FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Informační systém pro správu projektů v architektuře mikroservis
Student:	Bc. Sergey Dunaevskiy
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

- Na základě předchozí bakalářské práce (text, zdrojový kód) a posudků proveďte stručnou analýzu oblastí, jež by se mohly zlepšit v informačním systému.
- Zanalyzujte oblast architektury mikroservis a jejich využití v daném informačním systému.
- Navrhněte přepis monolitní architektury na architekturu mikroservices a implementujte.
- V rozumné míře nahraďte uživatelské testování v původním projektu automatickým.
- Navrhněte a proveďte kontejnerizaci informačního systému pro možnost plynulého nasazování a udržování na serveru.
- Výsledný systém zdokumentujte a zhodnoťte v porovnání s předchozím stavem. Navrhněte možný rozvoj sytému.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 5. října 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Informační systém pro správu projektů v architektuře mikroservis

Bc. Sergey Dunaevskiy

Katedra softwarového inženýrství

Vedoucí práce: Ing. Michal Valenta, Ph.D.

18. dubna 2022

Poděkování

Děkuji Ing. Michalu Valentovi, Ph.D. za cenné rady a pomoc při vytváření této diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 18. dubna 2022 ……

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Dunaevskiy, Sergey. *Informační systém pro správu projektů v architektuře mikroservis*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022. Dostupný také z WWW: ⟨`https://github.com/dunaevskiy/thesis-masters`⟩.

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací informačního systému v architektuře mikroslužeb na základě existující bakalářské práce. Cílem je přepsat původní architekturu, automaticky otestovat a zajistit kontejnerizaci. Na základě analýzy oblasti mikroslužeb byly vybrány a adaptovány vhodné varianty realizace pro server i klienta. Vzhledem k rozsahu práce původní funkcionalita byla částečně přenesena do budoucího rozvoje. V závěru je nové řešení porovnáno s původním a zhodnocen výsledek.

Klíčová slova architektura, mikroslužby, server, klient, javascript, nodejs, nextjs, správa projektů

Abstract

This master's thesis deals with designing and implementing an information system described in the existing bachelor's thesis in microservices architecture. The goal is to rewrite the original architecture, automatically test and ensure containerization. Based on the analysis of the area of microservices, suitable variants of the architecture were chosen and adapted. Due to the scope of the work, some functionality was partially transferred to future development. In conclusion, the new solution is compared with the previous, and the result is evaluated.

Keywords architecture, microservices, server, client, javascript, nodejs, nextjs, project management

Obsah

Seznam obrázků

Úvod

Architektura aplikací tvoří nedílnou součást každého vývoje. Její náročnost je variabilní v závislosti na typu, složitosti, frekventovanosti použití a dalších parametrech aplikace. Zpravidla čím menší nároky se na výsledek kladou, tím je navrhovaná architektura primitivnější, aby se ušetřilo na nákladech během vývoje a provozu aplikace. Komplexnější požadavky však potřebují pro svůj provoz zajistit stabilní prostředí, které se definuje například stabilitou aplikace, způsobem opravy neočekávaných chyb, rychlosti odezvy apod. Všechny tyto aspekty je mnohdy složité řešit najednou a přirozeně se rozdělují na menší atomické celky, jež by se daly spravovat samostatně.

Koncept mikroslužeb vznikl přibližně před 10 až 15 lety. Explicitně, jako pojem, byl zmíněn až v roce 2011 v rámci popisu stylu architektury, se kterou se tehdy experimentovalo [1]. Prudký růst zájmu o MSA (microservice architecture) a mikroslužby celkově (dle statistických údajů Google Trends) byl zaznamenán v listopadu 2014 a začátku roku 2017. V průběhu roku 2019 dosahoval největší popularity dle relativních počtů vyhledávání [2]. Aktuálně MSA zůstává velice frekventovaným trendem [2], z tohoto důvodu lze danou architekturu předpokládat za stále využívanou nebo alespoň jevící zájem u určitého množství lidí.

Jelikož daný koncept je poměrné mladý, tak je málo pravděpodobné, že byl plně prozkoumán. Představuje potenciální oblast pro experimentování a zdokonalování, modifikaci a odvozování jiných, nezávislých vývojových praktik. Průzkum takových skutečností se nejlépe ověřuje v praxi, v daném případě budou vhodné středně velké aplikace s teoreticky neomezenou možností rozvoje. Jako příklad takové struktury může být IS (informační systém) pro správu projektů v akademických institucích.

Cíl práce

Cílem této diplomové práce je primárně průzkum MSA se subjektivní úvahou o různých přístupech, technikách a myšlenkách týkajících se daného tématu. Samostatné části budou věnovány zpracováním databázových transakcí u serverové části a rozdělení klientské webové aplikace na menší, samostatné celky.

Nabyté znalosti se uplatní v praxi ve formě IS pro správu projektů, který byl původně navržen a implementován jako prototyp monolitické architektury v bakalářské práci „Informační systém pro správu studijních projektů“ [3]. I když tento systém bude brán jako stěžejní pro uplatnění znalostí o architektuře a bude uskutečněn přechod z monolitické architektury na architekturu mikroslužeb, tak za cíl není kladeno projekt výrazně zlepšit z uživatelského hlediska nebo uvést do provozu v době dokončení diplomové práce. Postupovat se bude inkrementálně – nejdřív se implementuje nejdůležitější, nezbytná osnova a zbylá funkcionalita bude dokončena dle časových možností, případně uvedena v seznamu pro další rozvoj.

Větší část analýzy a specifikace uvedené v bakalářské práci [3] bude převzata, některé aspekty však budou explicitně zaktualizovány a popsány v kapitolách „Analýza předchozí práce“ a „Specifikace nového systému“, protože mohou způsobovat nevhodný dopad – bezpečnost, nevyhovující funkcionalita apod.

Konečným výstupem bude především popis subjektivně prozkoumané oblasti MSA sloužící jako rychlý přehled architektury mikroslužeb s případnými dodatečnými materiály, zrefaktorovaná verze IS, jež je převedena z monolitické architektury na architekturu mikroslužeb, a dodaná příručka vývojáře. V rámci implementace bude zdokonaleno testování a nasazování výsledné služby.

Struktura

Kapitola 1 se věnuje stručné analýze výsledků předchozí práce, definuje silné a slabé stránky a navrhuje body pro zlepšení, zejména pro MSA.

Kapitola 2 se zabývá obecným zkoumáním problematiky MSA a typickými situacemi, které se potenciálně mohou řešit během implementace systému s využitím MSA na straně serveru i klienta.

Kapitola 3 rozebírá integraci mikroslužeb s datovými úložišti a problematiku transakčního zpracování v případě oddělených úložišť.

Kapitola 4 specifikuje požadavky nového projektu s návazností na analýzu předchozí práce a v souladu se zadáním diplomové práce.

Kapitola 5 obsahuje analýzu a implementaci serverové části IS v MSA s uplatněním informací popsaných v kapitolách analýzy MSA.

Kapitola 6 obdobně, jako v případě 5. kapitoly, obsahuje analýzu a implementaci, ale již klientské části aplikace.

Kapitola 7 se věnuje popisu zvloleného testování obou částí IS a automatizaci.

Kapitola 8 se zabývá kontejnerizací a nasazováním IS.

Kapitola 9 je věnována nedostatkům aktuálního řešení a potenciálnímu budoucímu rozvoji.

Kapitola 10 porovnává původní monolitický prototyp s novou realizací systému a uvádí přínosy a nevýhody přechodu.

V závěrečné kapitole je zhodnocen výsledek, dosažení předem stanovených cílů a splnění zadání diplomové práce.

V přílohách a na přiloženém médiu jsou uvedeny zdrojové kódy všech částí IS, dokumentace pro vývojáře a jiné dodatečné materiály.

Kapitola 1

Analýza předchozí práce

V dané kapitole:

- analýza posudků vedoucího a oponenta předchozí práce,
- analýza popsaných možností rozvoje systému,
- zkoumání silných a slabých stránek předchozí implementace,
- revize změn na fakultě z hlediska správy projektů.

Tato diplomová práce navazuje na bakalářskou práci z roku 2019 [3] a využívá jak analýzu/výsledky popsané v práci samotné, tak i přílohy (zejména zdrojový kód serverové aplikace, klientské aplikace a vývojářské a uživatelské dokumentace). Pro implementování daného systému v MSA a zajištění kvality výsledku je provedena opakovaná analýza problematiky a korekce potřebných oblastí.

1.1 Potenciální možnosti rozvoje systému

1.1.1 Posudky

Ze závěrečných posudků vedoucího a oponenta bakalářské práce je nutno vyčlenit několik významných bodů pro zlepšení.

- **Neúplná či nedostatečně propracovaná dokumentace [4]** – je třeba přepracovat poskytnuté dokumentace a doplnit relevantními informacemi – přidat informace o architektuře, způsobu fungování, zřetelnější práci s databází a použité technologie.

- **Velice stručně popsané testování [5]** – v poskytnutém systému je velice málo automatizovaných testů a probíhalo zejména manuální testování [3] – některé části se dají dobře začlenit do vývojového cyklu s automatickým spouštěním.
- **Nebylo popsané konkrétní určení informačního systému [5]** – informační systém od začátku nebyl cílený pro konkrétního spotřebitele.

Zároveň během obhajoby bakalářské práce bylo nabídnuto zvážit správu a ukládání projektu ve VCS (version control system) git místo manuálního ukládání v NoSQL databázi (MongoDB). To by mohlo zredukovat množství potřebného kódu a snížilo spotřebu fyzické paměti (jednotlivé snímky odevzdaných projektů by se neduplikovaly, ale ukládaly jako git značky[1]).

1.1.2 Možnosti rozvoje

Bakalářská práce navrhuje rozvoj 2 hlavními směry - obecné univerzální zdokonalování a rozvoj se zaměřením na FIT (Fakulta informačních technologií) ČVUT (České vysoké učení technické) [3]. Jelikož daná práce se soustředí především na MSA, tak rozvoj se zaměřením na fakultu bude považován za sekundární. V předchozí práci bylo nabídnuto několik bodů pro rozvoj [3]:

- **Podpora internacionalizace a lokalizace** – aplikace poskytuje pouze anglické rozhraní. Nepoužívá se žádný framework nebo knihovna, která by napomáhala snadné správě překladů.
- **Hromadné zakládání projektů** – neexistuje způsob hromadného zakládání projektů, i když dle předchozí analýzy by byl prospěšný.
- **Serverová implementace snímků** – nedokončená funkcionalita pro odevzdávání jednotlivých iterací projektů.
- **Nepříznivé scénáře API (application programming interface) dotazů** – v případě pádu serveru a neočekávané API odpovědi často chybí uživatelsky přijatelné oznámení/zpracování.
- **Nové interprety obsahu** – jedná se o omezený výběr vytvářených typů obsahů.
- **Propracovaná integrace se službami třetích stran** – funkcionalita propojení fakultních služeb pro známkování, autorizaci apod.
- **Analýza využití systému a aktualizace UX (user experience)** – bez produkčního prostředí (nebo jiného dlouhodobého uživatelského testování) nebylo možné získat daná data.
- **Optimalizace stávajícího systému** – různorodá optimalizace předchozího IS.

[1] git tags

1.1.3 Revize kódu

- **IS není plně kontejnerizovaný** – IS není plně převeden na kontejnery, může být zdlouhavější start projektu, to se v pozdějších fázích odrazí i na předpokládané škálovatelnosti služby.
- **Monolitická struktura aplikace** – jakákoliv úprava vyžaduje editaci celého systému, není možné pohodlně měnit jednotlivé části aplikace.
- **Staré a vyřazené z provozu knihovny** – v klientské React aplikaci se nachází starší knihovny, jež musí být aktualizovány (bezpečnost, funkcionalita apod.). Rovněž je možné uplatnit nový způsob psaní React aplikací – s React Hooks.
- **Konfigurace a zbytečné oddělování prostředí** – bude potřeba zvážit konfiguraci přes `.env` soubory a odstranit rozdělení způsobů startu aplikace dle prostředí.
- **Struktura projektu, architektura** – struktura složek a části architektury se zdají být v mnoha aspektech zbytečné a přispívají k horší čitelnosti.

1.2 Silné stránky

Z hlediska pozitivních prvků poskytnuté implementace lze vytknout několik skutečností:

- **JavaScript, TypeScript a Node.js** – serverová a klientská část aplikace jsou psány v jazyce JavaScript (případně TypeScript), je to výhodné z hlediska udržování systému.
- **Knihovna Next.js** – klientská část je psána v jedné z moderních knihoven Next.js s podporou SSR (server side rendering) a SSG (static generation) [6].
- **Dynamicky generovaný obsah projektů** – způsob tvorby obsahu stránky připomíná implementaci mikroslužeb na straně klienta.
- **Kontejnerizace aplikací** – databáze jsou poskytovány s pomocí služby Docker a potřebují minimum konfigurací.

1.3 Alternativní systémy správy projektů na fakultě

Vzhledem k relativně dlouhé době (3 roky) od dokončení bakalářské práce je potřeba provést stručnou aktualizaci seznamů alternativních řešení pro správu projektů na fakultě. U předchozích systémů fungujících na fakultě (v návaznosti na rešerši způsobů správy projektů na FIT ČVUT [3]) je možné zaznamenat následující změny:

- **Project/SwinPro** – provoz aplikace je ukončen ke dni 1. 10. 2021 a veškerá data jsou dostupná pouze na požádání prostřednictvím osobního e–mailu [7].

- **Roundcube**[2] – byl nahrazen službou Microsoft 365 s vlastními e–mailovými schránkami [8].
- **Microsoft Teams** – během vzdálené výuky některé předměty využily funkcionality služby Microsoft Teams pro vytvoření požadavku na odevzdání souboru semestrální práce[3].
- **Studentský odevzdávací systém** – vznikl nový portál pro studentské projekty. Jelikož se jedná o komplexní řešení, bude rozebrán samostatně v následující podkapitole.

1.3.1 Studentský odevzdávací systém (SOS)

Autor: kolektiv autorů (závěrečné práce)
Analyzovaná verze: 0.2.3-alpha
Datum analýzy: 31. 10. 2021
URL: `https://sos.fit.cvut.cz/`

Studentský odevzdávací systém je relativně nový portál – nachází se v alpha verzi a využívá se například v předmětu NI-NUR. Konceptuálně je myšlen jako univerzální systém pro všechny možné projekty, které vyžadují iterativní odevzdávání souborů a obdržení ohodnocení za odevzdanou práci. Z pohledu studenta lze vytvořit návrh projektu, který může, ale nemusí být schválen vyučujícím, bohužel nelze vytvářet nekontrolované projekty pro vlastní účely. Následuje tvorba projektu mimo systém a odevzdávání výsledků v podobě samostatných souborů s případným komentářem. Započatý projekt nelze přerušit ze strany studenta. Daný systém se v mnoha ohledech podobá IS, který byl analyzován v předchozích podkapitolách.

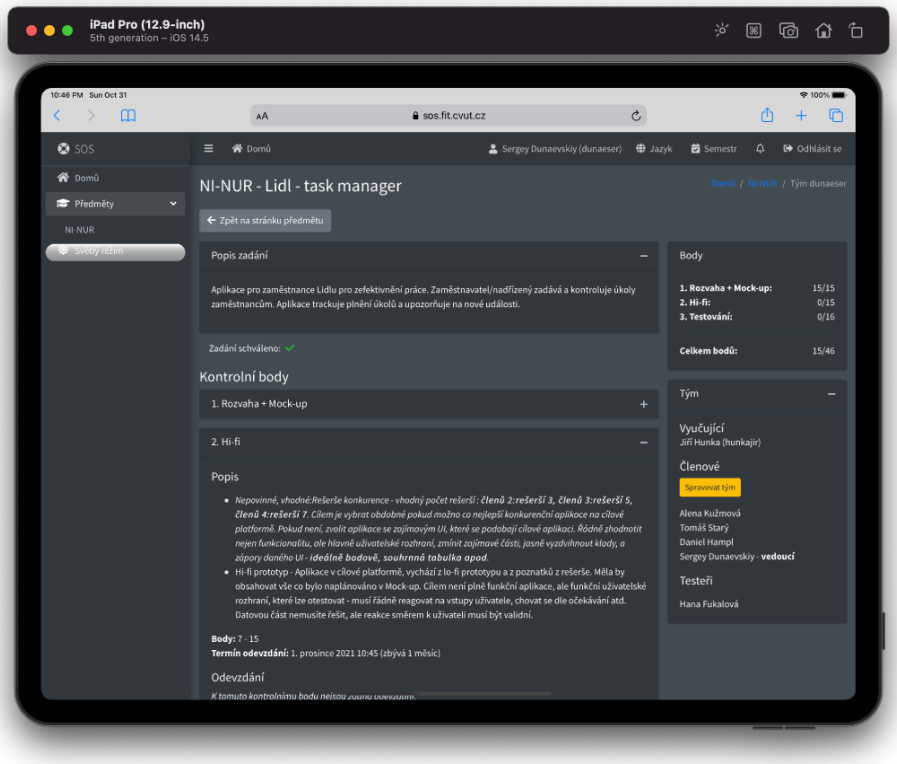
Společné rysy

- **Iterativní odevzdávání projektu** – založený projekt se odevzdává v iteracích (kontrolních bodech), které mohou být okomentovány a ohodnoceny ze strany vyučujícího.
- **Odpovědná osoba** – existuje role (vyučující), která dokáže ohodnotit kontrolní bod určitých počtem bodů.

Výhodné prvky

- **Nahrávání souborů** – v rámci iterace lze nahrávat libovolné soubory.
- **Integrace s fakultními systémy** – IS je víc přizpůsobený pro fakultní účely.

[2] Webmail
[3] Jedná se o osobní zkušenost z předmětů NI–MPI a NI–PIS



Obrázek 1.1: Studentský odevzdávací systém – ukázka detailu projektu z pohledu studenta

- **Tmavé rozhraní** – UI (user interface) má volitelný tmavý režim, viz obrázek 1.1.
- **Jazyky** – UI je poskytováno v anglickém a českém jazyku s libovolným přepínáním.

Negativní prvky

- **Restrikce zakládání projektů** – studenti nemohou zakládat vlastní, nezávislé projekty, musí spadat pod určitý předmět. Nelze založit a spravovat víc projektů.
- **Nelze založit víc projektových rolí** – projektové role i pro vedoucího projektu jsou omezeny na členy týmu a testery, nelze přidat libovolnou roli.
- **Neintuitivní rozhraní** – některé prvky uživatelského rozhraní nejsou intuitivně pochopitelné (subjektivně) a při kritické změně nevyžadují potvrzení.

Kapitola 2

Obecný úvod do architektury mikroslužeb

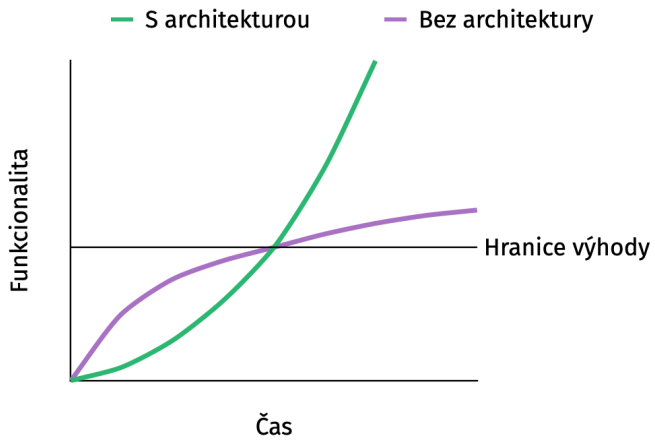
V dané kapitole:

- obecné informace o architektuře a proč je důležitá,
- porovnání vybraných architektur – MSA, MA (monolithic architecture) a SOA (service oriented architecture),
- typy dekompozice specifikace na MSA architekturu,
- komunikace mezi mikroslužbami a řešení závislostí,
- testování a validace mikroslužeb,
- nasazení mikroslužeb na server a základní monitorování,
- krátký úvod do služeb na straně webového klienta.

Softwarovou architekturu jako pojem je těžké exaktně definovat, každý vývojář k ní může přistupovat a vnímat jinak. Obecně může být popsána jako jistý řád a pravidla, vzniklá následkem mnoha rozhodnutí v průběhu analýzy a vývoje produktu. Veškerý následující rozvoj by se měl striktně řídit těmito pravidly, aby umožnil vznik dlouhodobě udržovatelného výsledku [9].

Vývoj softwaru bez architektury nebo s nepřesně definovanou osnovou může být přínosný v krátkodobé perspektivě nebo z hlediska šetření počátečních nákladů na čas a finanční složky [9]. V případě dlouhodobého projektu to však může znamenat hromadění technického dluhu [10]. Dle článku [10] přínos architektury lze vizuálně znázornit

grafem 2.1, kde je uvedena kumulativní funkcionalita v závislosti čase spotřebovaným projektem. Používání architektury ze začátku způsobuje zpomalení vývoje, ale od jisté hranice výhody se stává přínosnou a urychluje rozvoj. Tento předpoklad však funguje pouze v případě, že se jedná o dobře zvolenou a popsanou architekturu (v textové nebo diagramové podobě), která má pozitivní vliv a je dostatečně ohebná [10].



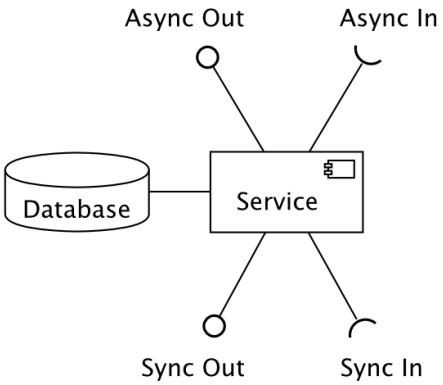
Obrázek 2.1: Bod přínosu dodržování architektury [10]

Ačkoliv neexistuje přesná definice architektury obecně, existují detailněji popsané typy architektur, které jsou vhodné pro vývoj aplikací. Jejich volba a případná adaptace vyžaduje pochopení konečného cíle požadovaného výsledku [9]. V návaznosti na zadání diplomové práce v dané kapitole bude popisována především MSA a bude porovnávána s jinými architekturami, které by ji potenciálně mohly nahradit. Tyto vybrané architektury – MA, SOA[4], SA (serverless architecture) – budou zkoumány vzhledem k přístupu k určitým aspektům, poskytovaným možnostem a výhodám a nevýhodám vůči MSA. Vzhledem k dříve použitému jazyku TypeScript/JavaScript budou i srovnání zaměřeno na tento jazyk, případně Node.js prostředí.

Než se začne s konkrétním porovnáním, je třeba definovat jeden společný pojem všech 3 architektur – službu. Služba v této práci je chápána jako atomicky fungující celek, z větší části nezávislý na ostatních – soustředí se na konkrétní funkcionalitě (nebo skupině funkcionalit), má přístup k databázovému úložišti a veškerá komunikace probíhá přes striktně definované rozhraní (vizualizaci takové služby je možné vidět na obrázku 2.2). Může fungovat jako celek poskytující, přijímající a zpracovávající nebo předávající datové zprávy. Jakýkoliv jiný vliv, než přes dohodnuté rozhraní, je ignorován

[4] též architektura orientovaná na služby

a musí být ideálně eliminován. Rozhraní takové služby může být popsáno s pomocí samostatné dokumentace nebo dokumentovaného kódu.



Obrázek 2.2: Abstraktní znázornění služby

Kvůli nejednoznačnosti pojmu „architektura“ nejspíš existuje nespočetné množství modifikací a adaptací výše uvedených architektur (MA, SOA, MSA, SA), proto se bude předpokládat, že se jedná o takto definované instance:

MA – jednoprocesový program atomické povahy – nelze z něho jednoduše vyčlenit funkční celky, které by se daly beze změn využívat v jiných programech. Obsahuje globální jednorázové připojení k datovému zdroji, které se provádí během startu. Takový program je sám o sobě službou.

SOA – jednoprocesový program s interním rozdělením na služby, které mezi sebou komunikují s pomocí zpráv přímo s využitím vyčleněného rozhraní nebo přes ESB (enterprise service bus) – sběrnicí určenou pro centralizaci komunikace mezi službami. Vazba na datový zdroje může, ale nemusí být jedna (každá služba může mít samostatné připojení).

MSA – několikaprocesový systém služeb (každá služba má právě jeden proces) organizovaný do větší interně kompatibilní struktury.

SA – architektura aplikace, která postrádá kontinuálně běžící serverový proces a je pouze rozmístěna na FaaS (function as a service) řešení. Jinými slovy funkcionalita není spuštěna v nepřerušovaném prostředí (jako démon), ale je dostupná na požádání. Během uživatelského dotazu je vytvořena potřebná instance aplikace, případně navázáno databázové spojení, vykonán požadavek a následně je instance odstraněna.

Pro porovnání architektur bylo vybráno několik klíčových pojmů, které mohou během návrhem a implementací programu mít největší vliv na rozhodování o výběru architektury.

Jednoduchost vývoje – náklady spojené s vývojem programu a začlenění nových vývojářů do týmu.

- **MA** – jednoduchý počáteční vývoj kvůli jednodušší koncepci architektury, IDE (integrated development environment) jsou danému typu vývoje více přizpůsobeny. S rostoucím vývojovým týmem může být problém udržování konzistentní a stabilní aplikace. Počáteční náklady pro začlenění nového vývojáře v pozdějších fázích mohou být vysoké kvůli potřebě pochopení celého systému a často zastaralých technologiích [11].
- **SOA** – rozdělení na služby přináší možnost rozdělit vývoj mezi několik nezávislých týmů, jež budou mít přesně definované rozhraní. V případě vyžuívání společných prvků, například ESB, je třeba zajistit, aby se konvence nerozcházely.
- **MSA** – velice podobný vývoji, jako SOA, služby jsou však nezávislé i technologicky, mohou být realizovány v různých jazycích a prostředích. Může to být přínosem, jelikož můžeme volit technologie dle potřeb každé služby, ale zhorší se tím univerzalita týmu (ne každý člen týmu bude mít potřebné znalosti).

Radikální změny – složitost změny nebo přidání obchodní logiky do aplikace, která by měla dopad na větší část dosavadní aplikace.

- **MA** – veškeré změny se týkají jednoho jediného celku zdrojového kódu. Nevýhodou je možný nekontrolovaný dopad na celou aplikaci, protože nejsou striktně oddělené části projektu [11]. Takové změny lze řešit například vhodným dělením zdrojového kódu na balíčky nebo knihovny.
- **SOA, MSA** – v případě vhodně zvolené soudržnosti a provázanosti radikální změny funkcionality by byly zčásti omezené službou jako takovou, případně rozhraním komunikace.

Tolerace chyb – chování systému v případě výskytu neočekávané chyby nebo výjimky, která není zpracována manuálně či JS (JavaScript) prostředím.

- **MA, SOA** – jelikož se jedná o jednoprocesovou aplikaci, tak jakákoliv neošetřená chyba způsobí kolaps celého systému.

- **MSA** – několikaprocesové prostředí zajišťuje větší toleranci chyb, ale vždy záleží na ovlivněné části systému. V případě sekundární služby, která komunikuje například přes asynchronní broker zpráv, výpadek nebude mít stejně závažný dopad, jako v případě selhání samotného brokeru nebo jedné z klíčových služeb (autorizace apod.).

Komunikační latence – během komunikace mezi jednotlivými funkcionalitami aplikace může dojít k zpomalení vzhledem ke zvolenému přístupu.

- **MA** – v monolitu můžeme předpokládat přímé volání potřebných metod, latence je zde potenciálně minimální.
- **SOA** – vzhledem k definovanému komunikačnímu kanálu, který vyžaduje výměnu zpráv, můžeme počítat s časem potřebným pro vytvoření, odeslání a přijetí zprávy před vykonáním potřebné činnosti.
- **MSA** – obdobně, jako v případě SOA, ale samotné doručení zprávy může být ještě víc zpomaleno vybranou technologií, například komunikace se vzdáleným serverem nebo prostřednictvím vnějšího brokeru zpráv.

Datové úložiště – využití perzistentního úložiště (SQL/NoSQL databáze) pro zápis a čtení informací. Počáteční inicializace databáze, vytvoření struktury, migrace.

U všech zkoumaných architektur může být využito jak jedno, tak i víc datových úložišť. V případě oddělených služeb (zejména u MSA) je někdy vhodné použít samostatné, oddělené datové úložiště pro každou službu kvůli zachování větší nezávislosti. Taková struktura přináší komplikace z hlediska provádění transakcí přes několik datových úložišť a spojování (`JOIN`), které vzhledem k fyzickému oddělení nelze provádět jedním SQL (structured query language) dotazem. Víc o tomto rozdělení bude zmíněno v kapitole „Správa dat v architektuře mikroslužeb“.

Horizontální škálování – škálování aplikace kvůli rozdělení zátěže na systém.

- **MA, SOA** – jednoduché škálování – vzniká větší počet instancí aplikace, které mohou být umístěny za prvkem vyvažující zátěž [11].
- **MSA** – vzhledem k samostatným procesům všech služeb škálování vyžaduje pokročilejší infrastrukturu a správu, avšak poskytuje i pokročilejší možnosti. V případě nerovnoměrné zátěže systému je možné tuto část samostatně škálovat (protože se jedná o samostatný proces) a přizpůsobovat potřebám.

Testování – přizpůsobenost architektury psaní automatizovaných testů a jejich schopnost spouštět se automaticky v předem definované situaci (během vývoje, po nasazení apod.).

- **MA, SOA** – jednoprocesová aplikace může být testována jako celek. Aplikace třetích stran mohou být nahrazeny falešnými servery[5] poskytujícími předpřipravená, nebo prázdná data.
- **MSA** – každá služba je zde jako jedna aplikace MA, všechny ostatní služby jsou pro ní třetí stranou, která musí být během integračních testů nahrazována.

Nasazování – časová náročnost a komplexita rozmístění funkční aplikace na server.

- **MA, SOA** – jednoprocesové aplikace mají jeden životní (nasazovací) cyklus.
- **MSA** – v případě samostatných služeb nasazení každé představuje vlastní životní cyklus, nejspíš budou potřeba některé synchronizované události (například spouštění některých služeb později, než jiných). Složitost nasazovacího cyklu se řídí složitostí architektury.

2.1 Využití MSA pro modelování světa

Pro člověka nejspíš neexistuje nic přirozenějšího, než prostředí, ve kterém se pohybuje a kterému rozumí – od osobních věcí a procesů, jež musí vykonávat, až po strukturu jeho okolí – město, stát, planeta, sociální vztahy, komunikace s lidmi.

Všechny tyto činnosti můžeme popsat pomocí subjektů – samostatných účastníků procesů, rozhraní, které poskytují pro komunikaci, a zpráv[6], jež se předávají v rámci komunikace. Každý subjekt je skupinou izolovaných funkcí s různě komplikovanou sadou komunikačních kanálů. Může mít svoje potřeby a vytvářet podněty pro ostatní subjekty. Ne každý subjekt je schopen zpracovávat veškerou informaci, která k němu přichází od jiných subjektů.

Daný koncept naprosto přesně napodobuje MSA. Jednotlivé subjekty jsou mikroslužby, mají vlastní rozhraní a vysílají informace v přesně definovaných strukturách. Mohou existovat samostatně, i když jejich smysl existence nemusí existovat. Zároveň jsou schopny zachytit a zpracovat zprávy, které byly určeny pro jejich použití a formát kterých je popsaný ve vnitřní logice.

[5] anglicky rovněž mock-server

[6] Zprávou se rozumí informace poskytnutá v samostatné struktuře – například věta nebo formát typu JSON (javascript object notation)

V případě mírného zjednodušení se můžeme dostat ke konceptu SOA. Subjekty zachovávají způsob komunikace s pomocí zpráv, ale již nejsou samostatní, potenciálně fungující celky, nýbrž moduly jednoho většího bloku. Komunikace u takové architektury může být centrálně řízena ESB [12].

Monolitní architektura v porovnání se SOA zjednodušuje i samotné rozdělení do modulů. Stále se jedná o samostatně fungující celek, ale vnitřní struktura už postrádá moduly s odděleným rozhraním komunikace s využitím zpráv. V rámci modelování světa se dá představit jako interně nedělitelný celek, který pouze poskytuje rozhraní pro komunikaci, tudíž je to subjekt a v rámci MSA může představovat mikroslužbu.

Na základě výše uvedených informací by logicky bylo nejjednodušší vždy vytvářet pouze MSA architekturu, protože je pro pochopení nejsnazší kvůli zkušenostem z každodenního života. Každý subjekt má svoje potřeby (potřeba vytvořit zprávu, potřeba zpracovat příchozí právu) a o zbylé aspekty se nestará. Problém nastává kvůli komplexnosti takového konceptu. Jakékoliv rozdělení celku implicitně předpokládá nové náklady pro definování komunikace, které jsou náročné pro představu. Proto může být nejvýhodnější začínat s architekturou předpokládající nejvíc redukovaným rozdělením – monolitem – a dle potřeby měnit hloubku rozdělení.

Navíc při detailním zkoumání ve všech výše uvedených architekturách se dá vypozorovat rekurzivita. Monolitická aplikace, nebo SOA aplikace může tvořit mikroslužbu v rámci MSA. MSA aplikace může tvořit mikroslužbu jiné MSA a fungovat jako monolit pro ostatní.

2.2 Dekompozice na MSA

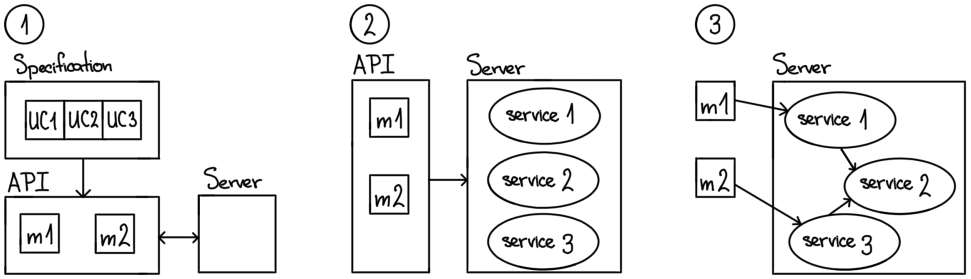
Projekt, u něhož bylo rozhodnuto o MSA, vyžaduje, jako jeden z kroků před implementací, dekompozici zadání na oblasti, dle kterých budou následně vytvářeny jednotlivé služby. Stejně jako v případě myšlenky modelování světa, účastníky procesů budou subjekty s vlastním rozhraním pro komunikaci, které v případě projektu lze vyčíst ze specifikace.

Definování konkrétní architektury projektu se dá provádět postupně ve 3 fázích [11] (vizualizace na obrázku 2.3):

1. Definování operací zpracovávaných serverem – na základě specifikace projektu, ve které jsou popsány případy užití očekávané od serverové části aplikace, formulujeme do konkrétních volání – získat, vytvořit, aktualizovat data apod.
2. Definování možných služeb – pro dekompozici na konkrétní služby existují dva

základní přístupy – rozdělení dle subdomén a rozdělení dle obchodních potřeb [11], oba přístupy budou stručně popsány v dalších podkapitolách. Rozdělování vychází z navržených volání a přiřazuje je jednotlivým službám.

3. Definování propojení požadavků se službami a komunikace služeb mezi sebou – prohlubuje definovanou komunikaci mezi službami a vytváří konkrétní interní a externí spoje.



Obrázek 2.3: Tři kroky návrhu MSA

V rámci rozdělování operací do konkrétních služeb (stejně jako návrh služeb) mohou pomoci i návrhové vzory týkající se vývoje samotného, jako jsou například:

Princip jedné odpovědnosti – každá třída musí mít právě jeden důvod, proč se měnit [13]. V kontextu mikroslužeb se to rovněž vztahuje i na službu samotnou – musí se zabývat pouze jednou subdoménou řešeného problému.

Vysoká soudržnost – služba obsahuje vše potřebné pro řešení oblasti, za níž je odpovědná [14].

Nízká provázanost – služba může získávat informace z ostatních zdrojů, ale snížená provázanost ulehčuje vývoj [14].

Vzhledem k širokému rozsahu dostupných modifikací MSA je sem možné začlenit mnohem větší spektrum pravidel a doporučení, vždy záleží na aspektech konkrétního zadání. Některé z nich budou popsány v následujících kapitolách a mohou mít vliv na rozhodnutí spojená s dekompozicí vytvářené funkcionality.

2.2.1 Dekompozice dle obchodních potřeb

Dekompozice oblastí dle obchodních potřeb je jednou ze dvou základních možností, jak dekompozici provádět [11]. Soustředí se na vazbě s architekturou společnosti, neboli něčem, co firmě přináší užitečnou hodnotu [15]. Uvažujme případ, že o vytvoření MSA požádala firma, která zpracovává různé druhý objednávek - oblečení, potraviny,

sportovní nářadí a spravuje interní pracovníky – stálé zaměstnance a příležitostnou výpomoc. Z takové struktury by mohlo vzniknout 5 služeb, které by spravovaly:

- objednávky oblečení,
- objednávky potravin,
- objednávky sportovního nářadí,
- správu zaměstnanců,
- správu výpomoci.

2.2.2 Dekompozice dle subdomén

Dekompozice z hlediska subdomény, která představuje druhý způsob rozdělení, na rozdíl od obchodních požadavků nevnímá architekturu společnosti jako zásadní, i když je nutná, a upřednostňuje logické rozdělování [16]. V případě stejného příkladu firmy by dekompozice dle subdomén mohla vypadat následovně:

- objednávky,
- správa pracovníků.

Takové služby už by nebyly specializované na konkrétním užití a tudíž by mohly být větší nároky na jejich schopnost se přizpůsobovat potřebám IS.

Typ dekompozice nelze přesně stanovit, je třeba se dívat na všechny potřebné aspekty zkoumané domény a přizpůsobovat dané metody konkrétním situacím [11].

2.3 Komunikace MS

Mikroslužby jsou definované jako vždy oddělené z hlediska procesů a pro komunikaci s jinými službami nebo externími systémy musí udržovat kompatibilní API vůči jejich rozhraní a prostředí. U takové komunikace můžeme zkoumat typ, technologii a hierarchii/strukturu, jež se řídí požadavky na systém.

2.3.1 Typ komunikace

Typ komunikace se abstrahuje od konkrétních řešení a zkoumá pouze konceptuální potřeby komunikace systému. Obecně se uvádí dělení na dvě podskupiny – dle počtu cílových služeb a dle synchronity komunikace, které se dají různě kombinovat, přičemž každá kombinace může mít víc podtypů [11]. Celý přehled je uveden v následujícím seznamu:

Synchronní 1:1 – synchronní komunikace dvou subjektů.

- **Dotaz/odpověď** – producent vytváří právě jeden dotaz na jinou službu a čeká na pozitivní, nebo negativní odpověď. Tento způsob vede ke zvýšení provázanosti MS (microservice).

Synchronní 1:N – synchronní komunikace s více cíli nemůže existovat, protože by vznikla sekvence synchronních požadavků 1:1 kvůli jednomu producentu.

Asynchronní 1:1 – asynchronní komunikace dvou subjektů.

- **Asynchronní dotaz/odpověď** – producent vytváří jeden dotaz a nečeká na okamžitou odpověď. Odpověď se může poskytnout kdykoliv (nebo vůbec) – chování nesmí být blokujícího typu.
- **Oznámení** – producent vytváří jeden dotaz (oznámení), zpětná komunikace se neočekává a neexistuje.

Asynchronní 1:N – asynchronní komunikace s více cíli.

- **Producent/konzument** – producent vytváří jeden dotaz a předává všem konzumentům.
- **Producent / asynchronní odpovědi** – producent vytváří jeden dotaz a předává všem konzumentům. Následně čeká předem danou dobu na potenciální asynchronní odpovědi od konzumentů.

2.3.2 Technologie komunikace

Technologií může být myšlen protokol nebo systém doporučení, který může být realizován v konkrétním programovacím jazyce. Vzhledem k Node.js se může jednat například o následující způsoby:

- **REST** – běžné GET/POST/…dotazy přes HTTP (hypertext transfer protocol).
- **GraphQL** – dotazovací jazyk nad strukturovanými daty [17].
- **gRPC** – framework založený na RPC (remote procedure call) protokolu [18].
- **RabbitMq** – broker zpráv [19].
- **Kafka** – platforma pro distribuované streamování událostí [20].
- **MQTT** – standard pro IoT (internet of things) komunikace [21].

Výběr technologie má dopad na provázanost služeb v systému. V případě například REST (representational state transfer) implementace bude provázanost větší kvůli požadované odpovědi v relativně krátké době. Volba RabbitMq naopak může zajistit menší provázanost tím, že zpracování se bude řídit brokerem zpráv, avšak v případě nutné okamžité odpovědi může způsobovat komplikace.

2.3.3 Struktura komunikace

Pokud pomineme konkrétní způsob výměny informací mezi mikroslužbami a podíváme na se obecnou organizaci komunikace, tak existují dva základní způsoby – orchestrace a choreografie [22].

Orchestrace – předpokládá analogii s hudebním orchestrem, kdy každý člen systému zná vlastní roli, ale stejně musí být řízen dirigentem. V tomto případě dirigent je zastoupen orchestrační mikroslužbou, která vystupuje jako prvek, jenž zprostředkovává veškerou komunikaci. Tento koncept je úzce spjat s RESTful API a může v případě obrovského počtu MS způsobovat komplikace, protože orchestrační prvek bude muset zvládat stovky až tisíce mikroslužeb a ve výsledku dopadne jako těžce spravovatelná monolitická část [22]. Z implementačního pohledu však může být o něco jednodušší, protože jakákoliv propagace chyb, sběr dat a další požadavky mohou být přímočařejší a soustředěny na jednom místě.

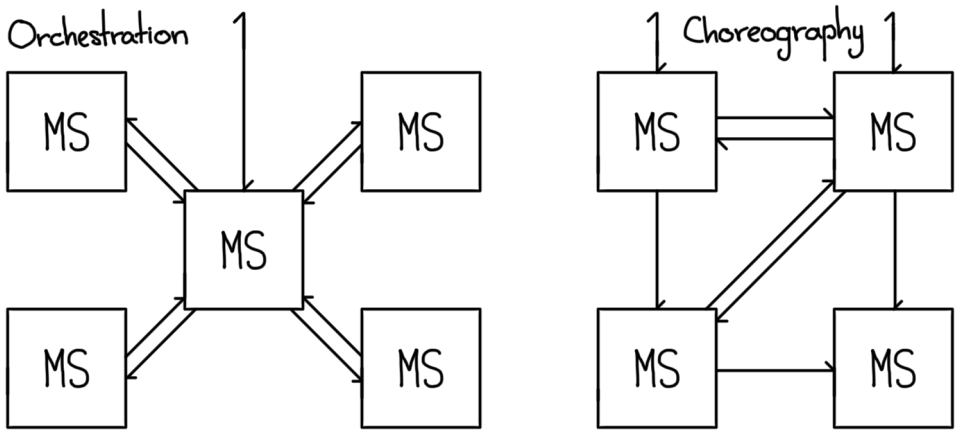
Choreografie – analogicky představuje taneční skupinu, kde neexistuje řídící prvek, ale každý člen reaguje na podnět (hudbu, jiné členy) a provádí potřebné kroky. Implementačně musí existovat komunikace řízená událostmi, kdy jedna služba vysílá zprávu do brokera a o zbytek se nemusí starat, vše je řízeno asynchronně. Zároveň každá služba odposlouchává pouze zprávy, na které má reagovat. Tento koncept zajišťuje mnohem větší ohebnost a menší provázanost služeb [22]. Přináší však potřebu lépe zpracovávat chyby kvůli asynchronním akcím, které nemusí nikdy doběhnout. Zejména se to projevuje u původně synchronních požadavků, například ze strany prohlížeče, kdy se musí čekat na odpověď. Aby nenastalo potenciálně nekonečné očekávání dokončení požadavku na server, tak se může zvolit přiměřeně dlouhá časová doba, kdy na straně serveru se vykonávání požadavku začne brát jako nezdařilé. V tomto případě se vrací chybná odpověď a je potřeba zrušit všechny změny, které byly provedené dosud a které se mohou provést po dokončení pomalu zpracovaného požadavku.

Možná vizualizace obou struktur je znázorněna na obrázku 2.4. Samozřejmě může docházet i k prolínání takových řešení – například choreografie se synchronní komunikací.

2.3.4 API Gateway

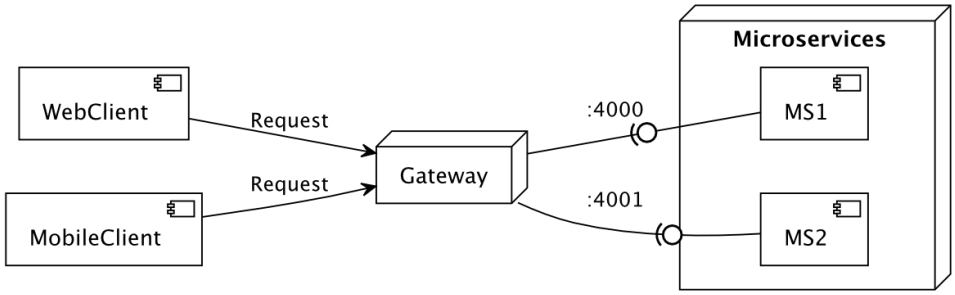
Na rozdíl od monolitické architektury, kde je veškeré rozhraní poskytováno v rámci jednoho portu, MSA, vzhledem ke své struktuře, vystavuje několik přístupových bodů (jeden port pro každou mikroslužbu). V takovém případě může vzniknout problém

s konflikty – využití jednoho portu několika službami, nebo nepohodlné volání rozhraní, kdy pro každou službu bude třeba doplňovat konkrétní port a celá aplikace již nebude působit jako celek. Další problém nastane v případě dynamické volby portů v rámci infrastruktury nebo potřebě dočasně znemožnit uživatelům volat určité části rozhraní [23].



Obrázek 2.4: Orchestrace a choreografie MS

Taková situace se může řešit s pomocí API Gateway vzoru [23], viz obrázek 2.5, kdy se vytváří jistý typ proxy prvku, který vystupuje pod jedním portem a veškeré požadavky třídí a směruje na jednotlivé služby dle unikátních příznaků.



Obrázek 2.5: Architektura s API Gateway

2.4 Udržování závislostí

V průběhu života systému a dodávaných řešení s MSA je pravděpodobné, že vlivem měnících se podmínek bude docházet i ke změnám potřeb poskytovaných rozhraní. Takové modifikace mohou vést k nekonzistenci a narušení systému jako celku. Tuto situaci je možné kontrolovat a řešit s pomocí verzování v různých variantách [24].

2.4.1 Sémantické verzování v URI

V tomto případě na základě konceptu sémantického značení se verzuje celý zdroj dat [24] – URI (uniform resource identifier) – jako `major.minor.patch` [25]. Zápis může vypadat následovně:

```
schema://domain/name-service/v1.0.0/entity
```

Výhodou takového způsobu je možnost udržovat víc aktivních API verzí v rámci jedné MS v případě nekompatibility napojených systémů. To má dopad na klientskou stranu, jež má v tomto případě plnou kontrolu nad tím, jaké rozhraní používá, a v systému pravděpodobně nehrozí náhlá chyba po aktualizaci serveru.

Přináší to však i řadu nevýhod. Provolávání API klientskou aplikací je vázáno na URI, tudíž veškeré změny bude třeba zpracovat i v klientské aplikaci, což nemusí být vždy vhodné. Řešením by mohla být redukce zápisu verze a ignorování `patch`, případně i `minor` složek, pak menší opravy nebudou vyžadovat opravu klienta. V případě udržování víc verzí jednoho rozhraní mohou vznikat zbytečné komplikace s udržováním kódu.

2.4.2 Verzování v hlavičkách HTTP

Velice obdobný způsob, jako je verzování v URI, přináší verzování v HTTP hlavičnách. Hlavním přínosem je neměnný URI pro provolávání ze strany klienta [24]. Tudíž nevyžaduje přísnou kontrolu verze provolávaného rozhraní, avšak poskytuje možnost zjišťovat a případně i omezovat volání na základě hlavičky. Poskytovaná verze může být využívána například v logování.

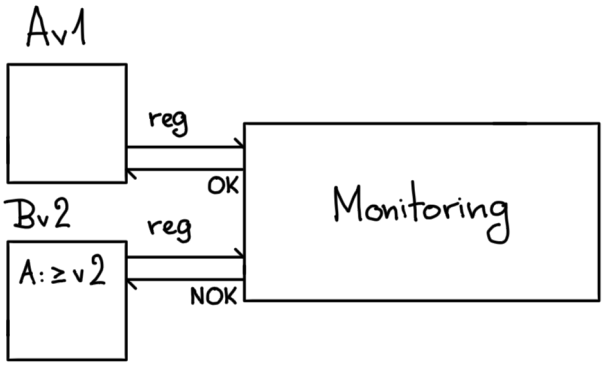
2.4.3 Sémantické verzování MS

Místo jednotlivých částí API je možné verzovat a hlídat samotné mikroslužby. Výrazně se tím zkomplikuje možnost udržování několika poskytovaných rozhraní, protože každá verze bude vyžadovat vlastní adresu v rámci systému. Na druhou stranu je možné u každé služby definovat seznam závislostí, který by se poskytoval v rámci obecného dotazu ohledně stavu služby. V případě existence konceptu registrace mikroslužby v systému by bylo možné tyto požadavky na závislosti kontrolovat a včas upozorňovat na nevhodnou sestavu mikroslužeb (viz obrázek 2.6).

2.4.4 Kalendářní verzování MS

Velice podobné sémantickému verzování zdrojů, neexistuje přesně definovaný formát, ale může se to vyplatit u pravidelně vydávaných aplikací (například na týdenní bázi). Využívá značky s uváděním roku, měsíce, dne a případně i času sestavení. V takových

případech je vidět zaostávání nasazování služeb (například kvůli chybám ve funkcionalitě) a je možné na to patřičně reagovat [24].



Obrázek 2.6: Registrace MS a kontrola závislostí

2.5 Testování a automatizace

Problematika testování architektury mikroslužeb se v základu ničím neliší od testování jiných typů aplikací – existují určité typy, které je nutné vhodně aplikovat na poskytnuté zdroje. Některé typy testů mohou plně podléhat automatizaci, něco bude vyžadovat dodání doplňujících materiálů a něco je lepší testovat manuálně kvůli lepším výsledkům testů. V rámci testování MSA můžeme brát jako základ například následující kategorie:

Statické testování kódu – nevyžaduje běh aplikace, jde například o testování kvality kódu, stylizace apod. [26]. V Node.js se o tuto část může starat `eslint` (pravidla psaní kódu) a `prettier` (formátování).

Jednotkové testování – testování nejmenších částí implementace (funkce aj.) [27].

Integrační testování – testování integrací v systému [27]. V rámci mikroslužeb můžeme testovat integraci jak mezi nejmenšími jednotkami, tak i brát celou MS a testovat její integraci se zbytkem aplikace.

Funkční testování – testování funkčnosti bez znalosti interní implementace [28], například dle testovacích scénářů. Zde se může jednat třeba o otestování jednotlivých rozhraní v rámci spuštěné aplikace, nebo navazující sekvenci takových volání.

Testování spolehlivosti – testování způsobu chování v případech přetížení/výpadku a schopnost se obnovit po výpadku [29].

Testování zátěže – testování výkonu aplikace s pomocí simulace aktivity systému [30].

Existují i další typy testů, jež se mohou provádět nad MSA, vždy je nutné volit to, co je pro vyvíjený systém potřebné. V ideálním případě je snaha mít každý automatizovaný typ testování automaticky spouštěný ve správnou chvíli během vývojového cyklu a nasazení sytému.

2.6 Nasazování

Nasazení mikroslužeb je jedna ze závěrečných fází vývoje systému, pokrývá široké spektrum možností – rozmístění produkčních balíčků na samostatných fyzických serverech, cloud-řešení, v klasterech, s vyvažováním zátěže, replikací apod. Daná podkapitola se bude věnovat pouze společným rysům – přípravě pro všechny typy nasazení – a kontejnerizací, jež je vyžadována zadáním diplomové práce.

2.6.1 Správa zdrojového kódu

Služba, případně mikroslužba, z hlediska chování byla v této práci definována jako samostatný, atomicky fungující celek. Toto se týká i existence spuštěné instance – musí být schopna existovat samostatně (alespoň z hlediska propojení s ostatními službami). Taková vlastnost má důležitý dopad na potřebné horizontální škálování a rozložení zátěže v nejvyužívanějších částech systému [31].

I přes takové předpoklady samotnou správu zdrojového kódu je možné uspořádat jak do jednoho (monorepozitář) tak i více repozitářů (za předpokladu, že se využívá VCS git). Oba přístupy mají své výhody a nevýhody.

Monorepozitář – existence jednoho repozitáře, kde jednotlivé služby jsou umístěny do vlastních složek.

Výhody

- Celý produkt je uchováván na jednom místě - pro tým to může znamenat lepší podmínky pro testování a vývoj, jelikož mají dostupné všechny vyvíjené částí všemi týmy [31].
- V případě refaktorování nebo testování systému je možné vše provádět na jednom místě [31].
- Vývojové prostředí je možné nastavit tak, že všechny služby budou sdílet stejnou konfiguraci (kontrola kvality kódu, formátování a další) [31].

Nevýhody

- Obrovské množství služeb v jednom repozitáři může znepřehlednit vývoj – git historie se týká nejen jednoho produktu, ale všech – značky a větve budou sdílené pro všechny služby.

- Existence možného zásahu do zdrojového kódu služeb vývojáři, jež pro to nemají formální oprávnění [31].

Více repozitářů – každá služba má vlastní repozitář, který je naprosto soběstačný.

Výhody

- Přesné rozdělení vývojových částí a přístupů mezi jednotlivými vývojovými týmy [31].
- Přehledná git historie pro každou službu včetně značek.

Nevýhody

- Absence jednoduchého spuštění všech částí projektu [31].
- V případě existence sdílených zdrojových kódů, je nutné tuto situaci řešit jinými způsoby, než extraci do sdíleného prostoru nadsložek.

Ze subjektivní zkušenosti je do tohoto přehledu možné přidat ještě jeden způsob vedení projektu. Spočívá v kombinaci obou přístupů s využitím `git submodules`. V takovém případě struktura projektu je organizována následovně:

- Každá služba má svůj vlastní repozitář a jejich vývoj je nezávislý.
- Existuje jeden repozitář, který importuje všechny služby jako git submoduly a přidává vhodnou konfiguraci vně submodulů (například `docker-compose`, testy apod.).

Takový přístup přináší některé výhody monorepozitáře, ale přístup k jednotlivým službám je samostatný a může být omezován právy pro každý repozitář. Tzn. v případě snahy o jednotnou statickou analýzu kódu (či obdobné záležitosti) je možné přesunou konfigurace ze všech repozitářů mikroslužeb do společného repozitáře se submoduly a odkázat původní konfigurace na konfiguraci v nadsložce. Tímto zásahem se ale znemožní separátní vývoj a bude potřeba vždy stahovat hlavní repozitář a z toho submodul, nad kterým se má provádět vývoj.

2.6.2 Kontejnerizace

Kontejnerizace je jistý způsob virtualizace za použitím menšího množství systémových zdrojů, než u plnohodnotné virtualizace [32]. Základní myšlenka spočívá v přípravě soběstačného balíčku (obrazu) s programem, který by se dal spouštět jednoduchým vytvářením nové instance (kontejneru) v jakémkoliv prostředí, které poskytuje dostatečné základní rozhraní [33].

V případě JavaScript aplikací založených na Node.js prostředí a využitím služby Docker se bude jednat o následující proces:

Stažení výchozího prostředí – obraz prostředí, ze kterého bude mikroslužba vycházet, v tomto případě `node`.

Přidání npm/yarn závislostí – do obrazu jsou staženy všechny vnější závislosti.

Přidání a sestavení samotného projektu – do obrazu jsou přidány zdrojové soubory samotného programu a spuštěno sestavení.

Otevření potřebných komunikačních kanálů – mikroslužba komunikuje na vlastním, předem určeném portu. Po vytvoření instance tento port musí být otevřen pro komunikaci s kontejnerem.

Definování příkazu pro spuštění – definice sekvence příkazů, které během vytvoření kontejneru připraví a spouští příkaz pro start webového serveru.

Takto se musí připravit každá služba v rámci systému. Spuštění celého projektu následně může být řízeno `docker-compose`, který kromě definovaných mikroslužeb ještě poskytne databáze a další potřebné programy třetích stran.

2.7 Monitorování

Monitorování je důležitou součástí vývoje a přizpůsobování mikroslužeb a aplikací obecně. Dává přehled o stavu sytému a může napomáhat predikci například budoucí zátěže na základě historických dat. Sledované hodnoty můžeme rozdělit na tři hlavní skupiny [34]:

Metriky – měřitelné hodnoty latence, chyb, zátěže a saturace systému. Může se jednat například o počet požadavků na server, počet chyb, objemu předaných dat, pokusů o opakované zpracování informací, využitou paměť a další hodnoty [34]. Takové historické informace se následně mohou podílet na přizpůsobování sytému dle nároků a potřeb.

Logy – informace o uskutečněných událostech.

Stopy – detailní záznamy o chybách a jejich původu.

Na takovém rozdělení se zakládá například i populární [35] platforma pro vizualizaci metrik Grafana [36].

Monitorování v architektuře mikroslužeb je samostatná oblast s mnoha způsoby realizace, analýzy a zpracování. Vzhledem k širokému rozsahu nebude blíže zkoumána, až na návrhový vzor `Health Check`.

2.7.1 Health check

Vzor `Health Check` je jednoduchým nástrojem pro monitorování aktuálního stavu mikroslužby [37]. Jedná se o speciální API `GET` rozhraní, které poskytuje základní informace o stavu mikroslužby. Může se jednat například o název, verzi, stavu komunikce, stavu fyzického zařízení a další informace [37].

Takové rozhraní, i když poskytuje jednoduchou kontrolu, tak nese i nevýhodu v podobě nedostupnosti, pokud celá mikroslužba nefunguje.

2.8 Mikroslužby webové aplikace

Posledním tématem věnovaným obecné analýze MS je otázka mikroslužeb klientských webových aplikací. Ve webovém prostředí se většinou jedná o `client-server` komunikaci – existuje zdrojový server a koncový klient, který zobrazuje a zpracovává stahovaná data. Omezíme se na skupinu nejtypičtějších souborů – `.html` `.css` `.js` a binární data (obrázky, písmo, zvuk, videa). Obsah pro uživatele v takovém případě může být dělen následovně:

Statický obsah – statické soubory, které jsou stahovány klientem přes HTTP protokol během základního načtítání stránky[7].

Dynamicky generovaný obsah na straně serveru – například HTML šablony s možnými datovými sety (viz SSG).

Dynamicky generovaný obsah na straně klienta – například AJAX (asynchronous javascript and xml) načítání obsahu po prvotním načtení stránky.

V prvních dvou případech dělení na dříve definované služby (atomický celek s komunikací pouze přes strikně definované rozhraní, jenž může být vyvíjen nezávisle na ostatních částech aplikace) není úplně vhodné – všechno je ve výsledku opět řízeno serverem (logika sestavování šablon). V posledním případě však můžeme vytvořit celky, jež mohou vykazovat samostatné, pseudo-izolované chování, a jejich životní cyklus by byl plně řízen klientem.

Bohužel na rozdíl od serverových aplikací, kde pro každou část je dostupný vlastní proces, na klienstké straně existuje pouze jeden (pokud budeme brát v úvahu nejrozšiřenější prohlížeč Chrome [38]) pro každou stránku, který zpracovává všechny požadavky [39]. Separovat takové části úplně nelze - musíme počítat se společným

[7] Základním načtením stránky je zde myšleno načtení dat bez interpretace JavaScript souborů, které mění stránku dodatečně (a obdobných technologií)

prostorem pro vykonávání JavaScript souborů a společné CSS (cascading style sheet) styly.

V rámci klientské aplikaci proto musíme předefinovat službu jako samostatný celek, komunikující pouze přes přesně dané rozhraní, vlastnící separovanou část datového úložiště, ale sdílící společné prostředky.

Pokud pomineme existenci `<iframe>` tagu v HTML, který do jisté míry vyhovuje, ale přináší i jistá omezení [40], tak se nabízí možnost definovat menší logické balíčky, stahované například přes AJAX dle vyžádání po načtení stránky. Vývoj daných balíčků by byl řízen určitými pravidly, při dodržení kterých by mohly být opakovaně používány na stránce nebo dokonce ve více aplikacích.

I když existence takových celků je možná, tak stále chybí jejich inicializace na stránce. Z tohoto důvodu se nelze vyhnout implementaci obecného monolitu, jenž by se načítal jako inicializátor obsahu. Hlavním úkolem by pro něj bylo řízení načítání a interpretace balíčků dle určitých příznaků (například URL adresa, nebo vlastní směrovací systém). V případě potřeby, by měl vytvářet a poskytovat komunikační kanály mezi samostatnými celky.

Ve výsledku se balíčky velice podobají chováním všemožných pluginů, jen se striktním omezením z hlediska integrace s prostředím.

Na základě obdobných úvah existuje manuál Micro Frontends, který realizuje myšlenku s pomocí základních JavaScript možností, nabývá však i svých omezení a nevýhod [41].

Kapitola 3

Správa dat v architektuře mikroslužeb

V dané kapitole:

- způsoby integrace datových zdrojů do MSA,
- spojování dat napříč mikroslužbami,
- transakční zpracování napříč mikroslužbami.

Databáze jako datový zdroj je jedním z možných řešení pro persistentní ukládání dat s případným následným zpracováním. Daná kapitola je věnována především dvěma návrhovým vzorům pro organizaci dat v relačních databázích dodržujících ACID (atomicity, consistency, isolation, durability) vlastnosti a spolupráci mikroslužeb pro vyhodnocování pokročilých klientských požadavků. Popisované databázové organizace dat se obecně nemusí týkat pouze architektury mikroslužeb, ale nachází zde svoje přímé uplatnění.

3.1 Typy organizace zdrojů

V jednoduché monolitní architektuře využití relační databáze může být zcela přímočaré – jeden monolit se připojuje k jedné databázi, která spravuje veškerá potřebná data. Mikroslužby však představují separaci na jednotlivé části dle zvolené dekompozice a intuitivně přináší i myšlenku dělení původně jednoho datového zdroje na obdobně dekomponované celky. Na základě této úvahy můžeme definovat dva návrhové vzory pro organizaci dat:

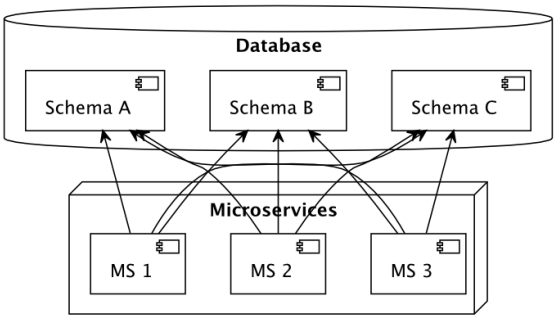
Sdílená databáze – jedna instance databáze je plně přístupna pro všechny připojené mikroslužby [42].

Databáze pro každou službu – každá deklarovaná mikroslužba používá vlastní databázi (nebo schémata), do které má výhradní přístup [43]. V takovém případě se samozřejmě může jednat i o fyzicky samostatné databázové servery.

Oba přístupy mají svoje využití dle poskytovaných vlastností.

3.1.1 Sdílená databáze

Sdílená databáze poskytuje všechna svoje úložiště všem mikroslužbám, možná realizace je znázorněna na obrázku 3.1. ACID vlastnosti zde jsou řízeny databází a logicky se ničím neliší od komunikace s monolitickou aplikací.



Obrázek 3.1: Architektura sdílené databáze

Výhody

- Typické použití ACID pro provádění transakcí [42].
- Správa jedné databáze je jednodušší [42].

Nevýhody

- Databázové migrace se netýkají pouze jedné mikroslužby, ale všech zároveň, musí se vyčlenit místo pro jejich uchovávání.
- V případě jakékoliv změny struktury databáze je třeba změnit a otestovat všechny mikroslužby [42].
- Jednotná databáze nemusí vyhovovat potřebám všech mikroslužeb [42].

3.1.2 Databáze pro každou službu

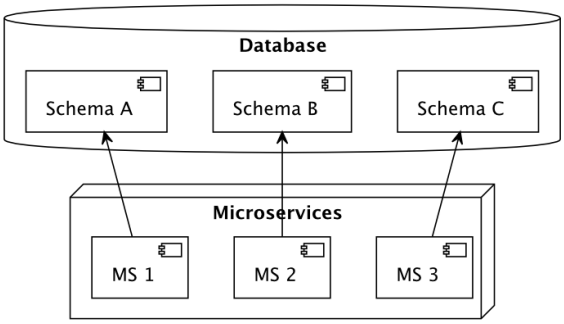
V případě vyčlenění jednoho schématu, či celé databáze pro každou mikroslužbu, můžeme mluvit o druhém typu struktury využití relačních databází, možná realizace je znázorněna na obrázku 3.2.

Výhody

- Zmenšení provázanosti mikroslužeb a dat, jež jsou využívány – lepší správa migrací a jiných změn [43].
- Lepší možnosti škálování a volby databáze – mikroslužbě je možné poskytnout takové prostředí, které bude nejvhodnější [43].

Nevýhody

- Problém se spojováním dat z více úložišť – nelze například provádět `JOIN`, párovat dle cizího klíče s kontrolou integritního omezení [43].
- Problém s transakčním zpracováním – transakční operace nad několika databázemi nelze jednoduše provádět a je lepší se jim úplně vyhýbat [43].
- Komplikovaná správa a konfigurace většího počtu databází [43].



Obrázek 3.2: Architektura databází (schémat) pro každou službu

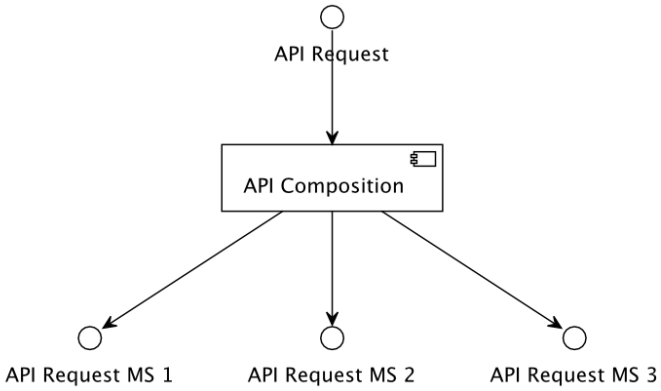
Nevýhody databází rozdělených dle mikroslužeb lze řešit s pomocí některých návrhových vzorů, jsou ve stručnosti popsány v následujících podkapitolách.

3.2 Spojování dat mezi mikroslužbami

Jako jeden z možných požadavků v rámci MSA je spojení dat mezi tabulkami, což v rámci oddělených služeb znamená netriviální úkol. Jako typický příklad lze vzít existenci tabulek `users` (osobní informace) a `payments` (složeno z částky a primárního klíče z `users`), jež se nachází v oddělených databázích, ze kterých je třeba získat seznam transakcí všech lidí s určitým věkem. Spojení v tomto případě nemůžeme nechávat na databázi, situace se může řešit API kompozicí nebo CQRS (command query responsibility segregation) přístupem.

3.2.1 API kompozice

API kompozice je řešení bez využití dodatečných systémů. Spočívá v logickém rozdělení původního dotazu na parciální dle jednotlivých mikroslužeb a následným provedení například simulace `JOIN` operace v aplikaci (v paměti) [44]. Schématicky je tato operace znázorněna na obrázku 3.3. Samozřejmě takové řešení může přinést větší nároky na paměť v případě pracování obrovských datových celků.



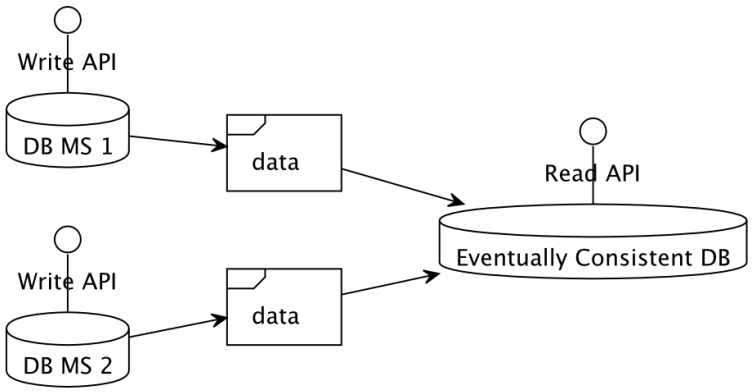
Obrázek 3.3: Schéma vzoru API kompozice

3.2.2 CQRS

CQRS vzor může sloužit jako řešení nedostatku paměti, případně výkonu, které vzniká operováním s daty v aplikaci. Absence možnosti spojení je tu řešena dodatečnou, eventuálně konzistentní databází [11], viz obrázek 3.4. Databáze slouží jako zdroj dat pro čtení, který obsahuje kopie původních dat a databázový pohled na strukturu potřebou pro provedení klientského dotazu. Existující relační databáze mikroslužeb fungují jako persistentní úložiště pro zápis, uchovávají si informace a posílají potřebnou kopii dat do databáze pro čtení [45]. Takovým způsobem se vytváří úložiště, které obsahuje minimální potřebné množství informací například pro operaci `JOIN`. Samozřejmě eventuální konzistenci takové databáze je nutné explicitně zajistit, aby nedocházelo ke ztrátě dat [45].

3.3 Transakční zpracování zápisů

V relačních databázích jsou transakce chápány jako logické celky pro čtení nebo zápis informací, které dodržují ACID vlastnosti [46]. Pokud existuje víc datových zdrojů a je třeba provést požadavek na zápis, jenž musí ovlivnit data v několika databázích, tak nastává problém neizolovanosti zápisu [11]. Během zpracování požadavku každá mikroslužba na základě komunikace dostane příkaz o zápisu a provede je v souladu



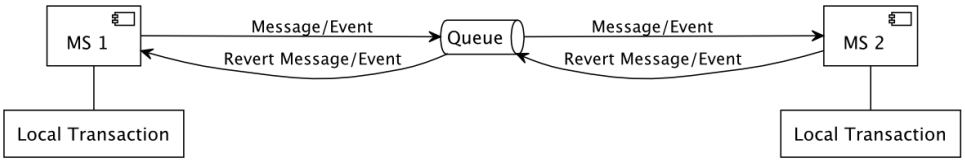
Obrázek 3.4: Zjednodušený model CQRS vzoru

s ACID, pokud však alespoň jedna z dílčích částí nebude úspěšná, tak `ROLLBACK` transakce bude uskutečněna pouze v chybné.

Takové situace se mohou řešit například s pomocí distribuovaných transakcí, nebo návrhového vzoru `saga` [11].

Distribuované transakce jsou metodou, která propojuje víc datových zdrojů (v tomto případě databází), zpravidla stejného typu, a během zpracování transakce zajišťuje ACID zápis napříč databázemi [47]. Dá se říct, že distribuce odstraňuje jedné transakci omezení z hlediska počtu databází prostřednictvím vhodné konfigurace [48].

Sága je realizována na straně serverové aplikace, jedná se o kontrolovanou sekvenci spouštění lokálních transakcí. Každá dílčí transakce po svém provedení posílá zprávu nebo vyvolává událost, jež spouští další transakci dle určeného pořadí. Pokud jedna z lokálních transakcí skončí chybou, pak obdobným způsobem se spouští sekvence zpětných transakcí všech předchozích úprav [49]. Tímto je docílena chybějící izolace napříč databázemi. Příklad takové struktury lze vidět na obrázku 3.5.



Obrázek 3.5: Koncept transakčního zpracování v ságách

Kapitola 4

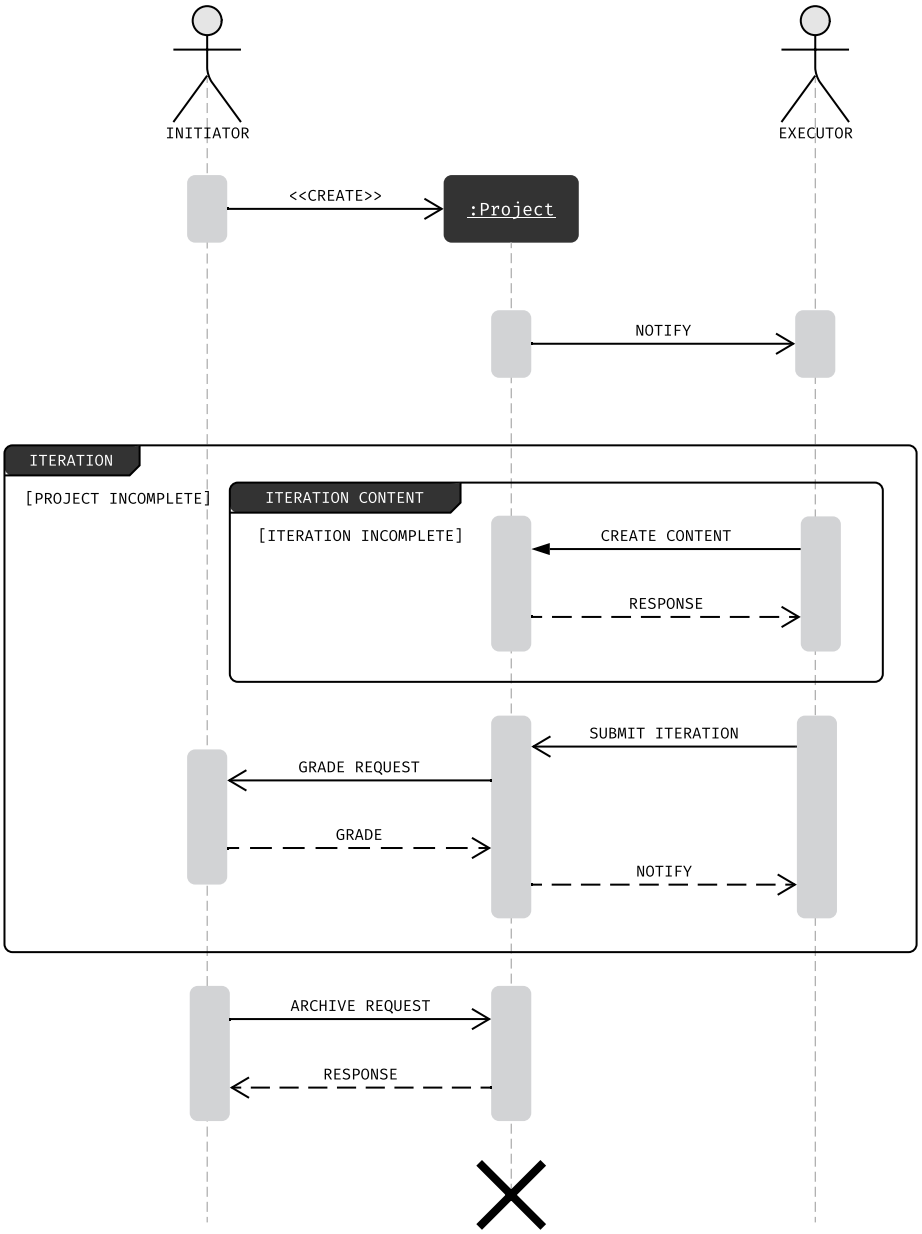
Specifikace nového systému

V dané kapitole:

- základní myšlenka IS,
- seznam povinných požadavků IS,
- seznam rozvojových požadavků IS.

Z hlediska specifikace za základ je považován popis IS definovaný v bakalářské práci v kapitole „Specifikace informačního systému“ [3]. Zachovává se hlavní myšlenka – iterativní zpracovávání studentských projektů na základě komunikace dvou subjektů (vedoucího a studenta), jež postupně tvoří obsah projektu (viz obrázek 4.1).

V tomto případě budou původní požadavky uměle upraveny dle aktuálních potřeb, aby korelovaly se zadáním diplomové práce, a budou rozděleny do dvou částí – povinné, které v plné míře poskytnou možnost realizovat přechod na MSA, a rozvojové, jež výhledově již nezmění architekturu implementovaného řešení a jen budou přidávat další funkcionality systému jako takového.



Obrázek 4.1: Zobecněný životní cyklus projektu [3]

4.1 Základní požadavky a případy užití

4.1.1 Funkční požadavky

FP00 Identita uživatelů – uživatelé v systému jsou jednoznačně identifikovány a mohou vykonávat vykonávat určité činnosti na základě přidělených práv.

FP00-UC00 Anonymní uživatel se může zaregistrovat.

FP00-UC01 Anonymní uživatel se může přihlásit.

FP00-UC02 Anonymní uživatel si může obnovit heslo dle emailu.

FP00-UC03 Přihlášený uživatel může vykonávat povolené operace dle práv.

FP01 Globální role – práva uživatelů jsou přidělována na základě rolí, administrátor systému má má neomezený přístup. Existuje 5 rolí: host (`guest`), uživatel (`user`), autorita (`authority`), administrátor (`admin`), zablokovaný uživatel (`banned`).

FP01-UC00 `admin` může měnit uživatelům role.

FP01-UC01 `banned` má právo pouze na autentizaci, prohlížení osobních dat a odtranění.

FP02 Uživatelská data – aplikace umožňuje uživatelům prohlížet si svoje data a v případě nutnosti odstranit referenci na jejich osobu (odstranit účet).

FP02-UC00 Přihlášený uživatel může prohlédnout svoje data uložená v systému.

FP02-UC01 Přihlášený uživatel může odstranit svůj účet.

FP03 Upozornění – aplikace posílá uživatelům relevantní upozornění na základě změn v IS.

FP03-UC00 `user` může označit upozornění jako přečtené/nepřečtené.

FP03-UC01 `user` může odstranit upozornění.

FP04 Projekt – životní cyklus – aplikace umožňuje uživatelům zakládat projekty s definováním detailních interních informací. Projekt může být následně upravován až do jeho zániku (permanentní odstranění), nebo pozastavení (archivování na dobu neurčitou).

FP04-UC00 `user` může založit projekt.

FP04-UC01 Pokud je projekt veden alespoň jedním `authority` uživatelem, tak je označen za důvěryhodný.

FP04-UC02 `user` s projektovou rolí `leader` může odstranit projekt.

FP04-UC03 `user` s projektovou rolí `leader` může archivovat projekt, nebo zrušit archivaci projektu.

FP05 Projekt – správa dat – aplikace umožňuje každému projektu definovat veřejná, skrytá (pouze pro účastníky projektu) a volitelně skrytá data.

- **Veřejná** – kategorie, název, veřejný popis, cleny týmu, status archivace.
- **Skrytá** – interní popis.
- **Volitelně skrytá** – vakantní pozice, obsah.

FP05-UC00 `user` s projektovou rolí `leader` může upravovat veškerá data.

FP05-UC01 `user` s projektovou rolí `leader` může upravovat viditelnost volitelně skytých dat.

FP05-UC02 `user` s projektovou rolí `contributor` může upravovat obsah.

FP05-UC03 `user` s projektovou rolí `visitor` může prohlížet skrytá data projektu.

FP06 Projekt – kategorie – aplikace umožňuje každému projektu definovat kategorie, dle kterých se dá vyhledávat.

FP06-UC00 `admin` může zakládat a upravovat kategorie.

FP06-UC01 `admin` může odstranit kategorii, pokud neobsahuje ani jeden projekt.

FP07 Projekt – tým a role – v rámci každého projektu každý uživatel má určitou roli, která mu přiděluje oprávnění pro prohlížení/úpravu projektu.

- **Vedoucí (** `leader` **)** – neomezený přístup k projektu, spravuje všechny detaily projektu. Prvním vedoucím se stává uživatel, jenž projekt založil.
- **Spolupracovník (** `contributor` **)** – podílí se na tvorbě obsahu projektu, nezasahuje do interní správy projektu.
- **Návštěvník (** `visitor` **)** – může prohlížet skryté detaily projektu.

Role Spolupracovník se nevyskytuje v projektu otevřeně, vedoucí projektu může definovat její podmnožiny, jako určité pozice v týmu s omezenou kapacitou.

FP07-UC00 `user` s projektovou rolí `leader` může povolit/zakázat projektovou roli `visitor`.

FP07-UC01 `user` s projektovou rolí `leader` může definovat role v týmu, které jsou podskupinou projektové role `contributor`.

FP07-UC02 `user` s projektovou rolí `leader` může povolit/zakázat nábor do týmu.

FP07-UC03 `user` se může přihlásit na projektovou roli, pokud je povolený nábor a je volná kapacita role.

FP07-UC04 Účastník projektu může tým opustit, pokud se nejedná o posledního uživatele s projektovou rolí `leader`.

FP08 Projekt – vyhledávání – aplikace umožňuje vyhledat projekt dle zadaných kritérií.

FP08-UC00 `user` s projektovou rolí `leader` může povolit nebo zakázat vyhledávání projektu.

FP08-UC01 `user` může vyhledat projekt dle kritérií, pokud je povoleno vyhledávání projektu.

FP09 Projekt – Správa obsahu – aplikace umožňuje vedoucím a spolupracovníkům tvořit obsah projektu. Obsah je členěn do částí – každá část má datovou složku a název interpretu, který bude danou datovou složku zpracovávat do vizuální podoby.

FP09-UC00 `contributor` nebo `leader` projektu mohou vytvářet, upravovat a odstraňovat obsah.

FP09-UC01 `admin` může přidávat a odebírat dostupné globální interprety.

4.1.2 Obecné požadavky

OP00 Veřejné API – aplikace bude nabízet veřejné i interní API s dokumentací pro vývoj mikroslužeb i klientských aplikací.

OP01 Dokumentace – součástí IS je vývojářská dokumentace.

OP02 Rozšiřitelnost – aplikace je přizpůsobena dalšímu rozvoji a umožňuje škálovat jednotlivé části funkcionality dle zátěže.

OP03 Uživatelské rozhraní – uživatelské rozhraní je ve webové podobě a podporuje prohlížeč Google Chrome. Mobilní rozhraní není potřeba.

OP04 Jazykové verze – rozhraní je připraveno pro překlad do dalších jazyků.

OP05 Úložiště dat projektů – úložiště dat projektů bude možné měnit před prvním spuštěním IS. Výměna úložiště se zachováním údajů během provozu IS není nutná.

4.2 Rozvojové požadavky a případy užití

FP10 Projekt – tagy – aplikace umožňuje každému projektu definovat veřejné tagy, dle kterých se dá vyhledávat.

FP06-UC00 Tag vzniká během založení nebo úpravě projektu.

FP06-UC01 `admin` může upravovat název tagu.

FP06-UC02 `admin` může odstranit tag, pokud ho nevyužívá ani jeden projekt.

FP11 Projekt – iterace a úkoly – projekt může být rozdělen na jednotlivé iterace, které se skládají z jednoho, nebo více úkolů. Jednotlivé části obsahu projektu mohou tyto úkoly označovat jako splněné.

FP09-UC00 `leader` projektu může založit projekt s iteracemi, nebo přidat iterace do existujícího projektu.

FP09-UC01 `leader` projektu může upravovat iterace a úkoly.

FP09-UC01 `leader` projektu může odstraňovat iterace a úkoly, pokud nejsou spjaty s existujícím ohodnocením.

FP09-UC02 Uživatelé tvořící obsah mohou označovat jednotlivé úkoly jako splněné u každé části obsahu.

FP12 Projekt – snímky iterací – stav obsahu projektu se dá zafixovat a kdykoliv prohlédnout. Tento stav bude neměnným historickým milníkem všech částí obsahu, u něhož je rovněž uložena informace ohledně autora snímku, splněných iterací a úkolů.

FP11-UC00 `leader` nebo `contributor` projektu může z aktuálního stavu projektu vytvořit snímek.

FP11-UC01 Účastník projektu může prohlédnout jakýkoliv historický snímek projektu.

FP13 Projekt – hodnocení iterací – odevzdané iterace (snímky iterací) se dají ohodnotit body. Každý úkol má minimální požadovaný a maximální počet bodů, kterými může být ohodnocen. Ke každému ohodnocení úkolu a iterace lze doplnit komentář.

FP11-UC00 `leader` nebo `contributor` projektu může u snímku vyžádat ohodnocení ze strany jednoho z uživatelů typu `authority` s projektovou rolí `leader`.

FP11-UC01 `authority` s projektovou rolí `leader` může ohodnotit odevzdanou iteraci.

FP11-UC02 `authority` s projektovou rolí `leader` může upravit svoje vlastní ohodnocení.

FP14 Projekt – Editace týmu – tým lze nabírat nejen z dobrovolníků, ale i striktním přiřazováním uživatelů.

FP11-UC00 `authority` s projektovou rolí `leader` může do týmu projektu přidávat účastníky bez jejich souhlasu.

Kapitola 5

Analýza a implementace serverové části

V dané kapitole:

- popis architektury serverové aplikace a metod zvolených pro realizaci,
- rozbor komplexnějších funkcionalit a aspektů systému.

Implementace serverové části na základě zadání a definované specifikace IS spočívá v anaýze aktuální monolitické architektury (z předchozí práce) a její přepis do MSA s využitím zkušeností získaných a popsaných v kapitolách „Obecný úvod do architektury mikroslužeb“ a „Správa dat v architektuře mikroslužeb“.

Ačkoliv původní implementace měla vyřešeno dost aspektů z hlediska funkcionality, přechod na novou architekturu způsobem využití existujících zdrojových kódů by odhadem zabral víc času, než nová realizace s pouhou inspirací z původního systému. Z tohoto důvodu bylo rozhodnuto napsat nový základ pro mikroslužby, který by víc vyhovoval aktuálním potřebám. Pro novou realizaci bylo rovněž vybráno jiné ORM (object relational mapping) – `prisma`, které přineslo zjednodušení vývoje za cenu úpravy implementace většiny funkcionalit spojených s relační databází, zejména funkcionalitou migrací. Detailní popis změn a přínosů je uveden v následujících podkapitolách.

5.1 Architektura

Základním krokem pro návrh architektury serverové části se stal výběr typu dekompozice poskytnuté specifikace. V rámci požadavků na možnou výměnu zdroje s projekty, nebo i budoucí přechod na jiný způsob autorizace, se zdála nejvhodnější dekompozice dle subdomén.

Dekompozice se provedla na následující mikroslužby:

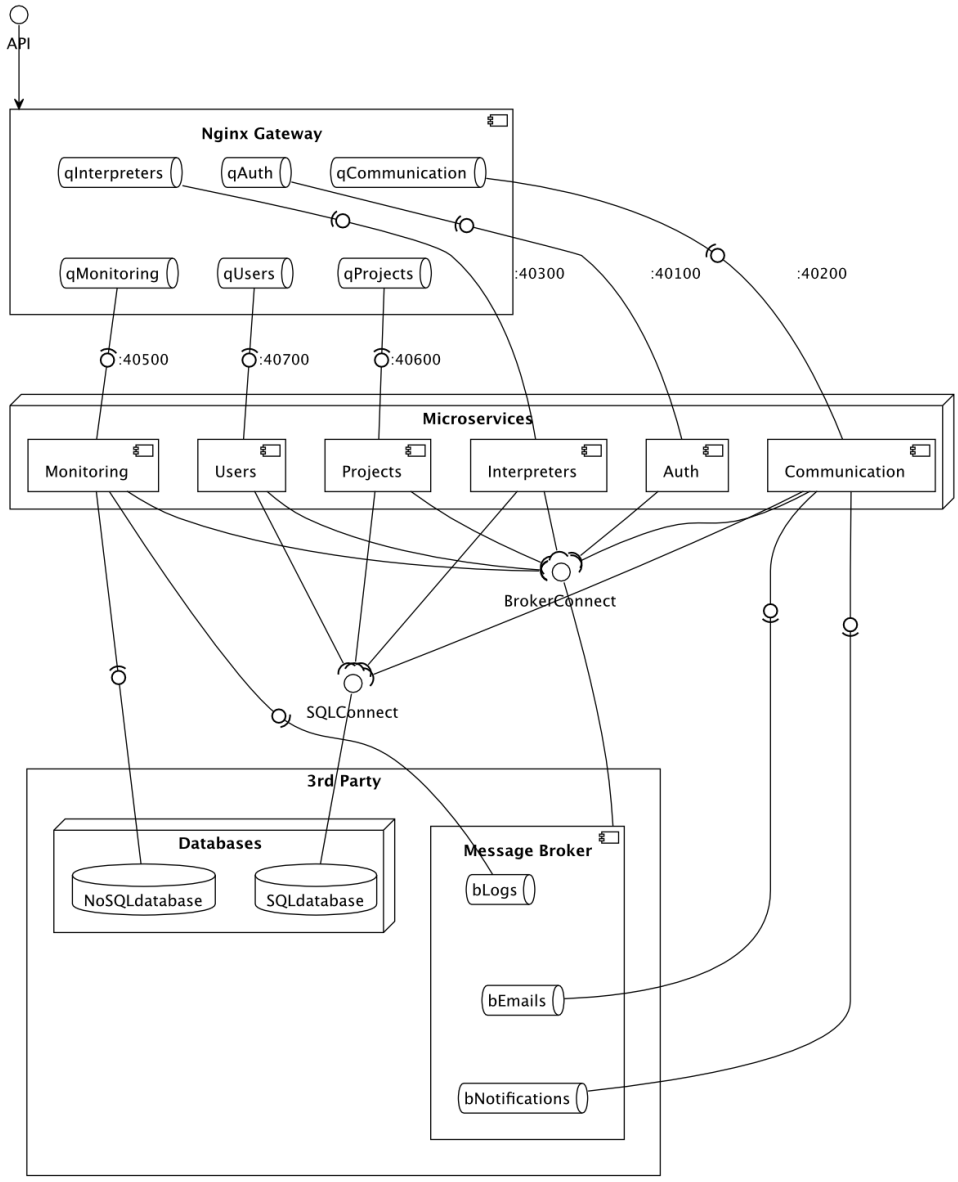
- **ms-users** – uživatelská data a vše, co je potřeba pro správu uživatelů.
- **ms-auth** – autentizační funkcionalita, kontrola autorizačních údajů a přístupových práv.
- **ms-projects** – služba pro správu projektových dat. Ukládá v relační databázi metadata o projektech a deleguje úpravy obsahů na vnější git úložiště.
- **ms-interpreters** – správa registrovaných interpretů a jejich verzí.
- **ms-communication** – zpracovávání komunikačních zpráv s uživateli, jako jsou e-maily a upozornění.
- **ms-monitoring** – služba pro centrální kontrolování stavů mikroslužeb a logování zpráv.

K tomu v rámci architektury byly přidány následující služby:

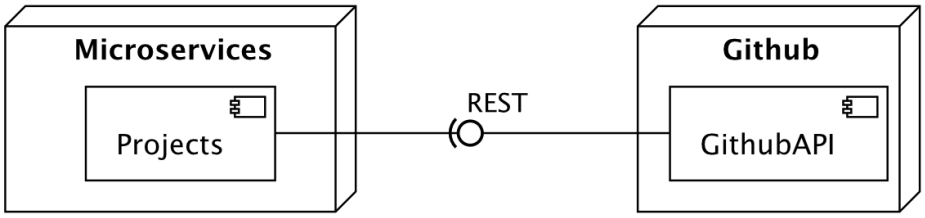
- **API Gateway** – implementace API Gateway fasády s pomocí `nginx`, která předává komunikaci klienta patřičným mikroslužbám.
- **SQL databáze** – SQL databáze v podobě `PostgreSQL` pro správu relačních dat.
- **NoSQL databáze** – NoSQL databáze v podobě `MongoDB` pro správu logovacích dat.
- **Message Broker** – `RabbitMQ` broker pro asynchronní komunikační kanály mezi mikroslužbami.

Komunikace všech těchto prvků je znázorněna na obrázcích 5.1 a 5.2.

Gateway v této implementaci rozděluje veškerou komunikaci se serverem na několik nginx streamů, které se rozpoznávají dle unikátních URI prefixů. Každý stream komunikuje právě s jednou mikroslužbou přes vlastní, unikátní port (viz obrázek 5.1). Mikroslužby jako takové jsou organizovány choreografickým způsobem a mohou volně volat rozhraní jiných mikroslužeb v rámci sítě. Pro danou komunikaci byl zvolen typ synchronní komunikace přes REST rozhraní, jako nejméně náročný pro implementaci. Výjimku však tvoří speciální případy, které vyžadují jistotu doručení zprávy, jedná



Obrázek 5.1: Architektura serveru



Obrázek 5.2: Vazba mikroslužby na Github

se o logování, odesílání e-mailů a odesílání důležitých notifikací konkrétním účtům. V těchto případech je použit broker zpráv, který funguje jako pojistka, že se zpráva neztratí v případě přetížené nebo nedostupné mikrošlužby. Je využíván princip asynchronní komunikace producent-konzument, tzn. jakmile se objevuje služba, která uložené zprávy může zpracovávat, tak ji jsou předány. V aktuální implementaci takové role zastupují mikroslužby `ms-communication`, která odbavuje e-maily a upozornění, a `ms-monitoring`, jež se stará o logování.

Z hlediska úložišť některé mikroslužby jsou napojené na SQL a NoSQL databáze, speciální případ tvoří však mikroslužba `ms-projects`, která vyžaduje existenci Github organizace s přístupovými právy pro správu repozitářů, aby v nich mohla uchovávat existující uživatelské projekty (viz kapitola „Správa projektů v git repozitářích“).

5.2 Struktura repozitářů

V rámci volby způsobu uspořádání repozitářů byl vybrán kombinovaný způsob popsaný v kapitole „Správa zdrojového kódu“. Každá mikrozlužba získala vlastní repozitář, jenž by mohl být vyvíjen naprosto separovaně. Zároveň se ale vytvořil repozitář, který definoval všechny mikroslužby jako git submoduly a přidal k tomu samostatně složku s Gateway, skripty pro rychlou tvorbu potřebných `.env` konfigurací ve všech mikroslužbách a `docker-compose` soubory pro vývojové a produkční prostředí.

Takové řešení poskytlo možnost instantně získávat téměř hotové (až na konfigurace) prostředí, navíc s variantou odděleného spuštění persistentních prvků (typu databází). Věškerý vývoj IS pak probíhal způsobem, že na externím serveru byly spuštěné databáze, Gateway a broker, které se mohly připojit k lokálnímu stroji pro vývoj, jenž byl zatížen pouze potřebnými prvky - mikroslužbami a klientskou aplikací.

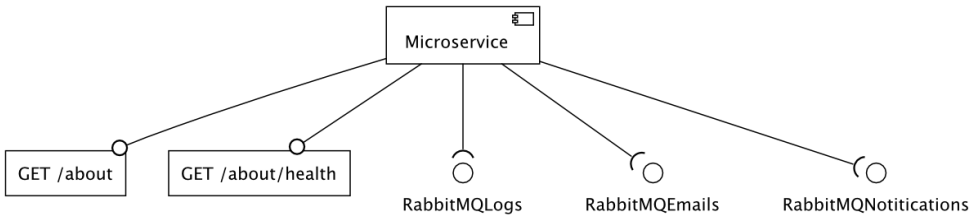
5.3 Šablona pro mikroslužbu

Všechny mikroslužby v architektuře sdílí určité chování a poskytované rozhraní, sloužící pro dotazování na stav mikroslužby. Kvůli takové konzistenci vznikl speciální repozitář, který udržuje základní šablonu, jež zahrnuje:

- `HTTP GET /about` – získání základních metainformací o službě (název, verze, vyžadované závislosti apod.).
- `HTTP GET /about/health` – primitivní kontrola dostupnosti mikroslužby, v případě dostupnosti vrací `HTTP 200`.

- `RabbitMQ log` – tvorba RabbitMQ zprávy s log zprávou a odeslání do brokeru.
- `RabbitMQ email` – tvorba RabbitMQ zprávy s email zprávou a odeslání do brokeru.
- `RabbitMQ notification` – tvorba RabbitMQ zprávy s upozorněním a odeslání do brokeru.

Na obrázku 5.3 lze vidět vizuální znázornění standardizovaného rozhraní:



Obrázek 5.3: Standardizované rozhraní mikroslužeb

Kromě rozhraní šablona stanovuje základní konfigurace pro statickou analýzu kódu, sestavovací nástroj, sdílené `.env` proměnné a další věci. Nevýhodou však je, že se tato šablona musí manuálně udržovat a jakoukoliv změnu je nutné kopírovat do ostatních repozitářů.

5.4 Správa dat a databáze

Informační systém využívá dvou typů databází pro uchovávání persistentních dat – SQL a NoSQL. MongoDB, jakožto zástupce NoSQL databáze, byla zvolena pro uchovávání logů, jež mají datovou složku, která vystupuje jako dynamická část dokumentu. Vzhledem k pravděpodobnému budoucímu vyhledávání, případně indexaci části takových dat, dané řešení je považováno za přiměřeně dobré. Pro ukládání striktně strukturovaných dat, byla zvolena PostgreSQL databáze s uplatněným vzorem využití jedné databáze (schématu) pro každou mikroslužbu zvlášť. Tuto volbu nejvíc ovlivnila potřeba tvorby další mikroslužby s migracemi, pokud by se použila jedna sdílená databáze. Jak se zjistilo později, tak i největší problém takto rozdělených datových zdrojů – konzistence při transakčním zpracování – mohl být díky vhodně zvolené struktuře dat eliminován.

V důsledku takto zvoleného uspořádání každá mikroslužba se samostatně stará o data a strukturu vlastní části – v produkčním nasazení před spuštěním webového serveru kontroluje a případně aplikuje chybějící migrace, aniž by ovlivnila zbytek sys-

tému. Jakékoliv přistupování k datům je rovněž komunikováno pouze přes mikroslužbu, ke které data patří.

5.4.1 Správa projektů v git repozitářích

V původní realizaci IS správa projektů byla implementována jako systém se sadou dat v dokumentové databázi. Z konceptuálního hlediska to bylo relativně racionální řešení – obsah projektu je JSON struktura, jež by se dala brát jako plnohodnotný dokument s dynamickou strukturou. CRUD (create, read, update, delete) operace nad každou částí byly intuitivně prováděny, jistá nedokonalost však nastávala během odevzdávání iterace projektu. V takovém případě bylo nutné vytvořit úplnou kopii všech částí projektu, aby se pro konkrétní odevzdání zachoval určitý stav. To vedlo k celkem nepříjemnému problému zvětšování objemu databáze.

Z tohoto důvodu se vytvořil koncept ukládání obsahů projektů v samostatném git repozitáři, kde místo úplného kopírování částí projektu by se dala uchovávat postupná historie tvorby projektu. Tím by se zamezilo zbytečné kopírování dat.

Se založením nového projektu v IS se tudíž zakládá i nový repozitář na Github, který byl zvolen kvůli poskytovanému API jako git server (může být vyměněn za jakýkoliv jiný). Párování mezi daty se základními informacemi o projektu (název apod.) a daty repozitáře se zajišťuje s pomocí jednotného UUID (universally unique identifier), který je použit jako název repozitáře a zároveň uložen v databázi projektu. Přístup k takovému repozitáři má pouze IS, nikoliv běžní uživatelé.

Vytvoření nové části obsahu projektu znamená vytvoření záznamu v relační databázi párovaného na nově založený soubor v repozitáři. Obsah souboru je postupně tvořen uživateli a ukládá datový JSON soubor, který je předáván z klientské části. Aby nenastávaly konflikty v případě úpravy jedné části více uživateli je využíváno Github API, které pro každou aktualizaci souboru vyžaduje `md5` klíč předchozí verze souboru [50].

Výše popsaná funkcionalita již byla realizována a ověřena z hlediska fungování. Co se týče zbývající funkcionality – odevzdávání zafixovaného stavu částí projektu – tak by se mohla řešit s pomocí git značek (`tag`). Jelikož úpravy projektu představují lineární historii, tak během přípravy náhledu odevzdávaného stavu by se samostné odevzdávání mohlo vázat ne na git větev, ale na určitý `hash` . Takové odevzdání iterace by v tom případě znamenalo kromě vytvoření záznamu v databázi také tvorbu git značky, opět párovaného na záznam dle UUID. Pro zobrazení obsahu by se pak využívalo stahování dat částí z repozitáře v určitém historickém stavu.

Kapitola 6

Analýza a implementace klientské části

V dané kapitole:

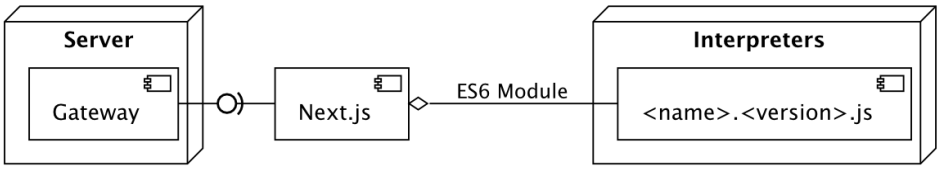
- popis architektury klientské aplikace,
- rozbor implementace interpretů obsahů,
- rozbor komplexnějších funkcionalit a aspektů realizace.

Klientská část aplikace ve srovnání s původní implementací z konceptuálního pohledu nepotřebuje výrazné zásahy nebo změny, až na způsob vykreslování obsahu projektu. Vzhledem k zastaralým závislostem projektu bude vhodné obnovit verze knihoven kvůli dostupnosti nových funkcionalit. Většímu zásahu bude podroben autorizační systém (vzhledem k přechodu na vlastní způsob autentizace). UX návrh bude z větší části převzat z předchozí práce. Drobné změny klientské implementace nebudou explicitně zmíněny.

6.1 Architektura

Po obecném prozkoumání problematiky mikroslužeb na klientské straně v kapitole „Mikroslužby webové aplikace“ lze udělat závěr, že přechod na plnohodnotné mikroslužby u tak omezené aplikace není výhodný. Jediným problémovým místem je realizace interpretů obsahů projektů, u které se předpokládá časté obnovování a rozšiřování funkcionality.

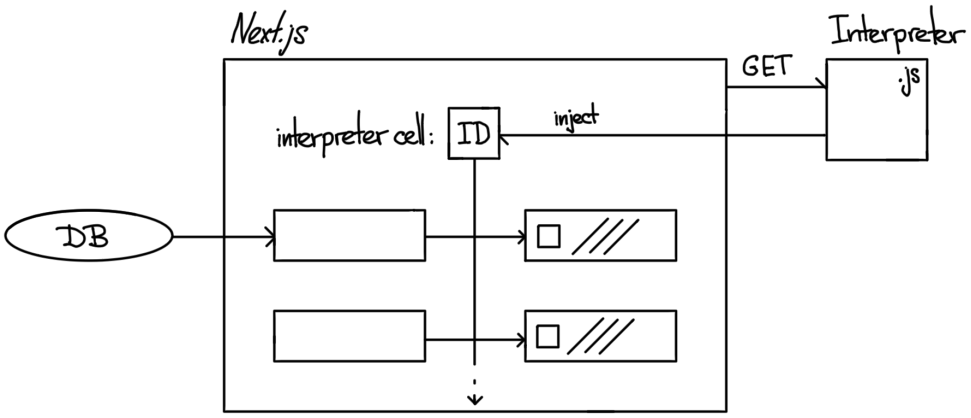
Základ architektury (obrázek 6.1) je tudíž převzat z bakalářské práce – řešení založené na frameworku Next.js s využitím REST přístupu k serveru. Obsah projektů i nyní bude tvořen s pomocí načítání externích `.js` souborů, způsob integrace však odlišný bude.



Obrázek 6.1: Architektura klientské aplikace

6.2 Interprety obsahu

Interprety obsahu tvoří jednu z těch více implementačně zajímavých částí klientské aplikace. Dá se říct, že se jistým způsobem pokouší napodobit způsob fungování mikroslužeb na straně klienta, i když věcně se spíš chová jako doplněk. V původní práci se jednalo o `.js` soubory, jež se v případě potřeby, která nastávala s prvním výskytem obsahu s vazbou na interpret, stahovaly z externích zdrojů a automaticky pouštěly. Byly tvořeny IIFE (immediately invoked function expression) funkcí, která do globální proměnné prostředí přidávala funkci renderování obsahu pod svým názvem. Když se objevoval další obsah s vazbou na stejnojmenný interpret, tak se kontrolovalo, zda renderovací funkce již existuje a opakovaně se pouštěla s informacemi dodanými ze serveru. Výsledkem se očekával textový HTML řetězec, jenž by se mohl vložit do DOM (document object model). Vizualizaci daného způsobu fungování lze vidět na obrázku 6.2.



Obrázek 6.2: Původní způsob tvorby obsahů s pomocí interpretů

Takový způsob nekladl žádná omezení z hlediska vybraných technologií pro interpret, ale počítal s jednoduchým vstupem (pro tvorbu jedné části obsahu se používalo jedno políčko textového obsahu) a vyžadoval validní HTML (hypertext markup language) na výstupu v textové podobě. Po vývoji několika ukázkových interpretů bylo jasné, že takový vývoj nebude zdaleka pohodlný a intuitivní.

Jako řešení se spíš nabízelo vytvořit jednoduchý interpret, který by poskytoval vestavěnou strukturu typu `iframe`, která by vedla na samostatný zdroj. Tímto by se vyřešila integrace všech typů obsahů a vývoj by závisel pouze na externím dodavateli. Po vyzkoušení implementace daného typu bylo jasné, že by se vyřešil i bezpečnostní problém vložených obsahů a separace prostředí. Nevýhodou byla pouze nutnost existence vnějšího serveru, co by takový obsah poskytoval, což nevyhovovalo očekáváním.

Na základě takových požadavků se stanovil nový cíl který měl splňovat následující kritéria:

- Interpret musí být spouštěn v instanci stejné stránky, jako je samotný klient, i když se tím sníží úroveň bezpečnosti.
- Snížená bezpečnost může být kompenzována dotažením pouze ze známých zdrojů a zajištěním větší separace od prostředí.
- Interpret musí být vyvíjen v novodobých knihovnách, nejen jednoduchý HTML, aby vývojáři nemuseli obcházet nepohodlný vývoj.
- Tvorba dat pro interpret by měla mít formu komplexnějšího formuláře.

Dané tři body ve spojení s již existující Next.js aplikací přivedly na myšlenku vestavění cizí, samostatně běžící aplikace do DOM. Jinými slovy interpret by mohl být vyvíjen jako například samostatná React, nebo Vue aplikace a výsledek knihovny by byl rendrovaný do přesně určeného DOM elementu.

V takovém případě by se během vykreslování obsahů se Next.js aplikace starala o stažení potřebných interpretů, jako v případě původní implementace, tvorbu `div` elementu s unikátním ID (identity) a předávkou daného ID stažené aplikaci pro renderování poskytovaného obsahu. Tudíž veškerá logika by se přesunula na JS kód poskytovaný interpretem. Nevýhodou takového řešení by se však nejspíš stala velká spotřeba paměti a zpomalení aplikace jako celku, pokud by se předpokládalo, že jeden obsah projektu měl 50 částí a každá by vedla na novou puštěnou instanci knihovny. Zároveň není jisté, jak by se chovaly shodné knihovny z hlediska konfliktů mezi sebou a s poskytovaným Next.js prostředím, který by nadále počítal s vlastní správou DOM.

Jako řešení takové situace se nabídla redukce rozmanitosti technologií a využití toho, co již existuje – instance Next.js React, a principu, který se uplatňuje například ve Webpack – rozdělování sestavování aplikace na dynamické podmoduly, které se s pomocí běžné JS funkce pro dynamický import (`import('url')`) stahují v pozdějších fázích běhu aplikace. Interpret by tudíž měl být psaný jako jednotná React komponenta s neomezeným využitím možnosti knihovny a exportovat tuto komponentu v transpilované podobě přes ES6 (ECMAScript 6) modul. Obdobně, druhým exportem by se mohla definovat proměnná pro popis struktury formuláře nutného pro úpravu obsahu, který by sloužil jako počáteční `props` do komponenty.

Během implementace se však vyskytl problém s pouštěním komponenty v existujícím React kontextu, externí komponenta po sestavení nemohla dostat přístup k Reactu jako takovému. Proto bylo nutné odstoupit od myšlenky samostatného, běžného sestavování React aplikace a přenést vývoj do manuální podoby. Pro vývojáře způsob ověřování funkčnosti interpretu se nezměnil – stále se jedná o komponentu v rámci samostatné React aplkace, sestavování ale vyžaduje psaní veškerého potřebného kódu v jednom souboru, manuální transpilaci například přes nástroj `SWC` (či jakýkoliv jiný, podmínkou je schopnost sestavování React aplikací s `JSX` syntaxí) a obalení výsledku funkcí, která dodává React kontext komponentě (předávání React proměnné jako argument).

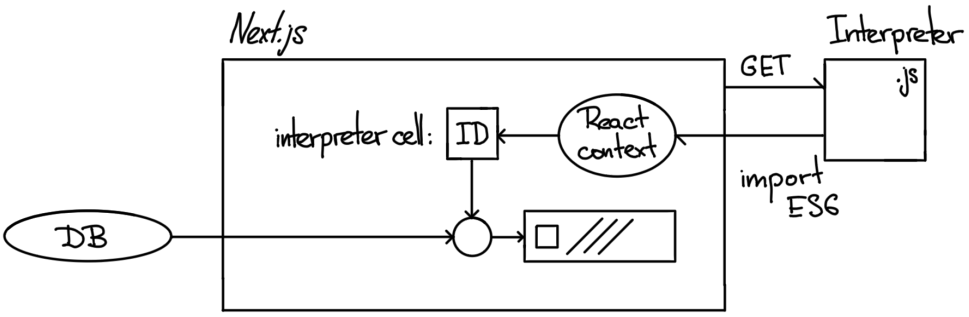
Takto připravená komponenta je ve výsledku načítána do globální proměnné Next.js aplikace a vystupuje jako standardní prvek systému. Samozřejmě takovým propojením vniklo riziko narušení funkčnosti původní aplikace, proto je důležité se k interpretům chovat jako k součástí aplikace, i když je slabě provázaná.

Takový přístup pro tvorbu interpretů byl ve výsledku zachován a byla napsána jednoduchá stavová komponenta pro vyzkoušení daného konceptu. Dle očekávání, vznikl problém s izolací CSS, protože styly využívané s aplikaci se používaly jako výchozí pro prvky samotného obsahu. Takovému vlivu se dá zamezit použitím například technologie CSS modulů, ale dané téma se ponechává pro další rozvoj aplikace.

Schématický výsledek fungování nové implementace interpretů je možné vidět na obrázku 6.3.

6.3 Autentizace

Větší rozsah změn se dotknul autentizace a autorizace uživatelů v aplikaci. Částečně přizpůsobený OAuth 2.0 se nahradil, nezávislou implementací „access“ a „refresh“ tokenů pro přístup k zabezpečeným zdrojům. Zjednodušil se tím postup registrace



Obrázek 6.3: Nový způsob tvorby obsahů s pomocí interpretů

a přihlašování a eliminoval uživatelky nepříjemný manuální přechod mezi záložkami kvůli synchronizaci dat. Za veškerou logiku získávání a obnovování tokenů nyní je zodpovědný tzv. `axios interceptor`. Sekvenční diagram celého průběhu dotazování na zdroje systému s obnovou tokenů je možné vidět na obrázku 6.4.

6.4 Jiná zdokonalení struktury a funkcionality

Kromě dvou zásadních změn (systém interpretování obsahu a autorizace) v klientské aplikaci došlo k následujícím úpravám:

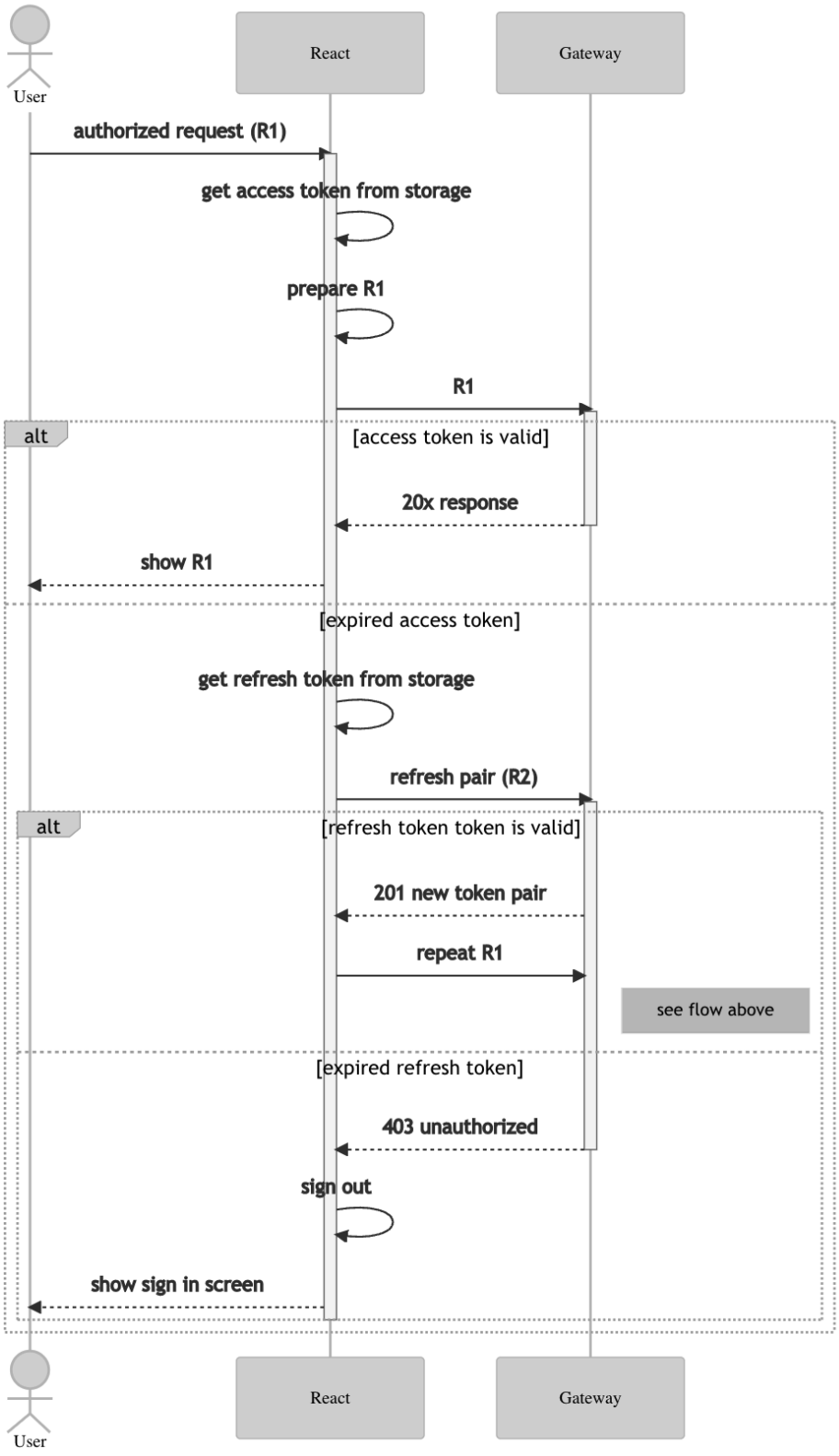
Aktualizace závislostí – aktualizace vyřešila problémy s instalací vyžadovaných npm balíčků a poskytla nové možnosti pro vývoj.

React hooks – přechod na modernější způsob zápisu React komponent s využitím „React hooks“ nebyl nutný, nicméně v rámci přepisování větší části klientské aplikace bylo rozhodnuto použít nový způsob zápisu. Přineslo to (v porovnáním s původním kódem) výrazné snížení objemu kódu a zlepšení přehlednosti.

Nepříznivé scénáře API dotazů – zpracování API dotazů bylo přepsáno na dotazování s pomocí obalů pro `axios` knihovnu s využitím `React-Query`, jenž odpovídá za stav požadavků. To poskytuje možnost definování globálního zpracování chyb ze serveru a v případě speciálních situací zobrazení patřičného uživatelského oznámení.

Vizuální vzhled rozhraní – vizuální rozhraní bylo částečně upraveno pro lepší UX.

Další změny byly menšího charakteru, rovněž však došlo k redukci z hlediska responzivního návrhu – dle nové specifikace nebyl kladen důraz na zařízení s menšími obrazovkami, aktuální řešení je přizpůsobeno pouze pro větší rozlišení, nicméně využití Bootstrap knihovny může pomoci vyřešení tohoto problému. Daná skutečnost byla přidána jako bod v možném budoucím rozvoji systému.



Obrázek 6.4: Autorizovaný požadavek a obnovení páru tokenů

Kapitola 7

Testování

V dané kapitole:

- krátké zdůvodnění vybraných typů testování IS,
- popis automatizace testování v projektu.

Vzhledem k méně komplexní struktuře IS – malý počet mikroslužeb s poskytováním pouze REST rozhraní a jednoduchý základ klientské aplikace – příliš detailní testovací metody nejsou vhodné, zvětšovaly by časové náklady, ale nepřinášely žádnou výhodu. Veškeré testování se proto omezilo na následující způsoby:

Statická analýza kódu – s pomocí nástrojů `eslint` a `prettier` byla zajištěna konzistence vzhledu a metodika psaní implementovaných částí. Kontrola se týká především `.ts` a `.js` souborů.

Jednotkové testování – ačkoliv takových testů není hodně, vzhledem k malému počtu samostatných funkcionalit vyčleněných do funkcí, pro každou mikroslužbou je nastaven `Jest` framework s možností kontroly pokrytí kódu testy.

Funkční testování – pro funkční testování jsou poskytovány automaticky vytvářené testovací účty uživatelů s různými rolemi. Testy jsou realizovány ve formě jednotlivých testovacích scénářů s pomocí `Jest` frameworku. Taková forma testů má za úkol testovat přístupová práva jednotlivých přístupových bodů, validace dat a sekvence volání REST rozhraní nad celým systémem.

Za potenciálně vhodné, ale nevyužité, jsou považovány následující testovací metody:

Intergrační testy MS – kontrola integrací konkrétní mikroslužby by byla vhodná při větším počtu mikroslužeb, kdy spuštění celého systému by bylo příliš nákladné. V takovém případě by bylo třeba vytvořit mock-funkcionalitu všech prvků, se kterou testovaná MS pracuje, a spustit je v samostatném prostředí. Toto testování je v aktuálním řešení nahrazeno funkčními testy celého IS, které plní stejný účel.

Zátěžové testy – kontrola zátěže by byla přínosná pro odladění systému, avšak vyžaduje víc lokálních, případně vzdálených, prostředků pro testování. Nebylo zavedeno vzhledem k předpokládaným nevypovídajícím výsledkům s ohledem na používané prostředí.

Testování klientského rozhraní – vizuální rozhraní bylo dle specifikace psáno s omezeními (pouze jeden prohlížeč a pouze vysoké rozlišení obrazovky), plnohodnotné testování webového prostředí by zde nebylo v aktuální situaci vhodné. Takové testování bude potřeba v případě dokončení responzivního webového rozhraní, například s pomocí `cypress` balíčku.

7.1 Automatizace

Na základě vybraných testů mohly být plně zautomatizovány pouze statická analýza kódu a jednotkové testy. Kontrola kvality je integrována s pomocí `git hooks` a spouští se během každé tvorby `commit` prvku. Jednotkové testy jsou řešeny obdobně, ale spouštění je prováděno až při každé `git push` akci na server.

Funkční testování je v dáno v podobě samostatného repozitáře a vzhledem k vazbě na celý systém a potřebě fungování všech částí, je lepší to z úplné automatizace vynechat. Implementované testy pokryvaji dokončenou funkcionalitu systému, část výsledku automatizovaného testování je vidět na obrázku 7.1.

```
PASS  test/ms-users/permissions/internal/patchUserPassword.test.js
PASS  test/ms-monitoring/statuses.test.js
PASS  test/ms-auth/token-verification.test.js
PASS  test/ms-auth/sign-in.test.js

Test Suites: 51 passed, 51 total
Tests:       235 passed, 235 total
Snapshots:   0 total
Time:        16.033 s, estimated 17 s
Ran all test suites.
✨  Done in 16.82s.
```

Obrázek 7.1: Výsledek funkčních testů API

Kapitola 8

Kontejnerizace a nasazení

V dané kapitole:

- klíčové aspekty kontejnerizace systému,
- zjednodušené nasazení instance s využitím `docker-compose`.

Vzhledem ke komplexnější struktuře aplikace, je vhodné použít kontejnerizaci jednotlivých mikroslužeb a s tím vyřešit i část problémů týkajících se architektury. Pro kontejnizaci je na základě osobních zkušeností zvolen nástroj Docker, přinese několik výhod:

Čisté prostředí – samostatné, nezávislé prostředí pro sestavování a spouštění služeb IS. Procesy nebudou vázány na (nejspíš) modifikované prostředí vývojáře.

Adresaci v samostatné síti – v Docker existuje funkcionalita pro nastavení síťového prostředí, která dovoluje jednotlivým Docker kontejnerům vytvářet neveřejné síťové spojení, vytvářet vlastní aliasy a vystavovat pouze nutné porty [51].

Integrace s Kubernetes – Docker je kompatibilní s Kubernetes a v případě nutnosti nastavení vyvažování zátěže, případně pokročilejších možností nasazování, by neměl vzniknout problém s integrací [52].

8.1 Kontejnerizace

Kontejnerizace je, jak již bylo zmíněno v analýze MSA, jistý způsob virtualizace. V případě technologie Docker se jedná o sestavení obrazu s aplikací (v případě daného IS jde o jednotlivé mikroslužby a klientskou aplikaci) s následnou tvorbou kontejnerů, neboli instancí obrazů.

V rámci kontejnerizace implementovaných částí kontejnery lze rozdělit na tři základní typy:

Mikroslužby bez databázových migrací – příkladem je autorizační služba, od startu aplikace se vyžaduje pouze sestavení produkčního balíčku a nastartování webového serveru.

Mikroslužby s databázovými migracemi – jedná se třeba o uživatelskou nebo projektovou mikroslužbu, kdy před každým naběhnutím serveru je nutné kontrolovat seznam migrací dodávaných s kontejnerem.

Klientská služba – obal pro Next.js aplikaci.

První dva typy využívají podobnou strukturu `Dockerfile` souboru, který slouží pro tvorbu obrazu. Sestavování je rozděleno na dvě fáze – příprava a sestavení produkční aplikace, pro kterou se využívá první prostředí, a kopírování sestavených zdrojů z první fáze do nového prostředí a nastavení počátečních příkazů pro spuštění. Rozdíl těchto typů je tvořen pouze rozšířením o kontrolu a provedení chybějících migrací před každým spuštěním kontejneru, čímž je zajištěno, že nenastane situace, kdy by `Node.js` aplikace vytvářela výjimky spojené s nekompatibilitou ORM schémat a připojené databáze.

Klientská aplikace takové fázování nutně nepotřebuje a vystačí si s jednodušší konfigurací. Samozřejmě, taková nastavení sestavení obrazů mohou být i dále zdokonalována z bezpečnostních a výkonnostních hledisek, pro běžné užití by však měly být dostatečně robustní.

8.2 Nasazení

I když jsou všechny mikroslužby a klientská čast aplikace plně kontejnerizovány, tak nasadit celý systém alespoň na lokální stroj je poněkud nákladné, protože kromě přístupových údajů bude nutné manuálně konfigurovat směrování aplikací dle IP (internet protocol) adres, aby fungovaly potřebné komunikační kanály.

Aktuálně tato konfigurace je uvedena v `.env` souborech všech služeb a vyžaduje opakované sestavování každé z nich. Pro ukázkové účely však tato rutina může být zjednodušena pomocí `docker-compose`. V souboru `docker-compose.prod.yml`, jenž je určen pro plynulejší nasazení celého systému, je řešeno:

- Automatické sestavování všech kontejnerů při startu – není třeba zvlášť vytvářet Docker obrazy.
- Vytvoření interních sítí pro komunikaci (zvlášť serverové mikroslužby a zvlášť klientské aplikace) – v rámci sítě jsou jednotlivým službám přiřazeny aliasy, tudíž v `.env` souborech lze standardizovat přístupové adresy.
- Zabezpečení přístupu k databázím z vnějších zdrojů – interní systémy jsou vystaveny pouze přes omezené cesty komunikace (v tomto případě pouze jednotný serverový port Gateway služby a klientská aplikace).

Nedokonale je rovněž řešena posloupnost naběhnutí služeb. Dle povahy systému je nutné načtení nejdřív databází, následně mikroslužeb a v poslední řadě Gateway služby. Výchozí Docker nástroj `depends_on`, jenž čeká na nastartování služby, nevystačí vzhledem k pomalejšímu startu samotných systémů v rámci kontejneru. Jedná se o již známý problém a v oficiální dokumentaci je rovněž uváděn způsob řešení s pomocí závislostí knihoven třetích stran [53]. Taková implementace je ponechána pro další rozvoj, protože její existence není nutná. Manuální řízení naběhnutí služeb je popsáno v rámci dokumentace na přiloženém médiu.

Ačkoliv tento problém s načítáním existuje a vyžaduje manuální zásah, tak zbytek cyklu nasazení je zjednodušen na 3 kroky:

- spuštění skriptu pro tvorbu produkčních `.env` souborů,
- manuální doplnění přístupových a bezpečnostních údajů (Github token, klíče apod.),
- spuštění `docker-compose` příkazu.

Po jejich dokončení je systém zcela zprovozněn.

Kapitola 9

Rozvoj systému

V dané kapitole:

- seznam bodů nutných pro dokončení systému,
- seznam možných zdokonalení aktuální implementace.

Implementace daného IS v rámci této práce byla soustředěna kolem návrhu základních principů fungování architektury mikroslužeb a uplatnění zdokonaleného principu tvorby rozdělené klientské aplikace. Vzhledem k obrovskému rozsahu práce a nepředpokládané časové náročnosti vývoje v architektuře mikroslužeb, mnohé funkcionality, jež většinou představují monotónní implementaci REST rozhraní se zápisem do databáze dle určitých podmínek, nebyly realizovány. Vše potřebné pro jejich vývoj však již bylo začleněno.

Tato kapitola definuje v sobě zbývající části systému, jež mají být dodány vzhledem k původní analýze, a poskytuje seznam vylepšení aktuálního řešení. K některým bodům je dodáno, jakým způsobem může být dané vylepšení realizováno.

9.1 Finalizace funkcionality

Dořešení systému práv a zobrazování dat – na základě specifikace v systému uživatelské role a nastavení projektu mají mít vliv na poskytované možnosti a dodávaná data. Rozdělení na role a možnost kontroly rolí byla realizována, avšak pouze na vyšší úrovni (volání rozhraní, zobrazování, nebo skrytí projektu). Je nutné dodělat detailní kontrolu dat, které jsou poskytovány uživatelům na základě rolí, a vytvořit výjimku pro administrátory.

Uživatelsky vhodné zobrazování chyb – v rámci serverové části bylo vytvořeno relativně přehledné kaskádování chyb s propagací zpráv s přesně daným rozhraním, zdaleka ne všechny jsou však zpracovávány klientským rozhraním do uživatelsky příjemné podoby.

Doplnění vhodných uživatelských upozornění – funkcionalita pro oznamování důležitých událostí je plně funkční, je nutné přidat tvorbu takových zpráv ve všech potřebných funkcích.

Podpora jazykových verzí – knihovna Next.js interně podporuje překlad rozhraní, nebylo však vůbec realizováno. Předpokládaný způsob integrace může být s využitím knihovny React-Intl.

A další požadavky, označované od začátku jako rozvojové (viz kapitola „Specifikace nového systému“):

Další možnosti vyhledávání projektů – zadávání a vyhledávání projektů na základě unikátních značek (tagů).

Iterace a úkoly – dělení projektů na iterace a úkoly, jež by se daly plnit tvorbou obsahových částí projektu.

Snímky iterací – odevzdávání stavu obsahu projektu, vzhledem k implementované funkcionalitě může být implementováno jako značka v gitu. Tzn. odevzdávat se bude historický bod projektu bez kopírování stavu projektu, jak to bylo realizováno v bakalářské práci.

Hodnocení iterací – ohodnocení odevzdaného snímku a tudíž i iterace osobou, jež je označována jako autorita v rámci IS.

Editace týmu – editace projektových týmů ověřenými uživateli bez souhlasu uživatelů, včetně pozvánek apod.

9.2 Možné zdokonalení systému

Zlepšení logovacího systému – aktuální řešení logování předpokládá v sobě záznam tvořený datem, původem (název mikroslužby) a zprávou, která se předává přes brokera do NoSQL databáze. Poskytuje to možnost filtrování výsledků a vyhledávání, nemusí to být však odolné vůči výpadkům brokera zpráv. Potenciálně lepším řešením by mohlo být zavedení klasických, textových logů, které by mohly sloužit jako odlaďovací nástroj v případě zkoumání mikroslužby bez prostředí. Zároveň je možné zlepšit strukturu samotného logu.

Aktivní monitoring fungování služeb – v klientské aplikaci aktuálně existuje stránka v administrátorském panelu, která `pull` způsobem zjišťuje aktuální stav mikroslužeb. Tato funkcionalita by se mohla vyčlenit do samostatně fungujícího sledování s upozorněním, například přes e-mail, že část systému měla výpadek. Případně prozkoumat a využít již existující řešení.

Monitoring a vyvažování zátěže – se sledováním stavu mikroslužeb je rovněž spojeno sledování vytíženosti služeb a brokeru jako takového. Dle zběžného průzkumu vyplynulo, že by se dala používat kombinace `Prometheus` pro sledování a `Grafana` pro vizualizaci, nicméně vyžadovalo by to komplexnější integraci.

Kontrola kompatibility mikroslužeb – jednotlivé služby mohou být aktuálně verzovány dle jejich rozhraní. Není však dořešena vzájemná kompatibilita na základě závislostí. Potenciálním řešením by mohla být definice závislostí v rámci sémantického verzování a kontrola kompatibility v monitorovací mikroslužbě.

Zabránění volání interních přístupových bodů ze vně – Gateway fasáda v současné době poskytuje ven všechna REST rozhraní (externí i interní) s tím, že interní v rámci provolání vyžadují speciální autorizační hlavičku s klíčem. V ideálním případě by se však interní rozhraní nemělo vůbec vystavovat. Řešením by mohla být explicitní filtrace požadavků na straně `nginx` nebo oddělení interního rozhraní s pomocí URI a překonfigurování všech mikroslužeb na nový typ zdroje dat.

Zakládání administrátorského účtu – administrátorský účet je nyní součástí základní sady dat pro databázi. Může se nahradit samostatnou, jednorázovou počáteční registrací před prvním startem IS.

Nahrávání souborů – v rámci tvorby obsahu může vzniknout potřeba nahrávat soubory, může to být řešeno s pomocí interpretátoru obsahu.

Zdokonalení testování – aktuální testování počítá s otestováním samostatných funkcí s pomocí jednotkových testů a celistvost fungování serverové části s pomocí integračních. V případě dokončení uživatelského rozhraní by mohlo vzniknout testování na základě simulace uživatelské činnosti a další testování, například zátěže, bezpečnosti apod.

Zdokonalení vývoje interpretů obsahu – interprety pro obsahy nyní jsou tvořeny manuální činností, která může být zautomatizována. Vývoj interpretů by následně mohl probíhat jako vývoj běžné React apliakce, kde by během sestavování programu redukoval obsah na potřebné minimum.

Rozšíření konceptu mikroslužeb u klienta – stejně, jako jsou realizovány interprety obsahu, by mohl fungovat zbytek klientské aplikace. Zde je však otázka, zda by se to v případě relativně malého projektu vyplatilo.

Paralelní úpravy obsahu – úpravy stejné části obsahu jsou nyní zabezpečeny s pomocí kontroly `hash` klíče původního obsahu. Mohlo by být vhodné realizovat paralelní úpravu 1 části více uživateli.

Separace kaskádových stylů a obsahu – v současné implementaci kvůli vlastnostem CSS není dokonale řešeno ovlivňování jednotlivých částí obsahu existujícími globálními styly. Měl by se ideálně definovat jednotný styl pro základní prvky (též nazývaný „style-guide“), který by ovlivňoval i obsahy, ale zbytek by mohl být napsaný s pomocí technologie CSS modulů. Tím by se vyřešily potenciální konflikty mezi CSS pravidly.

UX/UI – rozhraní může být přizpůsobeno mobilnímu rozhraní a vizuálně odladěno.

Obecně se dá říct (z osobních zkušeností vzhledem k předchozí implementaci IS), že architektura mikroslužeb má řadově víc možností, kde se dá zdokonalit, v porovnání s monolitickou architekturou. Taková volnost v řešení různých aspektů na jednu stranu přináší větší intuitivní chápání, na druhou zbytečně komplikuje vývoj a nevhodně oddaluje dosažení samotného cíle alespoň v první, konzistentní podobě. Práce s plnohodnotným dokončením tohoto IS se pro jednoho člověka může protáhnout na nevhodně dlouhou dobu.

Kapitola 10

Srovnání výsledků

V dané kapitole:

- zhodnocení výhody přechodu z MA na MSA,
- seznam zásadních změn ve srovnání se starou implementací IS.

Informační systém pro správu projektů byl úspěšně převeden na architekturu mikroslužeb. Tento přechod jednoznačně přispěl k systematizaci celého systému a začal se vnímat jako mnohem uspořádanější, než v monolitické architektuře. Jsou vidět striktně vymezené hranice a chování, systém je mnohem ohebnější z hlediska implementace a rozvojových možností. Mikroslužby však výrazně snížily rychlost vývoje a prodloužily fázi potřebné analýzy. Bylo nutné řešit víc problémů, se kterými se u monolitické architektury člověk nesetká – zejména infrastrukturní komunikaci a rozdělení API mezi různými službami. Nepomáhala tomu ani rozmanitost možností návrhu architektury jako takové. To vše mělo za následek zmenšení rozsahu dodávané funkcionality, oproti původnímu, monolitickému řešení, i když času, věnovaného implementaci, bylo řádově více.

Realizace IS v architektuře mikroslužeb byla, dle subjektivního hlediska, obecně přínosná, avšak ne dokonalá. Je možné, že jiný typ dekompozice a volby komunikace v tomto případě by mohl zredukovat potřebné prostředky pro realizaci a ušetřit čas. Prozkoumat všechny kombinace v rámci této diplomové práce bohužel nebylo uskutečnitelné.

V následujícím přehledu se uvádí seznam zásadních změn, které nastaly ve srovnání se starou implementací:

- Původní MA architektura potřebovala mnohem méně časů a prostředků pro vývoj. To se nejspíš bude projevovat i v pozdějším rozvoji.
- Vnímání fungování nové serverové architektury je víc intuitivní, než u předchozí. Přispěla tomu zejména dekompozice.
- Mikroslužby z tohoto systému lze ještě víc zobecnit a používat v jiných projektech. V případě monolitu by taková možnost neexistovala.
- Zhoršila se organizace vývoje pro menší počet vývojářů. Správa několika mikroslužeb jedním člověkem je méně přehledná, než správa monolitu.
- Kvůli zvolené struktuře začalo docházet ke kontrolované duplicitě kódu mezi repozitáři.
- Nový způsob implementace interpretů obsahů v klientské části přinesl jednodušší vývoj a pohodlnější prostředí, i když omezil technologii pouze na React.
- Původní systém obsahoval víc funkcionality s menším důrazem na přehlednost architektury, novější dává větší prioritu architektuře, ale aktuálně zaostává ve funkcionalitě.
- Testování v nové realizaci je automatizováno a vyžaduje méně manuálních zásahů.
- Došlo k úplné kontejnerizaci systému.

Závěr

Tato diplomová práce byla věnována analýze architektury mikroslužeb a návrhu realizace přechodu existujícího prototypu informačního systému pro správu projektů z monolitické architektury na architekturu mikroslužeb. V souladu se zadáním bylo stanovenu několik cílů, z nichž všechny byly úspěšně dokončeny.

V rámci práce na základě existujících podkladů byla nejdřív provedena analýza poskytovaného systému. Výstupem této činnosti se stal seznam negativních a pozitivních skutečností, který spolu s aktualizovanou rešerší fakultních systémů se stal předlohou pro novou specifikaci projektu. Následně proběhla analýza architektury mikroslužeb, jež byla vzhledem k obrovskému rozsahu a nejednoznačným přístupům zredukována na potřebné minimum pro vytvoření nového prototypu. Samostatná kapitola byla věnována využití databází a transakčnímu zpracování v případě oddělených částí relačních databází.

S využitím všech získaných znalostí byl navržen a realizován přechod serverové a klientské části aplikace na architekturu mikroslužeb. Nejvíc přínosné nebo implementačně zajímavé aspekty byly popsány v rámci kapitol věnujících se vývoji aplikace. Serverová část byla rozdělena na mikroslužby s choreografií, propojena s Github, který nahradil dřívější MongoDB, a plně kontejnerizována s pomocí Docker. Pro práci s daty se kromě databází začal využívat broker zpráv RabbitMQ. Klientská část byla rovněž kontejnerizována a začala používat novou funkcionalitu pro vykreslování částí projektů s pomocí vnějších React komponent distribuovaných v podobě ES6 modulů.

Manuální testování bylo do určité míry nahrazeno automatizovaným. Statická analýza kódu a jednotkové testy se staly součástí životního cyklu vývoje aplikace a automaticky se spouštějí v potřebných chvílích. Funkčně-integrační testy je třeba pouštět

manuálně, byly definovány v samostatném repozitáři a automaticky testují poskytované rozhraní z hlediska přístupových práv a typických testovacích scénářů.

Pro výsledný stav informačního systému byl definován možný budoucí rozvoj a sepsána vývojářská dokumentace. Celý výsledek byl porovnán s původním řešením – změna byla přínosná z hlediska architektury, přinesla však zvýšené náklady pro vývoj, což pro tak malý projekt nemusí být výhodné.

Zdrojové kódy všech částí IS byly zveřejněny na Github serveru. Spolu s dodatečnými materiály se rovněž nachází na přiloženém médiu. Projekt, jako celek, není v dokončené fázi a o rozvoji jeho současného stavu se bude uvažovat i bez akademických cílů.

Architektura mikroslužeb má svůj účel a využití v rámci vývoje systémů, je intuitivnější, než některé jiné architektury, ale její realizace je komplikovanější, než se na první pohled zdá. Může najít svoje uplatnění v komplexních strukturách a systémech s potřebou parciálního škálování.

Literatura

[1] Mauersberger, L.: Microservices: What They Are and Why Use Them. 2017, [cit. 2021-10-30]. Dostupné z: https://www.leanix.net/en/blog/a-brief-history-of-microservices

[2] Google Trends: Explore search interest for microservices architecture by time, location and popularity on Google Trends. 2022, [cit. 2021-10-30]. Dostupné z: https://trends.google.com/trends/explore?date=all&q=microservice%20sarchitecture,microservice,microservices

[3] Dunaevskiy, S.: Informační systém pro správu studijních projektů [online]. 2019, [cit. 2021-10-24]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/86181/F8-BP-2019-Dunaevskiy-Sergey-thesis.pdf?sequence=-1&isAllowed=y

[4] Suchánek, M.: Hodnocení vedoucího závěrečné práce. 2020, [cit. 2021-10-24]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/86181/F8-BP-2020-posudek-Suchanek_Marek.pdf?sequence=-1&isAllowed=y

[5] Valenta, M.: Posudek oponenta závěrečné práce. 2020, [cit. 2021-10-24]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/86181/F8-BP-2020-posudek-Valenta_Michal.pdf?sequence=-1&isAllowed=y

[6] Vercel, I.: Next.js by Vercel - The React Framework. 2022, [cit. 2022-02-19]. Dostupné z: https://nextjs.org

[7] Tým SwinPro: Swinpro - Swinpro - Vítejte. Dostupné z: https://project.fit.cvut.cz/swinpro/

[8] FIT CTU Course Pages: E-mail. 2022, [cit. 2021-10-31]. Dostupné z: https://help.fit.cvut.cz/email/index.html#_footnote_1

[9] Podskalský, A.: Softwarová architektura. Co to vlastně je? 2022, [cit. 2022-01-22]. Dostupné z: https://profinit.eu/blog/softwarova-architektura-co-to-vlastne-je/

[10] Fowler, M.: DesignStaminaHypothesis. 2007, [cit. 2022-02-13]. Dostupné z: https://martinfowler.com/bliki/DesignStaminaHypothesis.html

[11] Richardson, C.: *Microservices Patterns*. Manning Publications Co., 2019, ISBN 9781617294549.

[12] Sengupta, S.: Service-Oriented Architecture vs Microservices Architecture: Comparing SOA to MSA. 2021, [cit. 2022-02-05]. Dostupné z: https://www.bmc.com/blogs/microservices-vs-soa-whats-difference/

[13] Janssen, T.: SOLID Design Principles Explained: The Single Responsibility Principle. 2020, [cit. 2022-02-15]. Dostupné z: https://stackify.com/solid-design-principles/

[14] Montiel, I.: Low Coupling, High Cohesion. 2018, [cit. 2022-02-15]. Dostupné z: https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6

[15] Richardson, C.: Pattern: Decompose by business capability. 2021, [cit. 2022-02-15]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

[16] Richardson, C.: Pattern: Decompose by subdomain. 2021, [cit. 2022-02-15]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html

[17] Foundation, T. G.: GraphQL | A query language for your API. 2022, [cit. 2022-02-25]. Dostupné z: https://graphql.org

[18] gRPC Authors: Introduction to gRPC. 2022, [cit. 2022-02-25]. Dostupné z: https://grpc.io/docs/what-is-grpc/introduction/

[19] VMware, Inc.: Messaging that just works — RabbitMQ. 2022, [cit. 2022-02-25]. Dostupné z: https://www.rabbitmq.com

[20] Apache Software Foundation: Apache Kafka. 2022, [cit. 2022-02-25]. Dostupné z: https://kafka.apache.org

[21] MQTT.org: MQTT: The Standard for IoT Messaging. 2022, [cit. 2022-02-25]. Dostupné z: https://mqtt.org

[22] Schabowsky, J.: Microservices Choreography vs Orchestration: The Benefits of Choreography. 2019, [cit. 2022-02-13]. Dostupné z: https://solace.com/blog/microservices-choreography-vs-orchestration/

[23] Richardson, C.: Pattern: API Gateway / Backends for Frontends. 2022, [cit. 2022-04-07]. Dostupné z: https://microservices.io/patterns/apigateway.html

[24] Kanjilal, J.: Get to know 4 microservices versioning techniques. 2021, [cit. 2022-04-04]. Dostupné z: https://www.techtarget.com/searchapparchitecture/tip/Get-to-know-4-microservices-versioning-techniques

[25] Preston-Werner, T.: Semantic Versioning 2.0.0. 2022, [cit. 2022-04-04]. Dostupné z: https://semver.org

[26] Hlava, T.: Statické a dynamické testy. 2011, [cit. 2022-04-05]. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/staticke-a-dynamicke-testy/

[27] Zemek, P.: Proč rozlišovat jednotkové a integrační testy. 2015, [cit. 2022-04-05]. Dostupné z: https://cs-blog.petrzemek.net/2015-04-18-proc-rozlisovat-jednotkove-a-integracni-testy

[28] Dedík, V.: Testování software. 2015, [cit. 2022-01-30]. Dostupné z: https://web.archive.org/web/20150924012259/http://www.fi.muni.cz/~xdedik1/PB029/files/zapoctovy_dokument_1.pdf

[29] Kitner, R.: Typy testování software (třídění testů). 2021, [cit. 2022-02-03]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/

[30] Lukeš, P.: Zátěžové testování webových aplikací - úvod. 2014, [cit. 2022-04-05]. Dostupné z: https://www.etnetera.cz/blog/zatezove-testovani-webovych-aplikaci-uvod

[31] Belagatti, P.: Microservices: Mono repo vs. multiple repositories. 2016, [cit. 2022-04-04]. Dostupné z: https://medium.com/@pavanbelagatti/microservices-mono-repo-vs-multiple-repositories-6e139b5ca44a

[32] Polednik, M.: Virtualizace a kontejnerizace. 2022, [cit. 2022-04-02]. Dostupné z: https://www.fi.muni.cz/~kas/pv090/referaty/2014-podzim/virt.html

[33] Docker Inc.: Use containers to Build, Share and Run your applications. 2022, [cit. 2022-04-07]. Dostupné z: https://www.docker.com/resources/what-container/

[34] Morgan Bruce, Paulo A. Pereira: *Microservices in Action*. Manning Publications Co., 2019, ISBN 9781617294457.

[35] Skedler Team: Everything You Need to Know about Grafana. 2021, [cit. 2022-04-11]. Dostupné z: https://www.skedler.com/blog/everything-you-need-to-know-about-grafana/

[36] Grafana Labs: Your observability stack. 2022, [cit. 2022-04-11]. Dostupné z: https://grafana.com

[37] Richardson, C.: Pattern: Health Check API. 2022, [cit. 2022-04-11]. Dostupné z: https://microservices.io/patterns/observability/health-check-api.html

[38] Refsnes Data: Browser Statistics. 2022, [cit. 2022-02-04]. Dostupné z: https://www.w3schools.com/browsers/

[39] Reis, C.: Multi-process Architecture. 2008, [cit. 2022-02-04]. Dostupné z: https://blog.chromium.org/2008/09/multi-process-architecture.html

[40] Rifki, N.: The ultimate guide to iframes. 2020, [cit. 2022-02-04]. Dostupné z: https://blog.logrocket.com/the-ultimate-guide-to-iframes/

[41] Geers, M.: Micro Frontends. 2022, [cit. 2022-02-04]. Dostupné z: https://micro-frontends.org

[42] Microservices.io: Pattern: Shared database. 2022, [cit. 2022-02-07]. Dostupné z: https://microservices.io/patterns/data/shared-database.html

[43] Microservices.io: Pattern: Database per service. 2022, [cit. 2022-02-07]. Dostupné z: https://microservices.io/patterns/data/database-per-service.html

[44] Richardson, C.: Pattern: API Composition. 2022, [cit. 2022-04-11]. Dostupné z: https://microservices.io/patterns/data/api-composition.html

[45] Richardson, C.: Pattern: Command Query Responsibility Segregation (CQRS). 2022, [cit. 2022-04-11]. Dostupné z: https://microservices.io/patterns/data/cqrs.html

[46] Techopedia: Transaction. 2020, [cit. 2022-04-11]. Dostupné z: https://www.techopedia.com/definition/16455/transaction-databases

[47] Hazelcast, Inc.: Distributed Transaction. 2022, [cit. 2022-04-11]. Dostupné z: https://hazelcast.com/glossary/distributed-transaction/

[48] steve: How to process distributed transaction within postgresql? 2022, [cit. 2022-04-11]. Dostupné z: https://stackoverflow.com/questions/21109362/how-to-process-distributed-transaction-within-postgresql

[49] Richardson, C.: Pattern: Saga. 2022, [cit. 2022-04-11]. Dostupné z: https://microservices.io/patterns/data/saga.html

[50] GitHub, I.: Create or update file contents. 2022, [cit. 2022-04-07]. Dostupné z: https://docs.github.com/en/rest/reference/repos#create-or-update-file-contents

[51] Docker Inc.: Networking overview. 2021, [cit. 2022-04-02]. Dostupné z: https://docs.docker.com/network/

[52] Docker Inc.: Deploy on Kubernetes. 2021, [cit. 2022-04-02]. Dostupné z: https://docs.docker.com/desktop/kubernetes/

[53] Docker Inc.: Control startup and shutdown order in Compose. 2021, [cit. 2022-04-02]. Dostupné z: https://docs.docker.com/compose/startup-order/

Příloha **A**

Seznam použitých zkratek

ACID atomicity, consistency, isolation, durability.
AJAX asynchronous javascript and xml.
API application programming interface.
CQRS command query responsibility segregation.
CRUD create, read, update, delete.
CSS cascading style sheet.
DOM document object model.
ES6 ECMAScript 6.
ESB enterprise service bus.
FaaS function as a service.
FIT Fakulta informačních technologií.
HTML hypertext markup language.
HTTP hypertext transfer protocol.
ID identity.
IDE integrated development environment.
IIFE immediately invoked function expression.
IoT internet of things.
IP internet protocol.
IS informační systém.
JS JavaScript.
JSON javascript object notation.
MA monolithic architecture.
MS microservice.
MSA microservice architecture.
ORM object relational mapping.

REST representational state transfer.
RPC remote procedure call.
SA serverless architecture.
SOA service oriented architecture.
SQL structured query language.
SSG static generation.
SSR server side rendering.
UI user interface.
URI uniform resource identifier.
UUID universally unique identifier.
UX user experience.
VCS version control system.
ČVUT České vysoké učení technické.

Příloha **B**

Obsah přiloženého média

Vizualizace struktury souborů na přiloženém médiu.

```
README.txt ..... soubor s popisem
src ..... adresář se zdrojovými soubory
    application ..... adresář se zdrojovými soubory aplikace
        its ..... repozitář s informačním systémem
        its__documentation ..... dokumentace informačního systému
        its__ms_template ..... šablona pro mikroslužby
        its__testing ..... sada integračních testů
    thesis ..... adresář se zdrojovými soubory textu
text ..... adresář s textem diplomové práce
    thesis.pdf ..... diplomová práce v PDF formátu
resources ..... adresář s přílohami
    postman ..... adresář s Postman kolekcí API
    its__interpreters ..... adresář s ukázkou interpretů
    testing ..... výsledky integračního testování
```