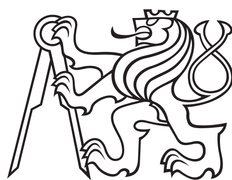
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Transformace z monolitické aplikace na mikroservisní aplikace

Iurii Lebedev

Vedoucí: **Ing. Jiří Šebek**
Obor: **Enterprise systémy**
Studijní program: **Softwarové inženýrství a technologie**
Květen 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Lebedev** Jméno: **Iurii** Osobní číslo: **511260**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

Specializace: **Enterprise systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Transformace z monolitické aplikace na mikroservisní aplikace

Název bakalářské práce anglicky:

Transformation from monolitic app to microservice app

Pokyny pro vypracování:

V dnešní době se společnosti snaží, aby jejich aplikace byly škálovatelnější, flexibilnější a spolehlivější, protože technologie se rychle vyvíjejí. K dosažení tohoto cíle lze monolitické aplikace přeměnit na mikroslužby. Tento proces vývoje přináší značné výhody, ale také řadu výzev, jako je změna kultury vývoje, správa distribuovaných systémů a zajištění bezpečnosti. V těchto souvislostech je automatizace zásadní pro zjednodušení a urychlení přechodu na architekturu mikroslužeb.
Cíle této práce jsou:
Prozkoumat výhody monolitické a mikroservisní architektury.
Vytvořit tabulku rozdílů v architekturách (analýza).
Vytvořit možný způsob transformace monolitu na mikroslužby (návrh).
Následně implementovat aplikaci pro danou transformaci.
Posledním bodem práce bude otestovat scénáře průchodů jak na monolitické demo aplikaci tak i na výsledné microservisní aplikaci a zjistit, jestli se aplikace správně chová.

Seznam doporučené literatury:

Workload Characterization for Microservices
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7581269
Migrating Application from Monolith to Microservices
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9211252
An automatic extraction approach: transition to microservices architecture from monolithic application
https://dl.acm.org/doi/abs/10.1145/3234152.3234195

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **16.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Poděkování

Chtěl bych nejprve poděkovat svému vedoucímu Ing. Jiřímu Šebkovi za pečlivou pomoc a důležité rady, které mi velice pomohly u prací. Dále bych chtěl poděkovat všem svým blízkým za podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. května 2024

Abstrakt

ato bakalářská práce byla zvolena za účelem studia mikroslužeb a prohloubení znalostí v této oblasti. Během realizace projektu byla vytvořena aplikace, která může částečně nahradit člověka při transformaci architektury stávající aplikace. Byla provedena studie existujících aplikací a možných řešení problému automatizace transformace.

Klíčová slova: Microservicies, transformation, Java, SpringBoot, automatization

Vedoucí: Ing. Jiří Šebek
kabinet výuky informatiky FEL

Abstract

This bachelor thesis was chosen to study microservices and to deepen the knowledge in this area. During the implementation of the project, an application was created that can partially replace humans in transforming the architecture of an existing application. A study of existing applications and possible solutions to the problem of automating the transformation was carried out.

Keywords: Microservicies, transformation, Java, SpringBoot, automatization

Title translation: Transformation from monolitic app to microservice app

Obsah

1 Úvod **1**
1.1 Rizika a problémy při vývoji softwaru . . . 1
1.1.1 Problémy při vývoji softwaru . 1
1.1.2 Důsledky těchto problémů 2
1.2 Závěr . . . 3

2 Rešerše **5**
2.1 Vliv výběru architektury na vývoj projektu . . . 5
2.1.1 Dopady volby architektektonického stylu na vývoj projektu . . . 5
2.2 Obtíže při změně architektury v existujícím projektu . . . 7
2.2.1 Google Cloud . . . 7
2.2.2 Atlassian Compass . . . 8
2.2.3 vFunction . . . 9
2.2.4 IBM Mono2Micro . . . 11

3 Analýza **13**
3.1 Funkční požadavky . . . 13
3.1.1 Seznam funkčních požadavku 13
3.1.2 Technologické řešení funkčních požadavků . . . 14
3.2 Nefunkční požadavky . . . 15
3.3 UseCase Diagram . . . 15

4 Návrh **17**
4.1 Struktura aplikace před transformací . . . 17
4.2 Komponenty systému . . . 18
4.3 Struktura aplikace po transformací . . . 19

5 Implementace **21**
5.1 Průběh Implementace . . . 21
5.1.1 Parsovani input projektu 21
5.1.2 Konfigurace transformace . . . 22
5.1.3 Generování tříd a projektu . . 24
5.1.4 Použití Backend For Frontend Patternu . . . 26
5.1.5 Import Processing . . . 26
5.2 Proces Dekompozice . . . 27
5.3 Úspěšnost Projektu . . . 28

6 Konfigurace **29**
6.1 Demonstrační konfigurace dekompozice . . . 29

7 Regresní testování **31**

8 Závěr **33**

A Literatura **35**

Obrázky

3.1 Use Case Diagram 16

4.1 *Obecný princip fungování applikace* 17
4.2 *ClassDiagram demonstrační aplikace do transformace* 18
4.3 Class Diagram aplikace Modulizr 19
4.4 Mikrosevisní class Diagram Demonstrační aplikace 20

5.1 Seuence Diagram 27

7.1 Databáze uživatelů výsledné aplikace 32

Tabulky

Kapitola 1
Úvod

Vývoj softwaru je náchylný k různým problémům, které mohou ohrozit úspěch projektu[1]. Mezi hlavní problémy patří nedostatečné plánování, špatná komunikace v týmu, nekonzistentní požadavky, technický dluh a problémy s integrací. Tyto problémy mohou vést k překročení rozpočtu, zpoždění v harmonogramu, nízké kvalitě produktu a nespokojenosti zákazníků. Volba nevhodné architektury může tyto problémy ještě zhoršit, což zdůrazňuje důležitost pečlivého výběru architektonického stylu.[2, 3, 4]

1.1 Rizika a problémy při vývoji softwaru

Při vývoji softwaru se nevyhnutelně setkáváme s různými riziky a problémy[4]. Tyto problémy mohou mít značné dopady na konečný produkt, a proto je důležité je identifikovat a řídit. V této kapitole se zaměříme na nejčastější problémy, které mohou vzniknout při vývoji softwaru, jaké jsou jejich důsledky a jaké jsou jejich hlavní příčiny.[5]

1.1.1 Problémy při vývoji softwaru

Komplexita projektu

- Příčiny: Rostoucí požadavky a složitost systému mohou způsobit, že projekt se stane obtížně zvládnutelným.
- Důsledky: Zvýšená komplexita vede k chybám, obtížnému ladění a delšímu času na implementaci nových funkcionalit.[6]
- Řešení: Používání osvědčených metod a principů jako je rozdělení projektu na menší části, modulární design a pravidelné revize kódu.

Nedostatečná komunikace v týmu

- Příčiny: Špatně definované komunikační kanály nebo nejasné zadání úkolů.
- Důsledky: Nesprávně interpretované požadavky, zpoždění v harmonogramu a zvýšené riziko vzniku chyb.[6]

- Řešení: Zavedení pravidelných týmových schůzek, používání nástrojů pro řízení projektů a zajištění jasné a průběžné komunikace.

Nedostatek zdrojů

- Příčiny: Omezené finanční prostředky, nedostatečný počet vývojářů nebo nedostatečné technické vybavení.
- Důsledky: Nedokončení projektu v plánovaném rozsahu nebo kvalitě, časté změny v plánu a přetížení týmu.[7]
- Řešení: Realistické plánování rozpočtu, efektivní využití dostupných zdrojů a průběžné monitorování stavu projektu.

Technický dluh

- Příčiny: Kompromisy v kvalitě kódu kvůli tlaku na rychlé dodání nebo nedostatečné testování.
- Důsledky: Zvýšené náklady na údržbu, obtížné rozšiřování a integrace nových funkcí.[8]
- Řešení: Zavedení přísných standardů kódování, pravidelné refaktoringy a důkladné testování.

Nejasné nebo často se měnící požadavky

- Příčiny: Nedostatečně definované požadavky na začátku projektu nebo změny na základě zpětné vazby od zákazníků.
- Důsledky: Zvýšení nákladů a času na vývoj, časté přepisování kódu a nestabilní produkt.[9, 7]
- Řešení: Použití agilních metodik, které umožňují flexibilní reakce na změny, a důkladná dokumentace požadavků.

1.1.2 Důsledky těchto problémů

Neschopnost efektivně řešit výše uvedené problémy může mít vážné důsledky pro projekt. Mezi hlavní rizika patří:

- **Zpoždění projektu**: nedodržení stanoveného harmonogramu může vést ke ztrátě důvěry zákazníků a zvýšení nákladů.
- **Nekvalitní produkt**: produkty s chybami nebo nedostatečnou funkcionalitou mohou poškodit reputaci firmy a vést k dodatečným nákladům na opravy.
- **Překročení rozpočtu**: dodatečné náklady spojené s řešením problémů mohou překročit původní plánovaný rozpočet, což může ohrozit finanční stabilitu projektu.
- **Nespokojenost zákazníků**: nedodání produktu dle očekávání zákazníků může vést ke ztrátě klientů a negativním recenzím.

1.2 Závěr

Cílem této bakalářské práce je prozkoumat možnosti automatizace procesu transformace mezi monolitickou a mikroservisní architekturou. Tento proces je klíčový pro organizace, které se snaží zvýšit svou schopnost reagovat na měnící se tržní požadavky a zlepšit škálovatelnost a udržitelnost svých systémů. Přechod z monolitického designu na mikroservisní může přinést řadu výhod, včetně lepší modularity, snadnějšího nasazování a vyšší resilience.[10, 11]

Automatizace tohoto procesu by mohla znamenat značné snížení času a nákladů spojených s transformací, a také by mohla zmenšit riziko chyb během přechodu. V této práci byly zkoumány různé nástroje a metodiky, které by mohly pomoci automatizovat některé aspekty migrace, včetně analýzy závislostí, rozkladu monolitů na menší služby a managementu těchto služeb v produkčním prostředí.[12]

Dále byly identifikovány hlavní výzvy spojené s automatizací transformace, jako jsou složitost rozpoznávání a mapování funkčních vazeb v monolitických aplikacích a obtíže spojené s testováním a zajištěním kvality rozdělených služeb. Tyto výzvy vyžadují další výzkum a vývoj v oblasti nástrojů a algoritmů, které by mohly podporovat hlubší a přesnější analýzu softwarových systémů. [13] Výsledky této práce naznačují, že i přes existující nástroje a technologie je plně automatizovaný přechod z monolitické architektury na mikroservisní v mnoha případech stále mimo dosah. Tato skutečnost zdůrazňuje potřebu dalšího výzkumu a vývoje v této oblasti. V závěru práce jsou navržena doporučení pro budoucí projekty a výzkum, které by mohly pomoci překonat stávající překážky a usnadnit cestu k efektivnější a bezpečnější transformaci architektur.

Kapitola 2

Rešerše

Volba architektury je při vývoji softwaru klíčovým aspektem, který určuje nejen počáteční fáze návrhu, ale také dlouhodobou udržitelnost, škálovatelnost a udržovatelnost systému. Zvolená architektura, pokud splňuje funkční požadavky projektu, může výrazně usnadnit proces vývoje a podpořit softwarový produkt, zatímco nevhodná volba může vést k dodatečným nákladům, zkomplikovat projekt a zpomalit proces vývoje. [14, **?**] Prozkoumáme, jaké existují nástroje pro usnadnění procesu transformace mezi architektury, a na příkladech těchto nástrojů zjistíme jakymi způsoby se dá automatizovat přechod od monolitických do mikroservisních architektonických stylů.[15]

2.1 Vliv výběru architektury na vývoj projektu

Výběr architektury má zásadní vliv na vývoj projektu[3]. Architektura určuje strukturu systému, způsob komunikace mezi komponentami a jejich závislosti. Špatně zvolená architektura může vést k problémům s udržitelností, škálovatelností a výkonem systému. Například monolitická architektura může být na začátku jednodušší na implementaci, ale může se stát obtížně škálovatelnou a udržovatelnou, jakmile systém roste. Naopak mikroservisní architektura může zpočátku vyžadovat více úsilí při implementaci, ale nabízí lepší škálovatelnost a flexibilitu v dlouhodobém horizontu. Volba architektury má zásadní vliv na celý vývojový proces, od počáteční fáze plánování až po nasazení a údržbu hotového produktu. Monolitická architektura může být vhodná pro menší projekty s omezeným rozsahem, zatímco mikroservisní architektura může být lepší volbou pro komplexní a škálovatelné systémy. Důkladné zvážení požadavků projektu, technických omezení a dlouhodobých cílů je klíčové pro úspěšný vývoj a provoz softwaru.

2.1.1 Dopady volby architektektonického stylu na vývoj projektu

Volba architektonického stylu je jedním z klíčových rozhodnutí, které ovlivňuje mnoho aspektů vývoje softwaru.[16] Toto rozhodnutí má dalekosáhlé důsledky na efektivitu vývoje, škálovatelnost, údržbu, náklady a celkovou výkonnost a spolehlivost systému.[17] V této sekci prozkoumáme, jak monolitické a

mikroservisní architektury ovlivňují klíčové oblasti projektu a zdůrazníme situace, kdy jedna může být preferována před druhou.[9]

Efektivita vývoje

Monolitická architektura může být zpočátku rychlejší na vývoj, protože všechny komponenty jsou integrovány do jednoho celku. To usnadňuje koordinaci a snižuje počet závislostí mezi komponentami. Avšak s rostoucí složitostí a rozsahem projektu se může údržba a rozšiřování stát náročnějšími, což může vést k pomalejšímu vývoji nových funkcí. Mikroservisní architektura vyžaduje důkladnější plánování a koordinaci, neboť každá služba funguje nezávisle. Toto rozdělení může zkomplikovat počáteční nastavení projektu, ale umožňuje paralelní vývoj více týmů na různých službách. V dlouhodobém horizontu to může významně zvýšit efektivitu vývoje a snížit čas potřebný k uvedení nových funkcí na trh.

Škálovatelnost

V monolitické architektuře je škálovatelnost omezená, protože škálovat je třeba celý systém najednou, což může být neefektivní a nákladné, zejména v případě, že zátěž není rovnoměrně rozložena mezi různé části aplikace. Mikroservisní architektura naopak umožňuje škálovat jednotlivé služby nezávisle na ostatních. To znamená, že můžete přidávat zdroje pouze tam, kde jsou potřeba, což zvyšuje celkovou flexibilitu a efektivitu systému při zvládání zvýšeného zatížení.

Údržba a rozšiřování

Údržba a rozšiřování jsou v monolitické architektuře často komplikovanější, protože změny v jedné části systému mohou nechtěně ovlivnit jiné části, což zvyšuje riziko chyb při nasazování nových verzí. Mikroservisní architektura usnadňuje údržbu a rozšiřování tím, že každá služba je izolovaná a nezávislá. To znamená, že změny v jedné službě obvykle nemají vliv na ostatní služby, což umožňuje rychlejší iterace a méně riskantní nasazování.

Náklady

Monolitické systémy mohou mít nižší počáteční náklady na vývoj a infrastrukturu, ale s rostoucí složitostí mohou náklady na údržbu a rozšiřování rychle narůstat. Mikroservisní systémy mohou vyžadovat vyšší počáteční investici do návrhu a infrastruktury, jako jsou servery a síťové komponenty pro komunikaci mezi službami. Avšak tyto náklady mohou být kompenzovány větší agilitou a efektivitou při škálování a provozu.

Výkon a spolehlivost

Monolitická architektura může trpět problémy s výkonem, pokud není dobře navržena, protože všechny požadavky jsou zpracovány jedním systémem. Mikroservisní architektura nabízí lepší možnosti izolace a optimalizace jednotlivých služeb, což může vést k vylepšení výkonu a spolehlivosti celkového systému.

2.2 Obtíže při změně architektury v existujícím projektu

Změna architektury v již existujícím projektu je často složitý a nákladný proces.[4] Vyžaduje pečlivé plánování, důkladnou analýzu stávajícího kódu a migraci jednotlivých komponent. [17]Přepisování kódu, přesouvání tříd a metod a hledání nových vzorů, které jsou pro projekt výhodnější, může být časově náročné a vyžaduje zkušený tým. Proto je důležité zvážit dlouhodobé důsledky architektonických rozhodnutí již v raných fázích projektu.[13]

2.2.1 Google Cloud

Google Cloud nabízí komplexní sadu nástrojů a zdrojů, které usnadňují přechod z monolitických aplikací na architekturu mikroservis. Tento proces zahrnuje několik klíčových fází: návrh, plánování a implementaci dekompozice aplikací.[18]

Mezi hlavní výhody patří:

- Škálovatelnost – mikroservisy umožňují jednotlivým komponentám škálovat nezávisle, což vede k lepšímu využití zdrojů a optimalizaci výkonu.
- Flexibilita – vývojové týmy mohou pro různé služby volit různé technologie a frameworky, což podporuje inovace a experimentování.
- Odolnost – architektura mikroservis zlepšuje izolaci chyb. Selhání jedné služby je méně pravděpodobné, že způsobí pád celé aplikace.
- Rychlejší nasazování – menší, nezávislé služby lze nasadit rychleji a častěji, což vede k rychlejšímu doručování nových funkcí a aktualizací.
- Zlepšená autonomie týmů – týmy mohou pracovat na různých službách nezávisle, což snižuje úzká místa a zvyšuje produktivitu.

Na druhé straně migrace přináší několik nevýhod:

- Složitost – správa a orchestrace množství mikroservis může být složitá a vyžaduje robustní DevOps postupy.
- Latence – zvýšená komunikace mezi službami může vést k vyšší latenci a režii.

- Výzvy v zajištění konzistence – zajištění konzistence dat napříč mnoha službami může být obtížné a vyžaduje pečlivé plánování a použití vhodných strategií.
- Bezpečnost – zvýšený počet služeb a jejich interakcí rozšiřuje útočnou plochu, což zvyšuje nároky na bezpečnost.
- Vyšší náklady na provoz – mikroservisy často vedou k vyšší spotřebě zdrojů kvůli nutnosti více instancí a kontejnerů, což může zvýšit provozní náklady.

Google Cloud poskytuje různé nástroje a služby pro usnadnění této migrace:

- Google Kubernetes Engine (GKE): umožňuje orchestraci a správu kontejnerů, což je klíčové pro nasazování mikroservis.
- Anthos: nabízí hybridní a multi-cloud platformu pro konzistentní provoz mikroservis napříč prostředími.
- Cloud Run: poskytuje plně spravované prostředí pro provoz kontejnerizovaných aplikací.
- Cloud Endpoints: spravuje a zabezpečuje API, což je zásadní pro komunikaci mezi mikroservisy.
- Istio: síťová služba, která poskytuje nástroje pro správu mikroservis, včetně řízení provozu, bezpečnosti a sledování.
- Cloud Pub/Sub: podporuje asynchronní posílání zpráv mezi mikroservisy, což zvyšuje oddělení a odolnost.

2.2.2 Atlassian Compass

Atlassian Compass je nástroj navržený pro efektivní správu architektury mikroservis. Nabízí různé funkce zaměřené na zlepšení viditelnosti, výkonu a spolehlivosti mikroservis. Mezi hlavní výhody patří:

- Centralizovaná správa – compass poskytuje centralizovanou platformu pro sledování a správu všech mikroservis, což usnadňuje udržování přehledu o stavu a zdraví systému.
- Zlepšená spolupráce: umožňuje lepší spolupráci mezi vývojovými týmy díky nástrojům pro dokumentaci, komunikaci a koordinaci.
- Metriky a monitorování – nástroj se integruje s různými monitorovacími a logovacími službami, aby poskytoval aktuální přehledy a metriky o výkonu mikroservis.
- Sledování závislostí – pomáhá vizualizovat a spravovat závislosti mezi různými službami, což je klíčové pro pochopení dopadu změn a vyhnutí se kaskádovým selháním.

- Standardizace – podporuje standardizaci postupů a nástrojů v celé organizaci, což vede k konzistentnější a spolehlivější správě služeb.

Nevýhody používání Atlassian Compass zahrnují:

- Složitost – implementace a integrace Compass s existujícími systémy může být složitá a vyžadovat značné úsilí a odborné znalosti.
- Křivka učení – ýmy mohou čelit křivce učení při přijetí a efektivním využívání nástroje, což může dočasně zpomalit produktivitu.
- Náklady – náklady spojené s používáním Compass mohou být značné v závislosti na velikosti organizace a rozsahu použití.
- Omezení přizpůsobení – i když Compass nabízí mnoho funkcí, nemusí pokrýt všechny specifické potřeby každé organizace, což může vyžadovat dodatečný vývoj nebo integraci s jinými nástroji.
- Škálovatelnost – správa velkého počtu mikroservis s Compass může být náročná, pokud není správně nakonfigurována, a mohou se objevit problémy s výkonem.

Hlavní funkce Atlassian Compass zahrnují:

- Katalog služeb: komplexní katalog všech mikroservis, včetně jejich vlastnictví, závislostí a dokumentace.
- Scorecards: automatizované skóre k hodnocení zdraví a výkonu mikroservis na základě předem definovaných metrik a kritérií.
- Správa životního cyklu komponent: nástroje pro správu životního cyklu mikroservis od vývoje i po deprekaci.
- Možnosti integrace: bezešvá integrace s populárními nástroji CI/CD, monitorovacími řešeními a dalšími součástmi ekosystému Atlassian, jako jsou Jira a Confluence.
- Upozornění a notifikace: konfigurovatelné upozornění a notifikace, které udržují týmy informované o stavu a problémech souvisejících s jejich službami.

2.2.3 vFunction

vFunction je platforma, která využívá algoritmy umělé inteligence k analýze monolitických aplikací a automatizaci jejich migrace na architekturu mikroservis. Tento proces má za cíl modernizaci zastaralých systémů a zlepšení jejich škálovatelnosti, výkonu a udržovatelnosti.[19] Mezi klíčové funkce patří:

- Automatizovaná analýza kódu – vFunction používá AI algoritmy k analýze stávajícího kódu monolitických aplikací, identifikaci komponent a závislostí, které lze modularizovat.

- Doporučení pro dekompozici – platforma poskytuje doporučení pro rozdělení monolitických aplikací na menší, spravovatelné mikroservisy na základě analýzy.
- Podpora refaktoringu – vFunction pomáhá při procesu refaktoringu, čímž zajišťuje, že nová architektura mikroservis zachovává funkčnost a integritu původní aplikace.
- Hodnocení škálovatelnosti – nástroj hodnotí potenciál škálovatelnosti refaktorovaných mikroservis, což organizacím pomáhá plánovat budoucí růst.
- Možnosti integrace – vFunction se integruje s různými CI/CD pipeline a DevOps nástroji, aby zjednodušil proces migrace.

Výhody použití vFunction zahrnují:

- Efektivita – automatizace procesu analýzy a migrace šetří značný čas a zdroje ve srovnání s manuálními metodami.
- Přesnost – AI-driven analýza může identifikovat složité závislosti a potenciální problémy, které by mohly být přehlédnuty lidskými vývojáři.
- Škálovatelnost – platforma usnadňuje vytvoření škálovatelných mikroservis, což umožňuje efektivnější využití zdrojů.
- Snížené riziko – díky poskytování podrobných přehledů a doporučení vFunction snižuje rizika spojená s migrací komplexních aplikací.
- Modernizace zastaralých systémů – pomáhá modernizovat zastaralé systémy, což je činí přizpůsobivějšími současným technologickým standardům a obchodním potřebám.

Mezi nevýhody patří:

- Počáteční nastavení: nastavení a konfigurace vFunction může vyžadovat značné úsilí a odborné znalosti.
- Křivka učení: týmy mohou potřebovat čas na seznámení se s platformou a jejími funkcemi.
- Náklady: náklady spojené s použitím vFunction mohou být značné v závislosti na velikosti a složitosti aplikací.
- Výzvy při integraci: integrace vFunction s existujícími vývojovými a nasazovacími workflow může představovat výzvy.
- Závislost na nástroji: organizace se mohou stát závislými na nástroji pro budoucí migrace a refaktoringy, což může omezit flexibilitu.

2.2.4 IBM Mono2Micro

IBM Mono2Micro je inovativní nástroj, který využívá technologie umělé inteligence k transformaci tradičních monolitických aplikací na architekturu mikroservis. Tato transformace má za cíl modernizaci zastaralých systémů, zlepšení jejich flexibility, škálovatelnosti a udržovatelnosti.[20] Klíčové funkce zahrnují:

- Automatizovaná analýza aplikací – Mono2Micro používá AI algoritmy k analýze stávající monolitické aplikace, identifikaci logických komponent a závislostí, které lze modularizovat.
- Doporučení pro mikroservisy – nástroj poskytuje doporučení pro dekompozici monolitu na menší, spravovatelné mikroservisy na základě analýzy.
- Vizualizační nástroje – obsahuje vizualizační nástroje, které pomáhají vývojářům rozumět struktuře jejich aplikací a navrhované architektuře mikroservis.
- Podpora refaktoringu – Mono2Micro podporuje proces refaktoringu, čímž zajišťuje, že nová architektura mikroservis zachovává původní funkčnost a integritu aplikace.
- Možnosti integrace – nástroj se integruje s různými vývojovými a nasazovacími pipeline, aby usnadnil hladký přechod na architekturu mikroservis.

Výhody použití IBM Mono2Micro zahrnují:

- Efektivita – automatizace procesu analýzy a transformace šetří značný čas a zdroje ve srovnání s manuálními metodami.
- Přesnost – AI-driven analýza může odhalit složité závislosti a potenciální problémy, které by mohly být přehlédnuty lidskými vývojáři.
- Škálovatelnost – výsledná architektura mikroservis může být snadněji škálovatelná, aby vyhovovala rostoucím obchodním potřebám.
- Snížené riziko – díky poskytování podrobných přehledů a doporučení Mono2Micro snižuje rizika spojená s transformací komplexních aplikací.
- Modernizace zastaralých systémů – pomáhá modernizovat zastaralé systémy, což je činí přizpůsobivějšími moderním technologickým standardům a obchodním požadavkům.

Mezi nevýhody patří:

- Počáteční nastavení – počáteční nastavení a konfigurace Mono2Micro může vyžadovat značné úsilí a odborné znalosti.
- Křivka učení – vývojové týmy mohou potřebovat čas na seznámení se s nástrojem a porozumění jeho výstupům.

- Náklady – náklady spojené s použitím Mono2Micro, zejména u velkých a složitých aplikací, mohou být značné.
- Výzvy při integraci – integrace Mono2Micro s existujícími vývojovými a nasazovacími workflow může představovat výzvy.
- Závislost na nástroji – organizace se mohou stát závislými na nástroji pro budoucí transformace a refaktoring, což může omezit flexibilitu.

Ačkoliv nástroje jako Google Cloud, Atlassian Compass, vFunction a IBM Mono2Micro nabízejí významné přínosy pro migraci monolitických aplikací na mikroservisy, žádný z těchto nástrojů není bez nedostatků: složitost implementace, vyšší provozní náklady, křivka učení a výzvy při integraci jsou faktory, které nelze přehlížet. Tyto nedostatky zdůrazňují potřebu důkladného zvážení při výběru nástroje a přístupu k migraci.[21] Tato bakalářská práce se zaměřuje na prozkoumání těchto nástrojů, identifikaci jejich výhod a nevýhod a navržení optimální strategie pro migraci monolitických aplikací na mikroservisy. Cílem je poskytnout komplexní pohled na současné možnosti a přispět k lepšímu porozumění a využití těchto technologií v praxi.

Kapitola 3

Analýza

V následující kapitole je uveden základ funkční části této bakalářské práce, zejména jsou popsány funkční a nefunkční požadavky a technologická řešení uvedených požadavků.

3.1 Funkční požadavky

Tato sekce se podrobně věnuje funkčním požadavkům, které definují specifické operace a funkcionalitu, jež musí systém splňovat, aby úspěšně podporoval transformaci monolitické architektury do mikroservisů. Tyto požadavky jsou základními stavebními kameny pro navrhování a implementaci funkčních aspektů systému, zajistí správnou integraci komponent a bezproblémovou operativnost celého řešení. Tato část poskytne jasný přehled o tom, co systém musí dokázat, aby byl považován za efektivní a splnil požadavky na moderní softwarové řešení.

3.1.1 Seznam funkčních požadavku

- **FR1 – Analýza kódu a správa tříd**

  Systém by měl po celou dobu transformace uchovávat zdrojový kód a získávat podrobné informace o třídách, metodách a vlastnostech, které jsou nezbytné pro následující kroky transformace.

- **FR2 – Přepracování kódu podle konfigurace**

  Podle určitých omezení a konfiguračních pravidel (1:1, 1:N, N:M) musí být systém schopen upravit stávající metody, proměnné a vlastnosti.

- **FR3 – Generování nových tříd a konfigurací**

  Automatické generování nových tříd a konfiguračních souborů, které splňují požadavky nově vytvořené mikroslužby.

- **FR4 – Oddělení obchodní logiky**

  Rozdělení funkčnosti a obchodní logiky mezi různé mikroservisy podle nové architektury.

- **FR5 – Obnova a správa vazeb**

  Identifikace a obnova ztracených nebo porušených vazeb mezi třídami a službami je velmi důležitá pro zachování integrity systému.

- **FR6 – Konfigurace interakcí mezi službami**

  Nastavení efektivní komunikace mezi mikroslužbami, včetně implementace a konfigurace vrstvy Backend-For-Frontend (BFF).

- **FR7 – Správa kódu v úložištích projektu**

  Systém by měl podporovat ukládání a správu kódu v příslušných repozitářích pro každou mikroslužbu, což zajistí lepší správu a izolaci služeb.

- **FR8 – Možnost vytvoření konfiguračního .yaml souboru**

  Systém by měl podporovat ukládání a správu konfiguračního souboru decomposition.config.yaml

- **FR9 – Ukládání vstupního projektu do resoursu**

  Systém by měl podporovat ukládání vstupního projektu do resource složky input a měl by mít přístup k těmto resorsům

3.1.2 Technologické řešení funkčních požadavků

- **Analýza kódu a správa tříd**

  Úkol: Analýza a úprava zdrojového kódu jazyka Java pro automatickou transformaci monolitické architektury na mikroslužby.

  Technologie: JavaParser.

  Realizace: Programové čtení a úprava tříd, metod a anotací, aby odpovídaly struktuře mikroslužeb a umožnily jejich nezávislou funkčnost.

- **Přepracování kódu podle konfigurace**

  Úkol: Čtení a zápis souborů YAML pro nastavení mikroservis a jejich vzájemných interakcí.

  Technologie: Jackson YAML.

  Realizace: Použití knihovny Jackson pro zpracování YAML souborů, které popisují strukturu a závislosti mikroservis.

- **Generování nových tříd a konfigurací**

  - Úkol: Správa závislostí projektu a adaptace konfigurace projektu pomocí Maven souborů pom.xml.
    Technologie: Maven XPP3.
    Realizace: Úprava a vytváření souborů pom.xml pro konfiguraci projektů Maven, aktualizace závislostí podle potřeb jednotlivých mikroservis.

   - Úkol: Čtení, úprava a ukládání konfiguračních souborů .properties, které jsou nezbytné pro konfiguraci aplikace.

     Technologie: Apache Commons Configuration2.

     Realizace: Zpracování souborů .properties pro nastavení aplikace, včetně konfigurací databází, portů serveru a dalších environmentálních nastavení.

- **Konfigurace interakcí mezi službami**

  Úkol: Nastavení směrování a řízení přístupu k funkcím mikroservis prostřednictvím centralizovaného BFF.

  Realizace: Vyvinutí BFF, které agreguje a řídí požadavky na mikroservisy, optimalizace vazeb 1:1 a 1:N, a řízení složitějších vazeb N:M na úrovni frontendu.

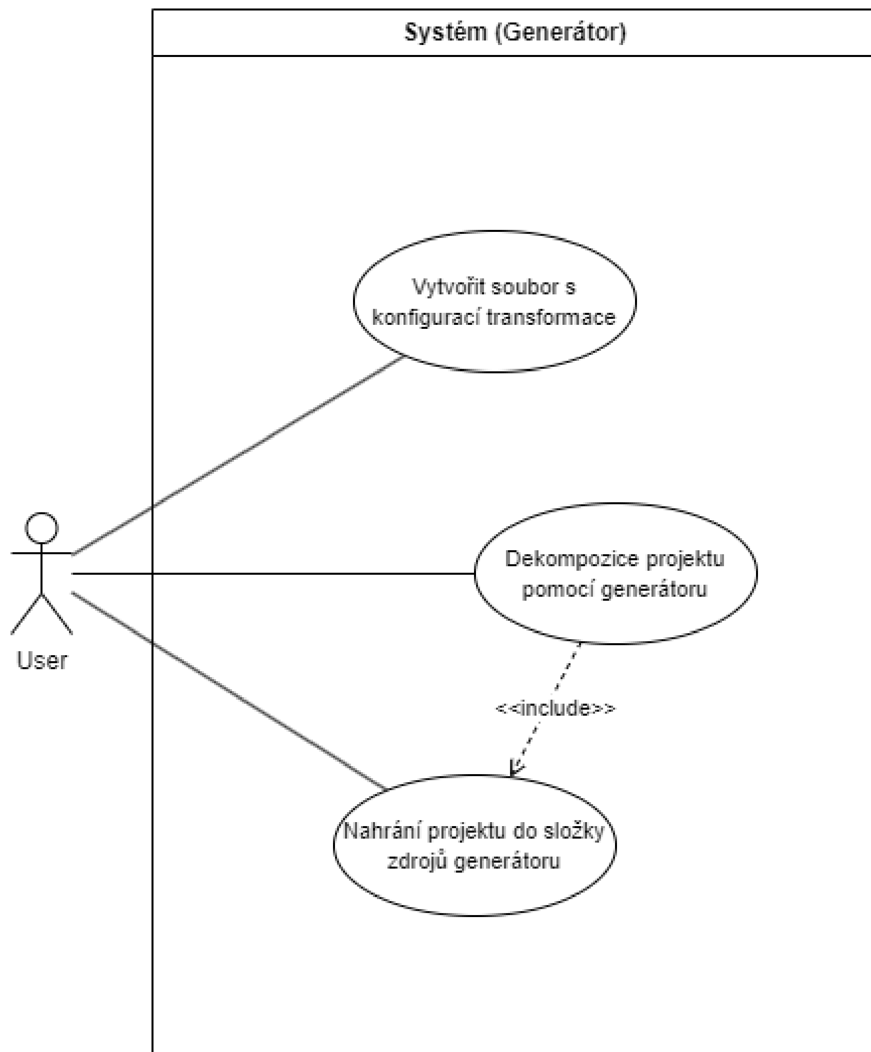
3.2 Nefunkční požadavky

- NFR1 Sýstem je spustitelný na libovolném zařízení které má Windows.

- NFR2 Sýstem je spustitelný na libovolném zařízení s Java Virtual Machine

3.3 UseCase Diagram

Na obrázku 3.1. je popsán diagram případu užití zachycující funkční požadavky do aplikace.

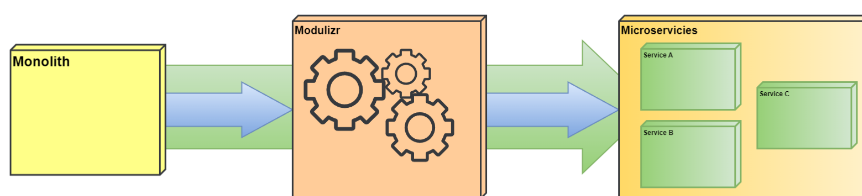


Obrázek 3.1: Use Case Diagram

Kapitola 4

Návrh

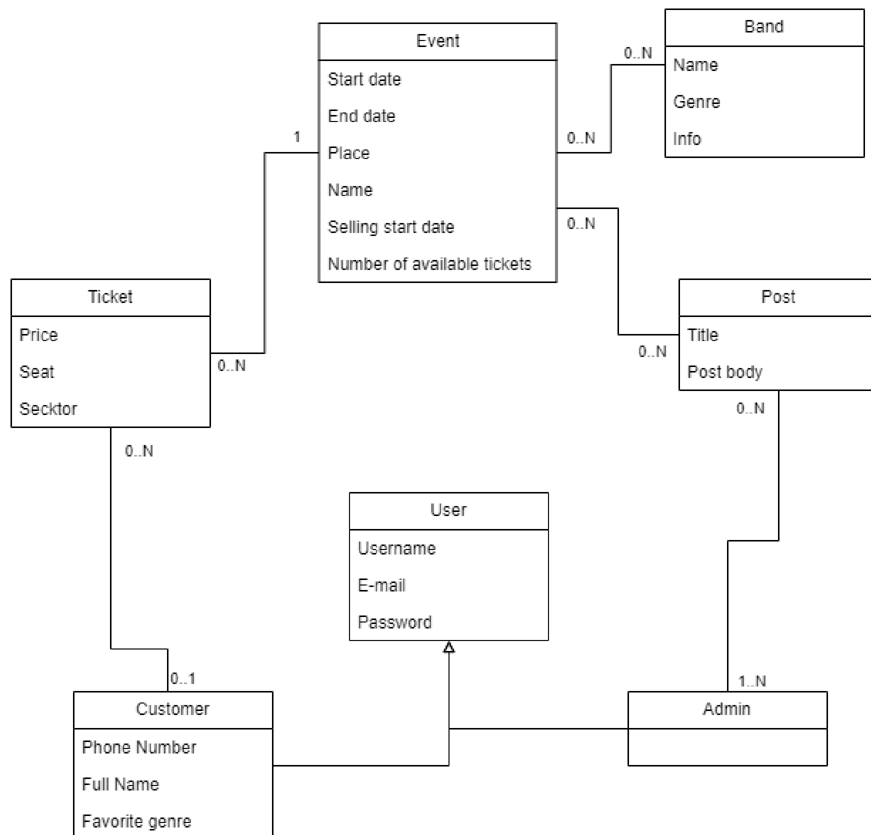
V této kapitole se zaměříme na návrh transformačního procesu z monolitické architektury na architekturu založenou na mikroservisách. Popíšeme hlavní komponenty systému, jejich vzájemné vztahy a změny, které jsou nezbytné pro přechod na mikroservisní architekturu. Nasledujici obrazek 4.1 reprezentuje obecne princip dekompozice projektu.



Obrázek 4.1: *Obecný princip fungování applikace*

4.1 Struktura aplikace před transformací

Na začátku je důležité pochopit a zdokumentovat existující monolitickou strukturu aplikace. K tomu využijeme klasický UML třídní diagram, který ukáže hlavní komponenty systému, jejich atributy a metody, stejně jako vztahy mezi nimi. Diagram představí základní bloky, jako jsou uživatelské rozhraní, databázové vrstvy a zpracování logiky. Zvolený demonstrační projekt představuje sebou standardní Spring Boot aplikace, ktera jako databázi využívá In-memory H2 databaze, demonstrační aplikace obsahuje funkcionalitu online obchodu vstupenek, business model tohoto projektu je představen na obrázku



Obrázek 4.2: *ClassDiagram demonstrační aplikace do transformace*

4.2 Komponenty systému

Projekt Modulizr je Java aplikace nabízející částečnou automatizace transformace monolitické Spring Boot aplikaci do mikroservis. Aplikace obsahuje 5 funkčních modulu, které vykonávají základní funkcionalitu projektu.

- **ProjectDecomposer:**

  Agregační modul pro zajištění pořadí provedení práce. Akumuluje funkcionalitu pro komunikaci s uživateli, řízení ostatních modulu a provedení načtení a ukladaní souboru.

- **ProjectGenerator:**

  Modul pro generování projektových baličku, souboru, resoursu a úpravu konfiguračních prvků.

- **BFFGenerator:**

  Modul dědící od ProjectGeneratoru veškerou funkcionalitu, přidaná funkcionalita slouží k zavedení Backend For Frontend vrstvy.

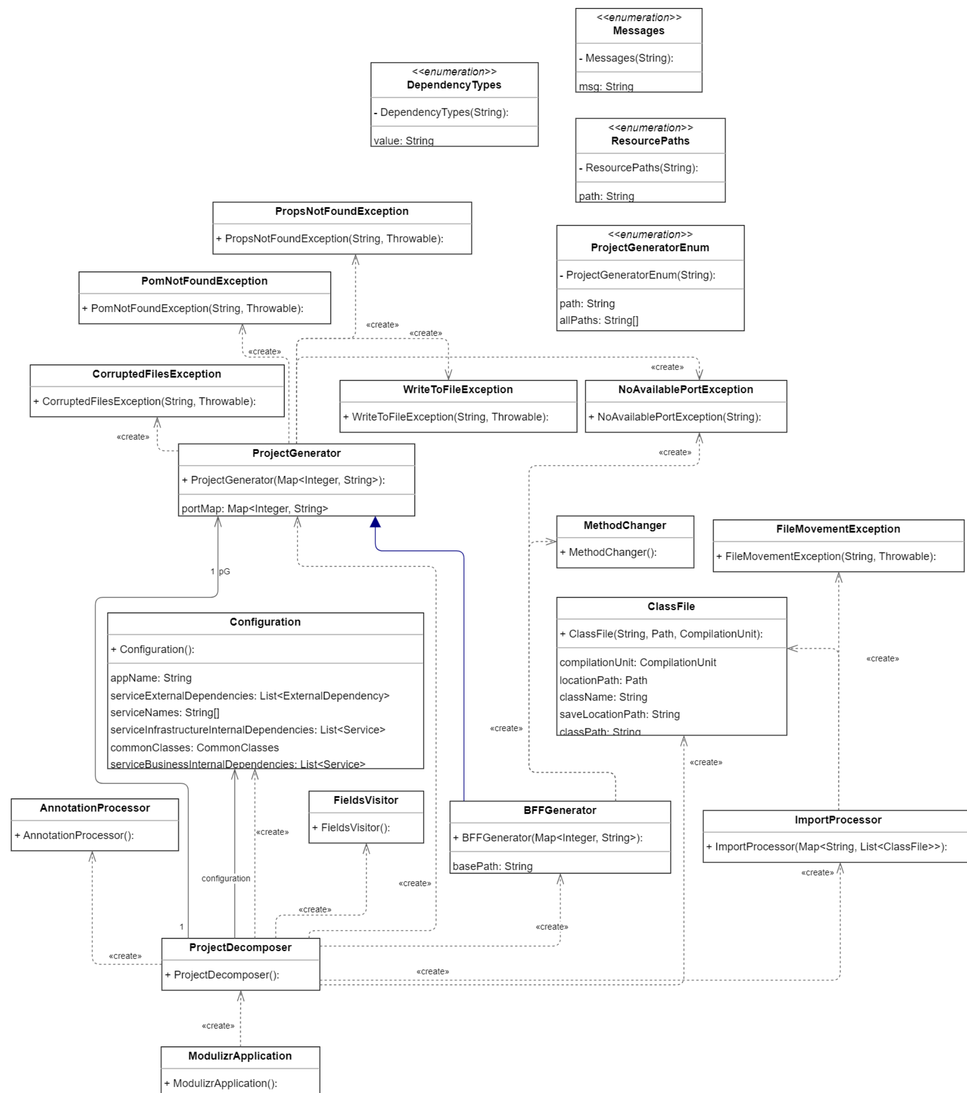
- **Configuration:**

  Datový model konfigurace transformace projektu.

- **ImportProcessor:**

  Refaktorizační modul, zajíštujici pristupnost veškere potřebné pro zadanou servisu funkcionality a obnovuje vazby mezi třídy uvnitř projektu.

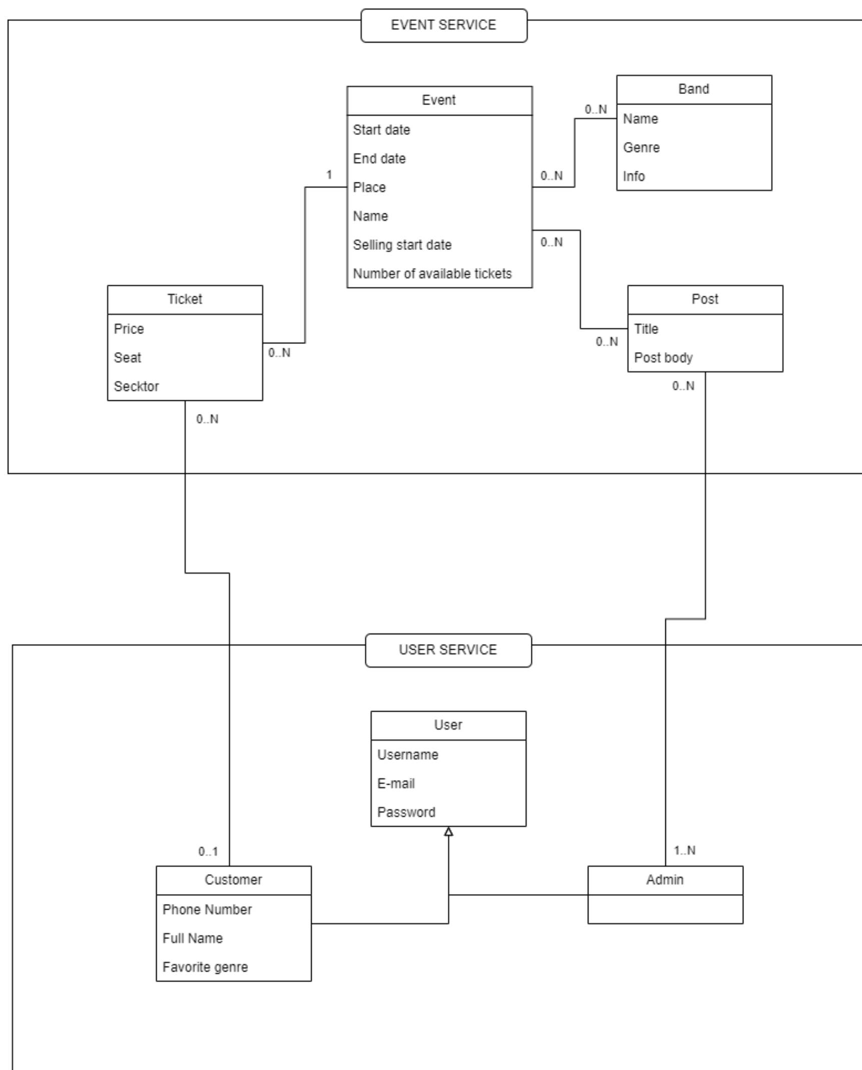
Veskery teto moduly spolu dovoluji systemu dekomponovat vstupní projekt na zvolene uzivatelem servisy. Celý class diagram výsledného projektu Modulizr je zobrazen na obrázku 4.3



Obrázek 4.3: Class Diagram aplikace Modulizr

4.3 Struktura aplikace po transformací

Zvolený demonstrační projekt po transformace představuje sebou tři spustitelné aplikací, kteří využivají vlastni In-memory H2 databazi, rozdelený business model tohoto projektu je představen na obrázku 4.4



Obrázek 4.4: Mikrosevisní class Diagram Demonstrační aplikace

Kapitola 5

Implementace

V této kapitole je podrobně popsána implementační fáze projektu. Provedeme analýzu procesu implementace, přičemž se zaměříme na naražené problémy, jejich aktuální řešení, a detailně probereme chyby, které byly během vývoje učiněny. Dále se podíváme na možné změny, které by mohly projekt vylepšit, a diskutujeme o budoucí práci, jako jsou doplňky nebo refaktoring stávajícího řešení.

5.1 Průběh Implementace

Během fáze implementace jsem se narazili na řadu technických a organizačních výzev, které vyžadovaly inovativní přístupy k řešení. Například, integrace nových mikroservis do stávajícího ekosystému přinesla nejen technické, ale i komunikační bariéry mezi týmy, které bylo třeba překonat.

5.1.1 Parsovani input projektu

K parsovani projektu, sestavení objektového modelu a analýze zdrojového kódu byla použita knihovna JavaParser. Zpočátku byla tato knihovna použita k vytvoření mapy souborů projektu, která zajišťuje ukládání informací o názvech souborů a jejich umístění a obsahuje také kompilační jednotku, což je popsané v příkladu kódu 5.1. Navzdory své stabilitě a snadnému vytváření neobsahuje mapa souborů analýzy při načítání souborů, což omezuje její funkčnost. [22, 17] Na rozdíl od mapy souborů představuje grafová struktura realističtější zobrazení struktury projektu, protože zohledňuje vztahy mezi soubory v podobě hran grafu. To usnadňuje analýzy s nízkou vazbou, což umožňuje sofistikovanější technické zpracování projektu. Tento přístup také umožňuje další rozšíření analýzy, včetně modulů pro vytěžování textu. Přechod od mapy souborů ke grafové struktuře je tedy výrazným zlepšením, které poskytuje hlubší a komplexnější analýzu projektu.[16]

```
1    private void processSourceFiles(List<Path> listFiles) {
2        System.out.println(Messages.Reading.getMsg());
3        listFiles.forEach(sourcePath -> {
4            try {
5                jPs.parse(sourcePath).ifSuccessful(cU -> {
6                    configurationApplication(cU);
```

```
7                       String serviceName = extractServiceName(cU);
8                       if (!serviceName.isEmpty()){
9                           List<ClassFile> serviceClasses = classMap.
    getOrDefault(serviceName, classMap.get("common"));
10                          ClassFile classFile = new ClassFile(
    getClassName(cU).getFirst(), sourcePath, cU);
11                          String output = classReplacement(sourcePath
    , classFile, serviceName);
12                          serviceClasses.add(classFile);
13                          if (!output.isEmpty()){
14                              classFile.saveClassFile(output);
15                          } else {
16                              classFile.setSaveLocationPath(
    sourcePath.toString());
17                          }
18                      }
19                  });
20              } catch (IOException e) {
21                  System.err.println(Messages.CorruptedFiles.getMsg()
     + "\n" + e.getMessage());
22              }
23          });
24      }
```

Listing 5.1: Métoda parsování Java souboru

5.1.2 Konfigurace transformace

Během realizace projektu je klíčová konfigurační fáze transformace, která zajistí, že přechod z monolitického systému na architekturu mikroslužeb bude přizpůsoben konkrétním potřebám aplikace. Konfiguraci lze přidat buď prostřednictvím konfiguračního souboru YAML uloženého ve zdrojích projektu, což umožňuje předem definovat všechny potřebné parametry transformace, nebo interaktivně v konzoli při spuštění aplikace, kde může uživatel dynamicky zadávat specifikace. Tento přístup umožňuje přesně splnit požadavky uživatele. Byli použity knihovny Jackson ke zpracování souborů YAML a vytvoření konfiguračních třídy, ktera je popsana v přikladu kódu 5.2. Textová analýza může výrazně zlepšit proces konfigurace při transformaci softwarových systémů tím, že poskytne nástroje pro hlubokou analýzu a extrakci zásadních obchodních logik z existujícího zdrojového kódu.[21] Využitím pokročilých algoritmů textové analýzy lze odhalit vzorce a závislosti, které jsou skryté při běžném low-coupling analytickém přístupu. Tento proces může být klíčový pro efektivní formulaci strategií rozdělení komponent aplikace do samostatných mikroservis, což je zásadní krok v procesu modernizace softwarové architektury. Vyvinutí takové technologie jako integrovaného modulu do aplikace by mohlo značně urychlit celý proces transformace. Nejen že by se zvýšila přesnost a hloubka analytického zpracování, ale také by se optimalizovala celková efektivita přechodu na mikroservisní architekturu. Díky tomu by byl proces méně náchylný k potenciálním chybám, které mohou vzniknout v důsledku nesprávné interpretace nebo neúplné analýzy závislostí a vzájemných vztahů v kódu. Zabývání se tímto rozsáhlým a komplexním úkolem by však mohlo vyža-

dovat značné množství práce a technického zručnosti, což by mohlo posloužit jako obsáhlý základ pro samostatnou práci. Vypracování takového projektu by poskytlo cenné poznatky a praxe v oblasti softwarového inženýrství a mohlo by představovat významný přínos v akademickém i profesionálním kontextu. Na závěr sekcí o konfiguraci transformace je třeba zdůraznit, že navzdory významným možnostem automatizace díky textové analytice a dalším technologickým řešením zůstává role lidského faktoru nezastupitelná. Hluboké pochopení obchodní logiky, strategická vize a schopnost přizpůsobit technologická řešení jedinečným podmínkám konkrétního podniku jsou aspekty, které stroje zatím nejsou schopny plně replikovat. Odborné znalosti a zkušenosti profesionálů zajistí, že při realizaci změn budou zohledněny nejen technické, ale i obchodní aspekty, což zajistí, že transformace podpoří dlouhodobé cíle organizace a přispěje k udržitelnému rozvoji. V další kapitole bude podrobně popsána samotná konfigurace transformace a odhaleny konkrétní metody a přístupy použité k přizpůsobení přechodu na architekturu mikroslužeb.[5]

```
1     @Setter
2     @Getter
3     public class Configuration {
4         @Setter
5         @Getter
6         public static class Service {
7             private String serviceName;
8             private List<String> classes;
9 
10            public Service() {}
11            public Service(String serviceName, List<String> classes
    ) {
12                this.serviceName = serviceName;
13                this.classes = classes;
14            }
15        }
16        @Setter
17        @Getter
18        public static class ExternalDependency {
19            private String source;
20            private String sourceClass;
21            private String dependencyType;
22            private String target;
23            private String targetEndpoint;
24            private List<String> targetClass;
25 
26            public ExternalDependency() {}
27            public ExternalDependency(String sourceClass, String
    dependencyType, String target,
28                                      String targetEndpoint, List<
    String> targetClass) {
29                this.sourceClass = sourceClass;
30                this.dependencyType = dependencyType;
31                this.target = target;
32                this.targetEndpoint = targetEndpoint;
33                this.targetClass = targetClass;
34            }
35        }
```

```
36
37        @Setter
38        @Getter
39        public static class CommonClasses{
40            private List<String> classes;
41
42            public CommonClasses() {}
43            public CommonClasses(List<String> classes) {
44                this.classes = classes;
45            }
46        }
47
48        private String appName;
49        private String[] serviceNames;
50        private List<Service> serviceBusinessInternalDependencies;
51        private List<Service>
    serviceInfrastructureInternalDependencies;
52        private List<ExternalDependency>
    serviceExternalDependencies;
53        private CommonClasses commonClasses;
54
55        public Configuration(){}
56    }
```

Listing 5.2: Configurační soubor

5.1.3 Generování tříd a projektu

Během procesu generování projektu je zvláštní důraz kladen na správu souborů pom.xml a application.properties, které jsou klíčové pro definování závislostí a konfigurace projektu. K tomuto účelu v projektu je zavedena trida "ProjectGenerator". Díky využití MavenXPP3 dochází k flexibilní správě závislostí prostřednictvím dynamických modifikací jmena projektu v pom.xml, což zajišťuje adaptaci projektu, kterou popisuje příklad kódu 5.3. Pomocí modulu Properties ze standardní knihovny java.lang je soubor application.properties přizpůsoben pro každý mikroservis zvlášť, což umožňuje nastavit konfigurační parametry s potřebnou úrovní izolace. Toto je klíčový prvek v zajištění správného nastavení pracovního prostředí aplikací. Obsah metody pro nastavení konfiguračního souboru application.properties ukazuje níže uvědený kod.

```
1     public void pomFileConstruct(Path xml, String basePath, String
    name) throws PomNotFoundException {
2         try (FileReader fileReader = new FileReader(xml.toFile()))
    {
3             Model model = reader.read(fileReader);
4             model.setName(name);
5
6             try (FileWriter fileWriter = new FileWriter(basePath +
    "\\pom.xml")) {
7                 writer.write(fileWriter, model);
8             }
9         } catch (Exception e) {
10            throw new PomNotFoundException( Messages.PomNotFound.
    getMsg(), e);
```

```
11          }
12      }
13
14      public void generateApplicationProps(String sourcePath, String
    basePath, String serviceName)
15              throws NoAvailablePortException, PropsNotFoundException
    , WriteToFileException {
16          Integer port = findFreePort(serviceName);
17          if (Objects.nonNull(port)){
18              Properties props = new Properties();
19              String propertiesFilePath = sourcePath +
    ProjectGeneratorEnum.MainResources.getPath();
20              String targetPath = basePath + "\\" + serviceName
21                      + ProjectGeneratorEnum.MainResources.getPath()
    + "/application.properties";
22              resourceCopy(propertiesFilePath, basePath, serviceName)
    ;
23              try (FileInputStream in = new FileInputStream(
    propertiesFilePath + "/application.properties")) {
24                  props.load(in);
25              } catch (IOException e) {
26                  System.err.println("Error loading properties file:
    " + e.getMessage());
27                  throw new PropsNotFoundException(Messages.
    PropsNotFound.getMsg(), e);
28              }
29
30              propertiesSetter(serviceName, port, props, targetPath);
31          } else {
32              throw new NoAvailablePortException(Messages.NoPort.
    getMsg());
33          }
34      }
```

Listing 5.3: Generator souboru pom.xml a application.properties

Integrace procesu generování projektů s infrastrukturou neustálé integrace a doručování (CI/CD) významně zvyšuje efektivitu vývoje a implementace softwaru. Tento přístup umožňuje automatizaci fází sestavení, testování a nasazování aplikací, což urychluje cykly vydání a zlepšuje kvalitu konečného produktu. Zavedení úzkého propojení mezi generováním projektů a systémy CI/CD zajišťuje neustálou interakci a výměnu dat mezi těmito procesy. Například ihned po generování projektu, který obsahuje soubory pom.xml a application.properties pro správu závislostí a konfigurace, může být projekt automaticky sestaven a otestován v rámci CI pipeline. To umožňuje rychle identifikovat a opravit chyby, čímž se zlepšuje kvalita kódu a urychluje vývoj. Kromě toho je možné implementovat sestavení společného modulu tříd, který bude využíván všemi mikroslužbami projektu. Vytvoření takového společného modulu umožňuje centralizovaně spravovat závislosti a opakovaně použitelné komponenty, což usnadňuje podporu a aktualizace systému. Po vytvoření může být společný modul nasazen ve formě artefaktu, na který se budou odkazovat všechny mikroslužby, čímž se zajišťuje konzistence a snižuje se duplicity kódu v projektu.

5.1.4 Použití Backend For Frontend Patternu

BFFGenerator, vyvinutý jako následník standardního ProjectGeneratoru, hraje velkou roli v procesu vytvaření architektury mikroservisů. BFF-Servisa kterou tato třída vyrábí slouží jako komunikační prostředí a API gateway, umožňující uživatelským rozhraním interakci s komplexním systémem mikroservisů prostřednictvím jediného vstupního bodu. To výrazně zjednodušuje klientovou architekturu a zlepšuje celkový výkon systému, protože snižuje počet potřebných volání a usnadňuje správu provozu.[15, 23] Proces tvorby BFF začíná generováním samostatného projektu prostřednictvím třídy BFFGenerator, která dědí vlastnosti základní třídy ProjectGenerator. Během generace nejsou vytvářeny jen základní strukturální prvky projektu, jako jsou adresáře pro kód a testy, ale také se inicializují klíčové konfigurační soubory – pom.xml pro správu závislostí a application.properties pro nastavení provozních parametrů. Dále proces zahrnuje vytvoření webových klientů pro každý mikroservis, které jsou konfigurovány tak, aby co nejefektivněji komunikovali s ostatními mikroservisy. Zvláštní pozornost je věnována bezpečnosti, ovladatelnosti a škálovatelnosti požadavků. Posledním krokem je vytvoření BFF kontroleru, který agreguje všechny metody a endpointy, poskytující sjednocené API pro frontendovou část. Tento proces je popsán v příkladu kódu 5.4 To nejen zvyšuje efektivitu zpracování požadavků, ale také výrazně usnadňuje monitorování a škálování systému jako celku.

```
1    public void bffServiceGeneration(Map<String, List<ClassFile>>
    classMap) {
2        createProjectStructure(getBasePath());
3        pomFileCreation();
4        generateApplication(servicePath, name);
5        try {
6            generateApplicationProps(getBasePath(), name);
7        } catch (NoAvailablePortException | WriteToFileException e)
     {
8            System.err.println(e.getMessage());
9        }
10       clientGeneration();
11       controllerGeneration(classMap);
12   }
```

Listing 5.4: BFF

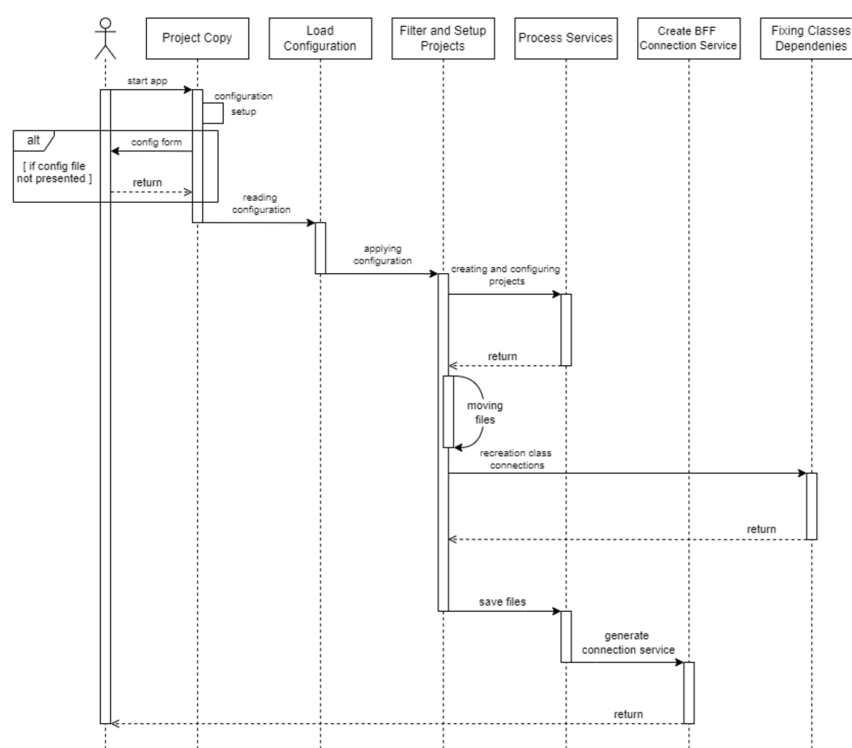
5.1.5 Import Processing

Následujícím krokem bylo zavedení mechanismu pro obnovu vazeb mezi třídami v rámci služby, který se aktivuje, když dojde k přerušení vazeb v důsledku změn v kódu nebo strukturálních úprav projektu. Jeho hlavním úkolem je identifikace ztracených nebo poškozených vazeb a jejich obnova. Obnova probíhá prostřednictvím dynamické analýzy závislostí a vztahů mezi třídami, přičemž algoritmus využívá informace z použitých komponent, specificky z importů a vlastností tříd. Tímto způsobem získává údaje o přerušených vazbách jak uvnitř, tak mimo danou službu. Aby se předešlo běžným chybám v importech, metoda systematicky prochází třídy v rámci téže mikroslužby, snaží

se lokalizovat a nahradit poškozené importy a zároveň odstraňuje všechny importy tříd z externích služeb, aby se eliminovaly potenciální konflikty a chyby způsobené nevhodnými vzájemnými závislostmi. Tento proces zajišťuje, že všechny komponenty systému zůstanou integrovány a funkční i po provedení změn v projektu.

5.2 Proces Dekompozice

V této podkapitole se bude zkoumán proces dekompozice, budou označené splnění funkčních požadavků a taky bude položena otázka jestli projekt byl vypracován úspěšné nebo ne. Proces Dekompozice je ilustrován na obrázku 5.1.



Obrázek 5.1: Seuence Diagram

FR1 – Analýza kódu a správa tříd: Projekt využívá knihovnu JavaParser pro syntaktickou analýzu zdrojového kódu, což umožňuje podporovat a analyzovat informace o třídách, metodách a vlastnostech v reálném čase.

FR2 – Přepracování kódu podle konfigurace: V projektu je implementována schopnost dynamicky modifikovat soubory pom.xml a application.properties, což umožňuje přizpůsobit kód a závislosti změnám v konfiguraci.

FR3 – Generování nových tříd a konfigurací: Projekt automaticky generuje nové třídy a konfigurační soubory, které splňují specifikace nových mikroslužeb, s použitím předdefinovaných šablon a parametrů.

FR4 – Oddělení obchodní logiky: Projekt podporuje funkční oddělení obchodní logiky a distribuci odpovědnosti mezi jednotlivé mikroslužby, což podporuje lepší škálovatelnost a údržbu kódu.

FR5 – Obnova a správa vazeb: Integrovaný systém pro detekci a obnovu vazeb využívá algoritmy pro analýzu závislostí mezi třídami, což pomáhá obnovovat funkčnost po změnách v projektu.

FR6 – Konfigurace interakcí mezi službami: Implementace Backend-For-Frontend (BFF) slouží k organizaci efektivní komunikace mezi frontendem a mikroslužbami, což zlepšuje výkon a správu dat.

FR7 – Správa kódu v repozitářích projektu: Systém podporuje organizaci a ukládání kódu ve vhodných repozitářích pro každou mikroslužbu, což zajišťuje lepší izolaci a bezpečnost.

FR8 – Vytvoření konfiguračního souboru .yaml: Projekt umožňuje vytváření a správu konfiguračního souboru decomposition.config.yaml, což umožňuje předem definovat všechny potřebné parametry pro transformaci systému.

FR9 – Ukládání vstupního projektu do zdroje: Projekt zajišťuje nahrávání vstupních dat projektu z resursní složky, což podporuje snadný přístup a správu dat.

5.3 Úspěšnost Projektu

Projekt implementuje veškeré funkční požadavky, což potvrzuje jeho efektivitu v oblasti modernizace architektury mikroslužeb. Schopnost generovat a konfigurovat nové mikroslužby, stejně jako jeho hluboká integrace analytických a automatizačních nástrojů pro adaptaci a optimalizaci kódu, dělá z tohoto projektu důležitý nástroj ve správě složitých softwarových systémů.

Kapitola 6

Konfigurace

Ke spravnemu fungovani aplikace Modulizr od uzivatele se vyzaduje prehled o konceptu fungování aplikace kterou bude uzivatel transformovat a je potřeba vytvořit konfiguraci transformaci. Samotnou konfiguraci uživatel si muze zadat dvema zpusoby: Bud si vytvoří soubor decomposition.config.yaml, anebo si to zada rucne v konsoli. Pro jasnější přehled na zkoumani si vezmeme příklad yaml souboru, který byl využit pri implementaci i testovani. Cilem dekompozice je aplikace založená na platformě Spring Boot, jako datové úložiště vyuziva In-memory variantu H2 database, business model demonstracni aplikace je popsan v kapitole číslo 4.

6.1 Demonstrační konfigurace dekompozice

Demonstrační konfigurační soubor je součástí odevzdaného projektu, v této sekci se jedná jenom o obecný popis informaci v konfiguračním souboru.

Konfigurační struktura souboru "decomposition.config.yaml".

- **appName**

  Název monoliticke aplikace určenou pro transformace;

- **serviceNames**

  Soupis servis na ktere bude rozdělena monolitická aplikace. Každá služba představuje samostatnou funkční jednotku.

- **serviceBusinessInternalDependencies**

  Seznam objektu reprezentujicich trid business modelu, ktere patri k specifikované servise. Tyto třídy jsou přímo spojené s obchodní logikou vaší služby.

  Reprezentace ve formátu .yaml:

```
1          serviceName: SERVICE NAME
2          classes:
3                 - CLASSNAME
4                 - …
5
```

- **commonClasses**

  Objekt reprezentujici seznam celoprojektovych trid ktery patri do vsech servis. Mohou to být utility, výjimky nebo základní třídy, které nejsou specifické pro konkrétní službu, ale jsou nezbytné pro fungování celé aplikace.

  Reprezentace ve formátu .yaml:

```
         classes:
                - CLASSNAME
                - …

```

- **serviceInfrastructureInternalDependencies**

  Seznam objektu reprezentujici tridy netykajici se business modelu, ale jsou nezbytné pro fungování služeb (například bezpečnostní konfigurace, utility pro autentizaci).

  Reprezentace ve formátu .yaml:

```
        serviceName: SERVICE NAME
        classes:
            - CLASSNAME
            - …

```

- **serviceExternalDependencies**

  Seznam objektu reprezentujicich vazby mezi servisy.

  - source: Služba-zdroj, která iniciuje závislost,
  - sourceClass: Třída ze služby-zdroje,
  - dependencyType: Typ závislosti (1:1, 1:N, N:M),
  - target: Cílová služba,
  - targetEndpoint: Koncový bod cílové služby,
  - targetClass: Třídy z cílové služby, na které je závislost;

  Reprezentace ve formátu .yaml:

```
        source: SERVICE NAME
        sourceClass: CLASSNAME
        dependencyType: (1:1, 1:N, N:M)
        target: SERVICE NAME
        targetEndpoint: ENDPOINT
        targetClass:
            - CLASSNAME
            - …

```

Kapitola 7

Regresní testování

Pro provedení testů byl použit ruční přístup testování. Při testování bylo ověřováno několik klíčových výstupů programů. Mezi takové výstupy patří:

- Po dekompozici výstupní aplikace se spouští bez chyb.
- Množina vytvořených databází zahrnuje množství informací, které byly v původní aplikaci.
- Funkčnost výstupní aplikace po dekompozici je identická s funkčností vstupní aplikace s monolitickou architekturou.

Testování proběhlo na aplikaci popsané v kapitole 4. Očekávané výstupy po běhu aplikace Modulizr mají následující vlastnosti:

- Tři mikroservisy (Dvě funkční servisy a BFF servisa)
- Dvě databáze pro každou mikroservisu
- Stejná funkčnost

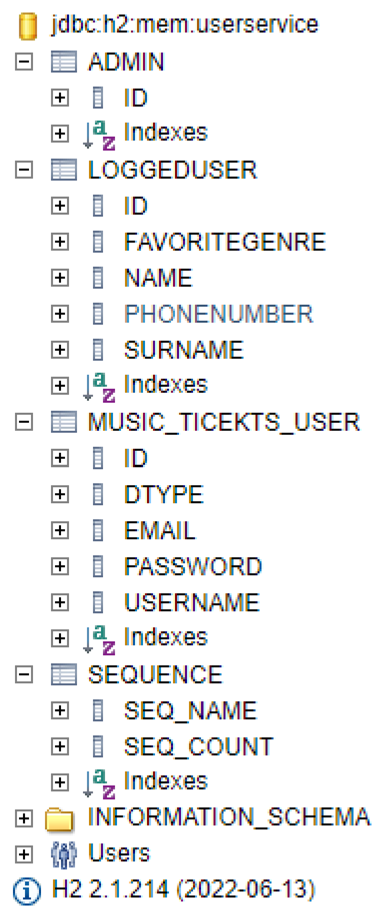
Aktuální výstupy aplikace ukazují úplnou funkčnost aplikace Modulizr. Podle definovaných kritérií Modulizr vytvořil dvě mikroservisy pro dvě služby a jendu kommunikační mikroslužby. Jedna mikroservisa je zodpovědná za správu uživatelů, druhá za správu událostí (Eventů).

Dvě databáze zpracované aplikace umožňují uchovat identické informace jako v databázi v aplikaci před zpracováním. Na obrázku 7.1 je možné vidět příklad databáze pro mikroservis zaměřený na uživatele (User).

Výsledná aplikace lze spustit bez kompilačních a runtime chyb, což zajišťuje splnění první klíčové vlastnosti výstupu.

Funkcionalita výsledné aplikace byla ručně otestována s tím závěrem, že nelze pozorovat zásadní rozdíl ve funkčnosti mezi vstupní a výstupní aplikací.

Tři definované klíčové vlastnosti zpracované aplikace byly splněny, a proto lze tvrdit, že aplikace Modulizr je funkční a testování bylo úspěšné.



Obrázek 7.1: Databáze uživatelů výsledné aplikace

Závěr

Na závěr lze říci, že transformace z monolitu na mikroslužby je komplexní, ale proveditelný proces, který může organizaci přinést významné výhody. Automatizace a šablonování hrají důležitou roli při zjednodušování a zefektivňování tohoto procesu, ale jejich účinnost závisí na hlubokém porozumění konkrétní aplikaci a strategickém přístupu k procesu transformace. Jak již bylo zmíněno, proces transformace lze do určitého bodu automatizovat, ale plně automatizovaný přechod je často nemožný vzhledem k jedinečnosti každé monolitické aplikace. Tato patternizace však pro efektivní aplikaci vyžaduje přizpůsobení a hluboké pochopení specifik každého projektu. Pro úspěšnou transformaci je tedy nutná kombinace jak automatizovaných nástrojů, tak šablonovitých přístupů a individuálních řešení přizpůsobených konkrétní monolitické aplikaci, což je však považováno za proveditelný, i když náročný úkol.

I přesto že zcela automatizovat tento proces se mi nepodařilo, sv§j projekt beru jako úspěšný.

Literatura

[1] VELEPUCHA, Victor; FLORES, Pamela. Monoliths to microservices-migration problems and challenges: A sms. In: 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST). IEEE, 2021. p. 135-142.

[2] DI FRANCESCO, Paolo. Architecting microservices. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017. p. 224-229.

[3] AL-DEBAGY, Omar; MARTINEK, Peter. A comparative review of microservices and monolithic architectures. In: 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI). IEEE, 2018. p. 000149-000154.

[4] MAZZARA, Manuel, et al. Microservices: Migration of a mission critical system. IEEE Transactions on Services Computing, 2018, 14.5: 1464-1477.

[5] SLYNGSTAD, Odd Petter Nord, et al. Identifying and understanding architectural risks in software evolution: An empirical study. In: Product-Focused Software Process Improvement: 9th International Conference, PROFES 2008 Monte Porzio Catone, Italy, June 23-25, 2008 Proceedings 9. Springer Berlin Heidelberg, 2008. p. 400-414.

[6] MEGARGEL, Alan; SHANKARARAMAN, Venky; WALKER, David K. Migrating from monoliths to cloud-based microservices: A banking industry example. Software engineering in the era of cloud computing, 2020, 85-108.

[7] BLINOWSKI, Grzegorz; OJDOWSKA, Anna; PRZYBYŁEK, Adam. Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access, 2022, 10: 20357-20374.

[8] KAZANAVIČIUS, Justas; MAŽEIKA, Dalius. Migrating legacy software to microservices architecture. In: 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream). IEEE, 2019. p. 1-5.

[9] SU, Ruoyu; LI, Xiaozhou; TAIBI, Davide. Back to the future: From microservice to monolith. arXiv preprint arXiv:2308.15281, 2023.

[10] SHORTT, D. Jeffrey; LARAMORE, Robert D. Practice makes performance: using a practice test to improve fe participation and pass rate. In: 31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No. 01CH37193). IEEE, 2001. p. F4A-19.

[11] SEBASTIAN, Sunil, et al. Transform monolith into microservices using docker. In: 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA). IEEE, 2017. p. 1-5.

[12] TAIBI, Davide; SYSTÄ, Kari. A decomposition and metric-based evaluation framework for microservices. In: Cloud Computing and Services Science: 9th International Conference, CLOSER 2019, Heraklion, Crete, Greece, May 2–4, 2019, Revised Selected Papers 9. Springer International Publishing, 2020. p. 133-149.

[13] TAIBI, Davide; SYSTÄ, Kari. From monolithic systems to microservices: A decomposition framework based on process mining. In: International Conference on Cloud Computing and Services Science. SciTePress, 2019. p. 153-164.

[14] FAN, Chen-Yuan; MA, Shang-Pin. Migrating monolithic mobile application to microservice architecture: An experiment report. In: 2017 ieee international conference on ai & mobile services (aims). IEEE, 2017. p. 109-112.

[15] TAIBI, Davide; LENARDUZZI, Valentina; PAHL, Claus. Architectural patterns for microservices: a systematic mapping study. In: CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress, 2018.

[16] ALSHUQAYRAN, Nuha; ALI, Nour; EVANS, Roger. A systematic mapping study in microservice architecture. In: 2016 IEEE 9th international conference on service-oriented computing and applications (SOCA). IEEE, 2016. p. 44-51.

[17] SELMADJI, Anfel, et al. From monolithic architecture style to microservice one based on a semi-automatic approach. In: 2020 IEEE International Conference on Software Architecture (ICSA). IEEE, 2020. p. 157-168.

[18] Google. Google cloud documentation. `https://cloud.google.com/docs`.

[19] vFunction. Blog. `https://www.vfunction.com/blog`.

[20] IBM. Ibm mono2micro documentation. https://www.ibm.com/cloud/mono2micro.

[21] ESKI, Sinan; BUZLUCA, Feza. An automatic extraction approach: Transition to microservices architecture from monolithic application. In: Proceedings of the 19th International Conference on Agile Software Development: Companion. 2018. p. 1-6.

[22] LI, Zhiding, et al. Microservice extraction based on knowledge graph from monolithic applications. Information and Software Technology, 2022, 150: 106992.

[23] MAZLAMI, Genc; CITO, Jürgen; LEITNER, Philipp. Extraction of microservices from monolithic software architectures. In: 2017 IEEE International Conference on Web Services (ICWS). IEEE, 2017. p. 524-53.