnection(services);
            services.AddHealthChecks().AddRabbitMQ(sp => Connection);
        }
        //Description: Připojení k RabbitMQ
        public async void MessageBrokerConnection(IServiceCollection services)
        {
            if (services is null)
            {
                throw new ArgumentNullException(nameof(services));
            }
            var factory = new ConnectionFactory() { HostName =
Configuration["ConnectionString:RbConn"] };
            factory.RequestedHeartbeat = 60;
            factory.AutomaticRecoveryEnabled = true;
            factory.NetworkRecoveryInterval = TimeSpan.FromSeconds(15);
            var retryPolicy =
Policy.Handle<BrokerUnreachableException>().WaitAndRetryAsync(5, i =>
TimeSpan.FromSeconds(10));
            await retryPolicy.ExecuteAsync(async () =>
{
            await Task.Run(() => {
                Connection = factory.CreateConnection();
            });
});
}
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseRouting();
    app.UseAuthorization();
    app.UseHealthChecks("/healthcheck", new HealthCheckOptions
{
    Predicate = _ => true,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
    app.UseHealthChecksUI();
    app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
}

```

```

        });
    }
}
```

## Konfigurace Monitoru (appsettings.json)

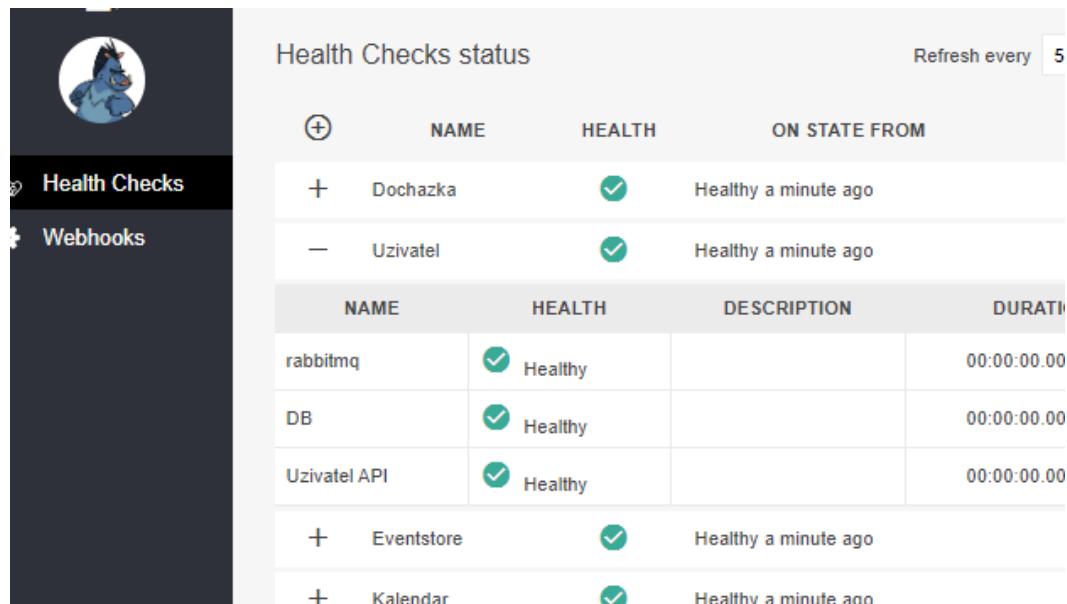
```
{
  "AllowedHosts": "*",
  "ConnectionString": {
    "DbConn": "Server=sqlServer;Initial
Catalog=Kancelar.Service.EventStore;User=sa;Password=Password123;MultipleActiv
eResultSets=true;",
    "Rabbit": "rabbitmq"
  },
  "Moduls": {
    "Services": [
      "Dochazka",
      "Uzivatel",
      "EventStore",
      "Kalendar",
      "ImportExport",
      "Aktivita",
      "Cinnost",
      "MailSender",
      "Mzdy",
      "Nastaveni",
      "Opravneni",
      "Soucast",
      "Struktura",
      "Ukol",
      "Vykaz"
    ]
  },
  "ConnectionStrings": {
    "Dochazka": [ "Server=sqlServer;Initial
Catalog=Kancelar.Service.Dochazka;User=sa;Password=Password123;MultipleActiveR
esultSets=true;" ],
    "Kalendar": [ "Server=sqlServer;Initial
Catalog=Kancelar.Service.Kalendar;User=sa;Password=Password123;MultipleActiveR
esultSets=true;" ],
    "Uzivatel": [ "Server=sqlServer;Initial
Catalog=Kancelar.Service.Uzivatele;User=sa;Password=Password123;MultipleActive
ResultSets=true;" ],
    "Eventstore": [ "Server=sqlServer;Initial
Catalog=Kancelar.EventStore;User=sa;Password=Password123;MultipleActiveResults
ets=true;" ],
    "Aktivita": [ "Server=sqlServer;Initial
Catalog=Kancelar.Aktivita;User=sa;Password=Password123;MultipleActiveResultSet
s=true;" ],
    "Cinnost": [ "Server=sqlServer;Initial
Catalog=Kancelar.Cinnost;User=sa;Password=Password123;MultipleActiveResultSets
=true;" ],
    "MailSender": [ "Server=sqlServer;Initial
Catalog=Kancelar.MailSender;User=sa;Password=Password123;MultipleActiveResults
ets=true;" ],
    "Mzdy": [ "Server=sqlServer;Initial
Catalog=Kancelar.Mzdy;User=sa;Password=Password123;MultipleActiveResultSets=tr
ue;" ],
    "Nastaveni": [ "Server=sqlServer;Initial
Catalog=Kancelar.Nastaveni;User=sa;Password=Password123;MultipleActiveResultSe
ts=true;" ],
  }
}
```

```

    "Opravneni": [ "Server	sqlServer;Initial
Catalog=Kancelar.Opravneni;User=sa;Password=Password123;MultipleActiveResultSets=true;" ],
    "Soucast": [ "Server	sqlServer;Initial
Catalog=Kancelar.Soucast;User=sa;Password=Password123;MultipleActiveResultSets=true;" ],
    "Struktura": [ "Server	sqlServer;Initial
Catalog=Kancelar.Struktura;User=sa;Password=Password123;MultipleActiveResultSets=true;" ],
    "Ukol": [ "Server	sqlServer;Initial
Catalog=Kancelar.Ukol;User=sa;Password=Password123;MultipleActiveResultSets=true;" ],
    "Vykaz": [ "Server	sqlServer;Initial
Catalog=Kancelar.Vykaz;User=sa;Password=Password123;MultipleActiveResultSets=true;" ],
    "Transfer": [ "Server	sqlServer;Initial
Catalog=Kancelar.Transfer;User=sa;Password=Password123;MultipleActiveResultSets=true;" ]
},
"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft": "Warning",
    "Microsoft.Hosting.Lifetime": "Information"
  }
}
}

```

## Uživatelské rozhraní



The screenshot shows a dashboard titled "Health Checks status". On the left, there's a sidebar with "Health Checks" and "Webhooks" sections. The main area displays a table of health checks:

NAME	HEALTH	ON STATE FROM
Dochazka	<span style="color: green;">✓</span>	Healthy a minute ago
Uzivatel	<span style="color: green;">✓</span>	Healthy a minute ago

Below this, another table lists services with their health status:

NAME	HEALTH	DESCRIPTION	DURATION
rabbitmq	<span style="color: green;">✓</span> Healthy		00:00:00.00
DB	<span style="color: green;">✓</span> Healthy		00:00:00.00
Uzivatel API	<span style="color: green;">✓</span> Healthy		00:00:00.00
Eventstore	<span style="color: green;">✓</span>	Healthy a minute ago	
Kalendar	<span style="color: green;">✓</span>	Healthy a minute ago	

## Příloha K Servis – EventStore

### Instalované balíčky

```
ASP.NET Core Health Checks
ASP.NET Core Health Checks.RabbitMQ
ASP.NET Core Health Checks.SqlServer
ASP.NET Core Health Checks.UI
ASP.NET Core Health Checks.UI.Client
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.VisualStudio.Azure.Containers.Tools.Targets
Newtonsoft.Json
NSwag.AspNetCore
Polly
RabbitMQ.Client
Swashbuckle.AspNetCore
Swashbuckle.AspNetCore.SwaggerUI
```

### Spuštění a konfigurace servisu (StartUp.cs)

Způsob startu servisu je totožný jako ostatní běžné servisy, například Uživatel. Liší se akorát v nastaveném konfiguračním souboru appsettings.json obsahující připojení do vlastní databáze.

### Posluchač a směrovač

```
public class Listener
{
    //string _BaseUrl;
    private readonly IRepository _repository;
    public Listener(IRepository repository)
    {
        _repository = repository;
    }
    public void AddCommand(string message)
    {
        var envelope = JsonConvert.DeserializeObject<Message>(message);
        switch (envelope.MessageType)
        {
            case MessageType.ProvideHealingStream:
                _repository.ProvideHealingStream(message);
                break;
            default:
                _repository.AddMessageAsync(message);
                break;
        }
    }
}
```

### Repositář

```
public class Repository : IRepository
```

```

{
    private readonly ServiceDbContext db;
    private readonly MessageHandler _handler;
    public Repository(ServiceDbContext dbContext, Publisher publisher)
    {
        db = dbContext;
        _handler = new MessageHandler(publisher);
    }
    public async Task<List<StoreMessage>> GetListByDate(DateTime datum)
    {
        return await Task.Run(() => db.Messages.Where(m => m.Created > datum).ToList());
    }
    public async Task AddMessageAsync(string msg)
    {
        //Description: Deserializace události jako obecné zprávy a uložení
do DB
        var origin = JsonConvert.DeserializeObject<Message>(msg);
        var message = new StoreMessage
        {
            Guid = Guid.NewGuid(),
            MessageType = origin.MessageType,
            MessageTypeText = origin.MessageType.ToString(),
            Created = origin.Created,
            EntityId = origin.EntityId,
            Message = msg
        };
        db.Messages.Add(message);
        await db.SaveChangesAsync();
    }
    public async Task<StoreMessage> Get(string guid)
    {
        return await Task.Run(() => db.Messages.FirstOrDefault(m => m.Guid == Guid.Parse(guid)));
    }
    //Description: Metoda poskytující proud událostí pro obnovu na základě
požadavku
    public async Task ProvideHealingStream(string message)
    {
        var envelope = JsonConvert.DeserializeObject<Message>(message);
        var ev =
JsonConvert.DeserializeObject<ProvideHealingStream>(envelope.Event);
        //Description: Vytvoření události o poskytnuté obnově
        var responseEvent = new HealingStreamProvided()
        {
            EntityId = ev.EntityId,
            MessageList = new List<string>()
        };
        //Description: Obnova pouze jedné entity
        if (ev.EntityId != Guid.Empty)
        {
            foreach (var type in ev.MessageTypes)
            {
                responseEvent.MessageList.AddRange(db.Messages.Where(m => m.MessageType == type & m.EntityId == ev.EntityId).Select(m => m.Message));
            }
        }
        //Description: Načtení všech požadovaných typů zpráv
        else
        {
            foreach (var type in ev.MessageTypes)
            {

```

```
        responseEvent.MessageList.AddRange(db.Messages.Where(m =>
m.MessageType == type).Select(m => m.Message));
    }
}
if (responseEvent.MessageList.Any()) {
    var msg = new Message();
    //Description: Publikace obnovovací události
    await _handler.PublishEventToExchange(responseEvent,
MessageType.HealingStreamProvided, Guid.NewGuid(), null, 0, (ev.EntityId ==
Guid.Empty) ? Guid.Empty : Guid.Parse(ev.EntityId.ToString()), ev.Exchange);
}
```

## Příloha L Servis Kalendář

### Instalované balíčky

```
• AspNetCore.HealthChecks (1.0.0)
• AspNetCore.HealthChecks.Rabbitmq (3.1.1)
• AspNetCore.HealthChecks.SqlServer (3.1.1)
• AspNetCore.HealthChecks.UI.Client (3.1.0)
• Microsoft.EntityFrameworkCore (3.1.3)
• Microsoft.EntityFrameworkCore.SqlServer (3.1.3)
• Microsoft.VisualStudio.Azure.Containers.Tools.Targets (1.9.10)
• NSwag.AspNetCore (13.4.2)
• Polly (7.2.0)
• Polly.Extensions.Http (3.0.0)
• RabbitMQ.Client (5.1.2)
• Swashbuckle.AspNetCore (5.3.3)
```

### Repozitář

Reakce na události publikované servisem Uživatel

```
public async Task CreateByUzivatel(EventUzivatelCreated evt)
{
    //Description: Pokus o vyhledání existujícího kalendáře
    var item = db.Kalendare.Where(k => k.UzivatelId == evt.UzivatelId
&& k.Rok == evt.EventCreated.Year).FirstOrDefault();
    //Description: Kalendář není nalezen
    if (item == null)
    {
        //Description: Vytvoření nového kalendáře
        item = await Create(evt);
        db.Kalendare.Add(item);
        await db.SaveChangesAsync();

        //Description: Publikace události o tom, že byl kalendář
vytvořen
        var ev = new EventKalendarCreated()
        {
            CeleJmeno = $"{evt.Prijmeni} {evt.Jmeno}",
            EventCreated = DateTime.Now,
            EventId = Guid.NewGuid(),
            Generation = 0,
            Rok = evt.EventCreated.Year,
            SourceGuid = evt.EventId,
            UzivatelId = evt.UzivatelId,
        };
        await _handler.PublishEvent(ev, MessageType.KalendarCreated,
ev.EventId, null, ev.Generation, item.KalendarId);
    }
}
public async Task UpdateByUzivatel(EventUzivatelUpdated evt)
{
    var kalendarList = db.Kalendare.Where(k => k.UzivatelId ==
evt.UzivatelId);
    if (kalendarList.Any())
    {
```

```

        foreach (var item in kalendarList)
        {
            var ev = new EventKalendorUpdated()
            {
                CeleJmeno = $"'{evt.Prijmeni} {evt.Jmeno}'",
                EventCreated = DateTime.Now,
                EventId = Guid.NewGuid(),
                Generation = item.Generation + 1,
                SourceGuid = evt.EventId,
                UzivatelId = evt.UzivatelId,
                Body = item.Body,
            };
            await _handler.PublishEvent(ev,
                MessageType.KalendarCreated, ev.EventId, null, ev.Generation,
                item.KalendorId);

            item.UzivatelId = ev.UzivatelId;
            item.EventGuid = ev.EventId;
            item.UzivatelCeleJmeno = ev.CeleJmeno;
            item.DatumAktualizace = DateTime.Now;
            db.Kalendare.Update(item);
        }
    }
    await db.SaveChangesAsync();
}

public async Task DeleteByUzivatel(EventUzivatelRemoved evt)
{
    var kalendarList = db.Kalendare.Where(k => k.UzivatelId == evt.UzivatelId);
    if (kalendarList.Any())
    {
        foreach (var item in kalendarList)
        {
            var ev = new EventKalendorRemoved()
            {
                EventCreated = DateTime.Now,
                EventId = Guid.NewGuid(),
                Generation = item.Generation + 1,
                SourceGuid = evt.EventId,
                UzivatelId = evt.UzivatelId,
            };
            await _handler.PublishEvent(ev,
                MessageType.KalendarRemoved, ev.EventId, null, ev.Generation,
                item.KalendorId);
            db.Kalendare.Remove(item);
        }
    }
    await db.SaveChangesAsync();
}

```

## Posluchač a směrovač

```

public class Listener
{
    //string _BaseUrl;
    private readonly IRepository _repository;

    public Listener(IRepository repository)

```

```

    {
        _repository = repository;
    }
    public void AddCommand(string message)
    {
        //Description: Deserializace zprávy
        var envelope = JsonConvert.DeserializeObject<Message>(message);
        //Description: Kontrola typu zprávy a rozhodnutí o způsobu
konzumace
        switch (envelope.MessageType)
        {
            case MessageType.HealingStreamProvided:
                var ev =
JsonConvert.DeserializeObject<HealingStreamProvided>(envelope.Event);
                    ReplayEvents(ev.MessageList, envelope.EntityId);
                    break;
            case MessageType.KalendarCreated:
                _repository.LastEventCheck(JsonConvert.DeserializeObject<EventKalendarCreated>
(envelope.Event).EventId, envelope.EntityId);
                    break;
            case MessageType.KalendarUpdated:
                _repository.LastEventCheck(JsonConvert.DeserializeObject<EventKalendarUpdated>
(envelope.Event).EventId, envelope.EntityId);
                    break;
            case MessageType.UzivatelCreated:
CreateByUzivatel(JsonConvert.DeserializeObject<EventUzivatelCreated>(envelope.
Event));
                    break;
            case MessageType.UzivatelUpdated:
UpdateByUzivatel(JsonConvert.DeserializeObject<EventUzivatelUpdated>(envelope.
Event));
                    break;
            case MessageType.UzivatelRemoved:
RemoveByUzivatel(JsonConvert.DeserializeObject<EventUzivatelRemoved>(envelope.
Event));
                    break;
            case MessageType.AktivitaCreated:
CreateByAktivita(JsonConvert.DeserializeObject<EventAktivitaCreated>(envelope.
Event));
                    break;
            case MessageType.AktivitaUpdated:
UpdateByAktivita(JsonConvert.DeserializeObject<EventAktivitaUpdated>(envelope.
Event));
                    break;
            case MessageType.AktivitaRemoved:
RemoveByAktivita(JsonConvert.DeserializeObject<EventAktivitaRemoved>(envelope.
Event));
                    break;
        }
    }
}

```

```

        }
    }
    //Description: Reakce na událost vytvoření uživatele
    private void CreateByUzivatel(EventUzivatelCreated evt) {
        _repository.CreateByUzivatel(evt);
    }
    private void UpdateByUzivatel(EventUzivatelUpdated evt)
    {
        _repository.UpdateByUzivatel(evt);
    }
    private void RemoveByUzivatel(EventUzivatelRemoved evt)
    {
        _repository.DeleteByUzivatel(evt);
    }
    private void CreateByAktivita(EventAktivitaCreated evt)
    {
        _repository.CreateByAktivita(evt);
    }
    private void UpdateByAktivita(EventAktivitaUpdated evt)
    {
        _repository.UpdateByAktivita(evt);
    }
    private void RemoveByAktivita(EventAktivitaRemoved evt)
    {
        _repository.DeleteByAktivita(evt);
    }
    private void ReplayEvents(List<string> stream, Guid? entityId)
    {
        var messages = new List<Message>();
        foreach (var item in stream)
        {
            messages.Add(JsonConvert.DeserializeObject<Message>(item));
        }
        var replayOrderedStream = messages.OrderBy(d => d.Created);
        foreach (var msg in replayOrderedStream)
        {
            AddCommand(JsonConvert.SerializeObject(msg));
        }
    }
}

}

```

## Kalendář

```

public class Kalendar
{
    [Key]
    public Guid Id { get; set; }
    public Guid KalendarId { get; set; }
    public Guid UzivatelId { get; set; }
    public string UzivatelCeleJmeno { get; set; }
    public int Rok { get; set; }
    public string Body { get; set; }
    public DateTime DatumAktualizace { get; set; }
    public virtual Year KalendarBody { get {
        return JsonConvert.DeserializeObject<Year>(this.Body);
    } }
    public Guid? EventGuid { get; set; }
    public int Generation { get; set; }
}

```

```
}
```

## Generování struktury kalendáře

```
public class Day
{
    public int Id { get; set; }
    public int TypId { get; set; }
    public string Nazev { get; set; }
    public bool IsSvatek { get; set; }
    public int AktivitaCount { get; set; }
    public List<Polozka> Polozky { get; set; }
}
public class Month {
    public int Id { get; set; }
    public List<Day> Days {get;set;}
    public int DayCount { get; set; }
    public virtual string MonthName { get { return
System.Globalization.CultureInfo.CurrentCulture.DateTimeFormat.GetMonthName(Id);
}; } }
}
public class Year
{
    public int Id { get; set; }
    public List<Month> Months { get; set; }
}
public class Polozka {
public Guid Id { get; set; }
public int AktivitaTypId { get; set; }
public string Nazev { get; set; }
public Guid UzivatelId { get; set; }
public string CeleJmeno { get; set; }
public DateTime DatumOd { get; set; }
public DateTime DatumDo { get; set; }
}
public class KalendarGenerator {
    public async Task<Year> KalendarNew()
    {
        var baseDate = DateTime.Today;
        var Rok = new Year()
        {
            Id = baseDate.Year
        };
        await Task.Run(() =>
        {

            Rok.Months = new List<Month>();
            for (int m = 1; m <= 12; m++)
            {
                var Mesic = new Month() {
                    Id = m,
                    DayCount = DateTime.DaysInMonth(baseDate.Year, m)
                };
                Mesic.Days = new List<Day>();
                for (int d = 1; d <= Mesic.DayCount; d++)
                {
                    var datum = new DateTime(Rok.Id,Mesic.Id,d);
                    var Den = new Day() {
                        Id = d,
                        IsSvatek = !string.IsNullOrEmpty(IsSvatek(datum)),
                        Nazev = IsSvatek(datum),

```

```
        TypId = (datum.DayOfWeek ==0)? 7 :  
    (int)datum.DayOfWeek,  
        AktivitaCount = 0,  
        Polozky = new List<Polozka>()  
    };  
    Mesic.Days.Add(Den);  
}  
Rok.Months.Add(Mesic);  
};  
});  
return Rok;  
}
```