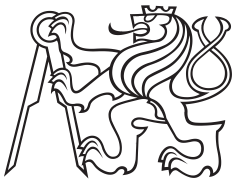
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Použití mikroslužeb v reálných aplikacích

Tomáš Palivec

Vedoucí: Ing. David Kadleček, Ph.D.
Obor: Softwarové inženýrství
Květen 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Palivec** Jméno: **Tomáš** Osobní číslo: **499249**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Použití mikroslužeb v reálných aplikacích

Název bakalářské práce anglicky:

Using micro services in real applications

Pokyny pro vypracování:

V současné době jsou mikroslužby skloňované jako nejlepší přístup pro psaní komplexních systémů a vůbec realizaci podnikových architektur. V realitě se ale ukazuje, že nesprávná implementace mikroslužeb nebo jejich použití v nevhodných situacích může být kontraproduktivní. Mezi současné trendy patří i kombinování mikroslužeb s principy reaktivních systémů a systémů řízených událostmi. Cíle této práce jsou:
1) provést rešerši používaných design patternů mikroslužeb s popisem v jaké situaci se aplikují
2) provést analýzu situací, kdy je vhodné použít mikroslužby (včetně typů) a které situace naopak vedou na jiný druh řešení
3) vytvořit knihovnu, která obsahuje vzorové implementace vybraných patternů mikroslužeb
4) využít tuto knihovnu na realistickém scénáři a provést analýzu výsledků

Seznam doporučené literatury:

1) Functional and Reactive Domain Modeling, Debasish Ghosh, October 2016 ISBN 9781617292248
2) Patterns, Principles, and Practices of Domain-Driven Design, Scott Millett, Nick Tune, May 2015, ISBN: 9781118714706
3) Building Microservices, 2nd Edition, Sam Newman, 2021, ISBN 1492034025

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. David Kadleček, Ph.D. Centrum znalostního managementu FEL

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

Datum zadání bakalářské práce: **14.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

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

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 poděkovat svému vedoucímu bakalářské práce Ing. Davidu Kadlečkovi, Ph.D. za cenné rady, připomínky a vstřícnost při zpracování této práce. Zároveň bych chtěl poděkovat své sestře Ing. Dominice Palivcové za pomoc s kontrolou práce a případné poznámky k opravám.

Prohlášení

Prohlašuji, že jsem predloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje.

V Praze, 23. května 2023

Abstrakt

Bakalářská práce se zaměřuje na mikroservisní architekturu, která je nyní populárním přístupem při implementaci komplexních aplikací. Cílem práce je shrnout výhody a nevýhody mikroservisní architektury a představit přehled návrhových vzorů spojených s mikroservisní architekturou. Na základě těchto výstupů je vytvořena příkladová knihovna obsahující implementaci návrhového vzoru CQRS v kombinaci s návrhovým vzorem materializovaný pohled. Výsledkem práce je ucelený přehled mikroservisní architektury a ukázka implementace zmíněných návrhových vzorů za pomoci synchronizace dvou databázových schémat (schéma pro zápis a schéma pro čtení) procesem CDC.

Klíčová slova: mikroservisní architektura, mikroslužby, CQRS, CDC, materializovaný pohled, Java, Kotlin, Gradle, Spring Boot, Kafka, Kafka connect, Debezium, PostgreSQL, Docker

Vedoucí: Ing. David Kadleček, Ph.D.

Abstract

Microservice architecture nowadays represents a popular approach for the implementation of complex applications. This thesis aims to summarize the advantages and disadvantages of microservice architecture and introduce an overview of design patterns associated with microservice architecture. Based on these outputs, an example library containing an implementation of the CQRS design pattern in combination with the materialized view design pattern is created. This work results in a comprehensive overview of the microservice architecture and an example implementation of the mentioned design patterns using the synchronization of two database schemas (write and read schema) by the CDC process.

Keywords: microservice architekture, microservices, CQRS, command and query resposibility segregation, CDC, change data capture, materialized view, Java, Kotlin, Gradle, Spring Boot, Kafka, Kafka connect, Debezium, PostgreSQL, Docker

Title translation: Using micro services in real applications

Obsah

Obrázky

Tabulky

Terminologie

- **HTTP** - Hypertext Transfer Protocol
- **JSON** - JavaScript Object Notation
- **Java** - objektově orientovaný programovací jazyk
- **JVM** - Java Virtual Machine (platforma pro spuštění Java programů)
- **SOA** - Service Oriented Architecture
- **ESB** - Enterprise Service Bus
- **FE** - Frontend
- **BE** - Backend
- **BFF** - Backend For Frontend (návrhový vzor)
- **API** - Application Programming Interface
- **REST API** - API založené na HTTP protokolu
- **ACID** - Atomicity (atomicita), Consistency (konzistence), Isolation (izolace), Durability (trvalost)
- **UUID** - Universal Unique Identifier
- **single point of failure** - kritická část bez které aplikace nemůže pokračovat ve správném fungování
- **stub** - falešný objekt poskytující námi definované výstupy (neovlivňuje testování)
- **mock** - falešný objekt napodobující chování skutečného objektu (může ovlivňovat testování)
- **stacktrace** - výpis aktuálního zásobníku (stack) při selhání programu
- **cluster** - skupina spolupracujících počítačů působící navenek jako celek

Kapitola 1
Úvod

1.1 Motivace

Mikroservisní architektura je aktuálně jedním z nejpopulárnějších přístupů k implementaci komplexních aplikací. Mezi oceňované vlastnosti mikroservisní architektury patří například dobrá škálovatelnost či zvýšení efektivity vývoje díky možnosti snadného rozdělení práce do nezávislých týmů. Ne vždy se však daří dosáhnout těchto požadovaných vlastností - vše záleží na vhodné volbě návrhu architektury a případné správné implementaci mikroslužeb. Z tohoto důvodu jsem se ve své bakalářské práci rozhodl vytvořit shrnutí výhod a nevýhod použití mikroservisní architektury včetně přehledu návrhových vzorů s mikroslužbami spojenými. Na základě uceleného přehledu jsem vytvořil příkladovou knihovnu obsahující vzorovou implementaci vybraných návrhových vzorů včetně testů ověřujících jejich funkčnost.

1.2 Cíl práce

Hlavní cíle této práce jsou:

1. Vytvořit rešerši používaných mikroservisních návrhových vzorů s popisem v jaké situaci dává použití smysl.
2. Vytvořit analýzu situací, ve kterých bychom se měli při vývoji vydat směrem mikroslužeb, a kdy naopak zvolit jiné řešení.
3. Vytvořit knihovnu se vzorovým vypracováním mikroservisního návrhového vzoru (CQRS s materializovaným pohledem).
4. Vytvořenou knihovnu použít na reálném scénáři a provést analýzu výsledků v podobě testů výkonu.

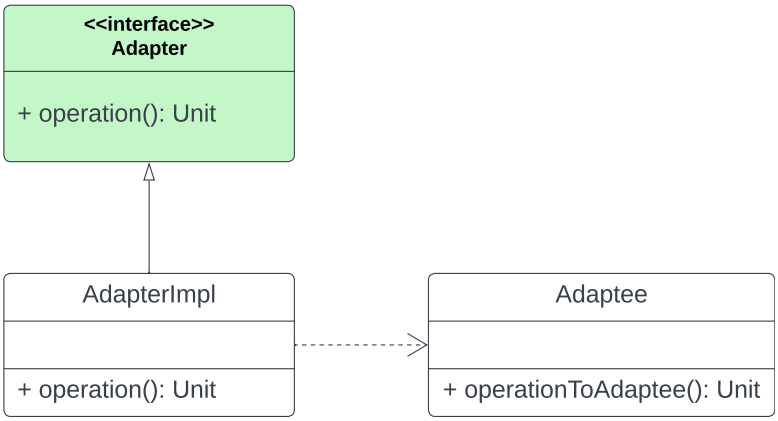
Kapitola 2
Základní pojmy

V této kapitole si představíme základní pojmy, které se v textu často objevují a je jim tedy důležité správně porozumět.

2.1 Návrhové vzory

Návrhový vzor si můžeme představit jako obecný popis či šablonu řešení pro určitý druh problému. Návrhové vzory nám pomáhají dosahovat co nejvyšší možné kvality a efektivity kódu a jsou určeny pro objektově orientované jazyky jako je například Java. Základem úspěšné aplikace návrhového vzoru je správné rozpoznání a pochopení případů, pro které se daný vzor hodí. [9]

Jako příklad si můžeme uvést návrhový vzor adaptér využívaný pro vytvoření jednotného rozhraní nad skupinou podobně se chovajících, ale odlišně implementovaných, rozhraní. Grafický popis návrhového vzoru adaptér můžeme vidět na obrázku 2.1.



Obrázek 2.1: Diagram návrhového vzoru adaptér

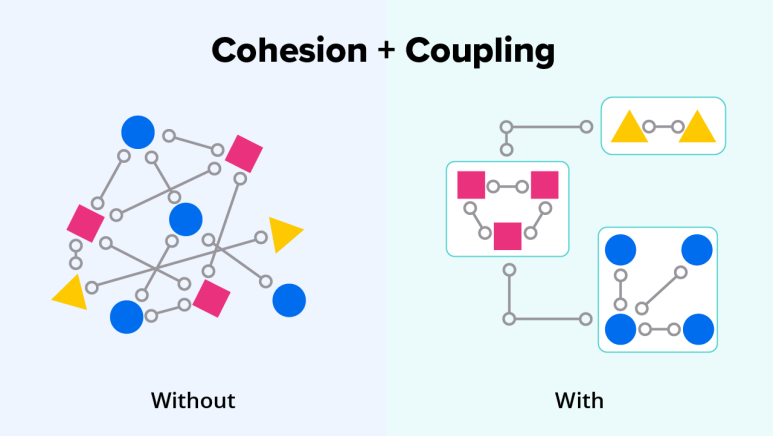
2.2 Provázanost a koherence

V anglickém překladu coupling and cohesion. Provázanost je v následujících kapitolách často zmiňovaný pojem, zejména snižování provázanosti. Prováza-

nost je vlastnost kódu či komponent, která nám říká, jak moc jsou na sobě jednotlivé komponenty závislé. Koheze nám říká, jak moc soudržný je kód v jednotlivých komponentách.

Pokud jsou komponenty silně provázané, pak je velmi obtížné je měnit, protože změna jedné komponenty může mít za následek změnu v dalších komponentách. Výsledkem snížení provázanosti je snížení nákladů za údržbu kódu a zvýšení jeho kvality. [1]

Správným přístupem je cílit na nízkou provázanost a silnou kohezi, viz. obrázek 2.2.



Obrázek 2.2: Silná provázanost a nízká koheze versus nízká provázanost a silná koheze [1]

2.3 Škálování

Při používání aplikace je žádané negativně neovlivňovat uživatele při vysoké zátěži. Negativním ovlivňováním je myšleno například dlouhé čekání při zpracování požadavku, nefunkčnost aplikace apod. Tomuto problému se snažíme předcházet škálováním. Škálování je tedy proces, kterým se aplikace přizpůsobuje různým požadavkům na výkon. Škálování může být buď horizontální nebo vertikální. Horizontální škálování znamená rozšiřování aplikace na více strojů, vertikální škálování znamená rozšiřování aplikace na stroj s většími výkonnostními parametry. [10]

2.4 Change data capture

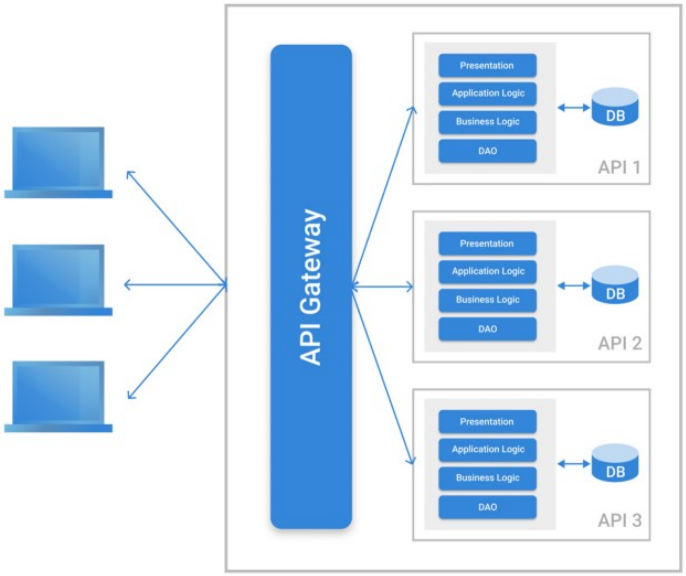
Change data capture (CDC), je proces zachytávání datových změn na úrovni databáze v reálném čase. CDC se používá pro zachycení změn v databázi a jejich následné propagace do dalších systémů. Tento přístup může být využit například pro replikaci dat do jiné databáze nebo pro zálohování. [11]

2.5 Architektura aplikace

Architektura aplikace je základním kamenem, na kterém se staví celá aplikace. Popisuje, které vzory a techniky bychom měli při vývoji aplikace použít. Jedná se o jakýsi předpis struktury aplikace sestavený z osvědčených postupů. Dodržením těchto předpisů a postupů bychom měli docílit dobře strukturované aplikace. [12]

2.5.1 Kombinace architektur

Než si ukážeme některé z architektur, je důležité si uvědomit, že například při implementaci mikroservisní architektury 2.5.6 bude velmi pravděpodobně použita v jednotlivých mikroslužbách i architektura vrstvená 2.5.4. A co více, celou takto implementovanou aplikaci můžeme označit i za klient-server architekturu, viz obrázek 2.3. Nejedná se tedy o vzájemně se vylučující pojmy, alespoň ne vždy, například kombinace klient-server a peer-to-peer architektur nedává smysl z důvodu, že klient-server architektura je založena na serveru, který poskytuje služby klientům a v peer-to-peer architektuře je každý člen sítě serverem a klientem zároveň. [13]



Obrázek 2.3: Znázornění kombinace klient-server, mikroservisní a vrstvené architektury [2]

2.5.2 Výběr architektury

Nelze jednoznačně určit pravidla, dle kterých by bylo možné vybrat nejvhodnější architekturu, protože každá architektura má své výhody a nevýhody,

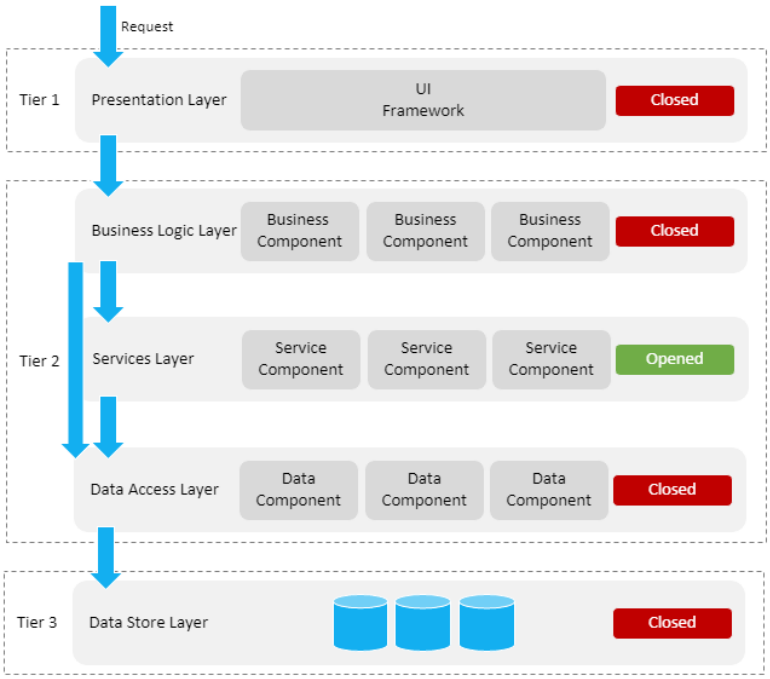
často i rozdílný rámec, jak si v následujících sekcích o jednotlivých architekturách ukážeme. Výběr architektury je tedy závislý na konkrétním problému, který řešíme a na požadavcích, které na aplikaci klademe. Zároveň je důležité vybrat správnou architekturu již na začátku vývoje aplikace, protože přechod na jinou architekturu může být velmi nákladný a časově náročný.

2.5.3 Monolitická architektura

Monolitická architektura, dle zdroje [14], obsahuje všechny komponenty aplikace v jednom balíčku. Výhodou tohoto přístupu je jednoduchost nasazení, implementace, snadné testování napříč celou aplikací a správa. Negativem jsou pak případné vyšší náklady z důvodu nutného škálování celé aplikace namísto pouhých komponent, u kterých je navýšení prostředků potřebné. Zároveň se aplikace při rozšiřování stává méně přehlednou a kvůli tomu se i její vývoj stává pomalejším.

2.5.4 Vrstvená architektura

Vrstvená architektura, anglicky layered architecture, rozděluje aplikaci na vrstvy. Každá vrstva má své specifické úkoly a komunikuje primárně pouze s vrstvou vedle sebe, kde pojem "vedle sebe" znamená vrstva přímo pod ní. Zda musí vrstva komunikovat pouze s vedlejší vrstvou se odvíjí od otevřenosti či zavřenosti daných vrstev, viz obrázek 2.4.



Obrázek 2.4: Popis významu otevřenosti a zavřenosti vrstev u vrstvené architectury [3]

Výhodou vrstvené architektury je, že je při správné implementaci snížena provázanost jednotlivých vrstev, čímž je sníženo i množství změn v aplikaci při úpravách a rozšiřování. Nevýhodou je, že je třeba vytvořit abstrakce mezi jednotlivými vrstvami, to může v případě malé aplikace vést ke zbytečně zdlouhavému vývoji. [3]

2.5.5 Servisně orientovaná architektura

Na servisně orientovanou architekturu, nebo také SOA, je nutné přihlížet s trochu širším pohledem než na ostatní zmíněné architektury. Jedná se spíše o architekturu specializující se na celý systém než na komponenty jako takové. Architektura se zaměřuje na komunikaci a kompatibilitu mezi jednotlivými komponentami, které jsou v SOA nazývány tzv. službami.

Každá služba vystavuje své standardizované rozhraní, které je nezávislé na implementaci služby samotné. Díky nezávislosti na implementaci je možné služby snadno nahrazovat novými implementacemi, aniž by o tom ostatní služby musely vědět. Zároveň je tak mezi službami zajištěna nízká provázanost.

Vystavená rozhraní jsou registrovaná v tzv. registru služeb, který je zodpovědný za jejich správu. Jedním z registrů služeb může být například ESB, který může být navíc zodpovědný za přenos zpráv mezi službami, jejich transformaci a mnoho dalšího. [15, 16]

Výhodou SOA je, že na sobě služby nejsou platformě ani technologicky závislé, dají se lehce přepoužít a škálovat. Nevýhodou tohoto přístupu je především komplexita řešení, která může vést k vyšším nákladům za údržbu. Další nevýhodou je zvýšení latence a výkonu při komunikaci mezi službami, jelikož komunikace probíhá přes síť. Zároveň použití centralizovaného servisového registru nebo ESB představuje single point of failure. [17, 18, 19, 20, 21]

2.5.6 Mikroservisní architektura

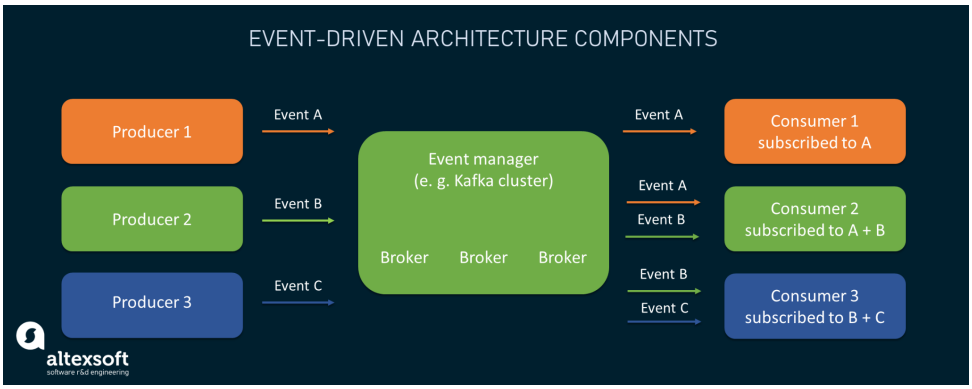
Mikroservisní architektura se na první pohled podobá servisně orientované architektuře, její rámec je však mnohem užší. Zaobírá se především rozdělením aplikace na několik menších komponent, tzv. mikroslužeb. Mikroslužby jsou spolu nízce provázané a za ideálních podmínek na sobě nezávislé. Každá z mikroslužeb je navržena pro jednu specifickou dílčí funkcionalitu, jejich seskupení pak tvoří celkovou funkcionalitu aplikace. Mikroslužby jsou navíc nezávislé na platformě a technologii. [18, 19, 20, 21]

Více si o mikroservisní architektuře řekneme v kapitole 3.

2.5.7 Událostmi řízená architektura

Událostmi řízená architektura, anglicky event driven architecture, je založená na událostech. Událost je zpráva, která je produkována v jedné z komponent a pohlcována v jiné, viz. obrázek 2.5. Komunikace probíhá asynchronně například skrze Kafku. Díky tomuto přístupu zajistíme nízkou provázanost jednotlivých aplikací. Výhodou je tedy snížená provázanost a benefity asynchronní komunikace. Nevýhodou je, že narozdíl od synchronní komunikace,

kde při neočekávané chybě dostaneme chybovou odpověď, zde musí tento mechanismus zajistit vývojář sám a potýkat se s dalšími problémy asynchronní komunikace. [4, 22]



Obrázek 2.5: Znázornění publish/subscribe mechanismu [4]

Kapitola 3
Mikroslužby

Mikroslužby nebo také mikroservisní architektura je architektonický styl popisující strukturu aplikace rozdělené na několik komponent, tzv. mikroslužeb. Každá mikroslužba je odpovědná pouze sama za sebe a má v kontextu aplikace malý rozsah byznys logiky. [5, 18, 20, 23, 24]

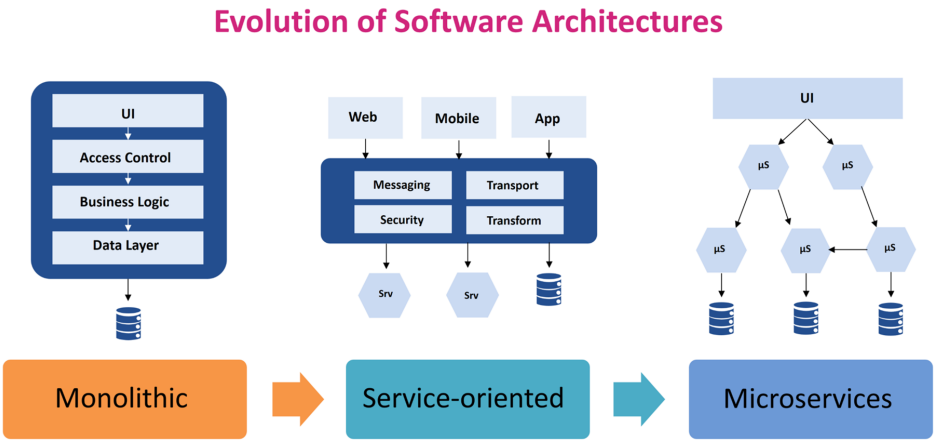
3.1 Vzestup mikroservisní architektury

Při psaní aplikací se snažíme o co nejmenší provázanost jednotlivých částí kódu například z důvodu snížení nutné komplexní znalosti aplikace i při jednoduchých úpravách. Z tohoto důvodu se aplikace začaly psát za pomocí vrstvené architektury, kde by ideálně každá vrstva měla mít svůj vlastní datový model, a tedy být co nejméně provázána se zbytkem aplikace. Takto navržená aplikace je již udržitelnější, nicméně i přes toto opatření má své limity. Stále se jedná o monolit, kde s narůstající velikostí narůstá i doba spouštění a objem zdrojů nutných pro chod. Při škálování je nutné škálovat celou aplikaci, ne pouze její části. Při povýšení na novou verzi je aplikaci nutné restartovat celou. Tyto limity mohou vadit zejména velkým společnostem, například bankám, které zaručují vysokou dostupnost svých služeb. Z těchto důvodů se začala využívat servisně orientovaná architektura, která ale neřeší problém s vysokou provázaností jednotlivých komponent, a tedy jejich závislostmi mezi sebou.

Z tohoto důvodu se při dekompozici aplikace udělal ještě jeden krok. Tímto krokem byl rozpad na ještě více dílů - vytvoření na sobě nezávislých mikroslužeb. Obrázek 3.1 ukazuje architektonickou evoluci (vrstvená architektura je zde součástí architektury monolitické).

3.2 Popularita mikroservisní architektury

Mikroservisní architektura je nyní, dle zdrojů [25, 26], populární a dle předpovědi by její popularita měla nadále stoupat.



Obrázek 3.1: Architektonická evoluce [5]

3.3 Výhody

Z předchozí části jsme si mohli všimnout následujících výhod plynoucích z použití mikroservisní architektury. [5, 20, 23, 24]

Snížení komplexity

Dekompozice velké aplikace na malé mikroslužby snižuje komplexitu a zároveň zvyšuje udržitelnost jednotlivých mikroslužeb.

Rychlost a nezávislost nasazení

Pro nasazení opravy chyby v monolitu je nutné restartovat celou aplikace. U mikroslužeb je nutné nasadit jen tu mikroslužbu, která byla opravou zasažena. To znamená, že nemusíme restartovat celou aplikaci, ale pouze její část. Díky menší velikosti je její nasazení i mnohem rychlejší. Zároveň je nám umožněno nasazovat jednotlivé mikroslužby bez ohledu na ostatních týmech. Toto je nám dovoleno zredukováním provázanosti komponent, jinak řečeno aplikací loose coupling.

Znovupoužitelnost

Jednotlivé mikroslužby jsou na sobě nezávislé, proto nám nic nebrání mikroslužby přepoužívat v jiných aplikacích.

Škálovatelnost

Při škálování monolitu je nutné škálovat všechny jeho části, ať už je používáme či ne. U mikroslužeb máme větší volnost, můžeme si vybrat jakou mikroslužbu budeme kolikrát replikovat. Tím získáváme flexibilitu a zároveň snižujeme náklady.

Autonomní vývoj

Při vývoji monolitu můžeme aplikaci spustit až ve chvíli, kdy jsou všechny její části zhotoveny. Týmy jsou velké, protože se jedná o komplexní aplikaci. Oproti tomu při vývoji mikroslužeb se můžeme setkat s malými týmy, kde každý tým vyvíjí jednu mikroslužbu. Tím získáváme autonomii vývoje, kterou můžeme výrazně zrychlit vývoj.

Technologie

Již několikrát bylo zmíněno, že jednotlivé mikroslužby jsou na sobě nezávislé. To znamená, že i použité technologie a platformy u jednotlivých mikroslužeb mohou být zcela nezávislé. Další výhodou v rámci technologií je například povýšení verze samotného programovacího jazyka či frameworku. Není nutné přepisovat komplexní monolit, namísto toho nám stačí postupně upgradovat jednotlivé mikroslužby.

Dostupnost

U monolitické aplikace riskujeme při neočekávané chybě pád celé aplikace. U mikroslužby přijdeme pouze o část aplikace, zbytek stále běží, ač ne všechny funkce. Navíc je možné použít několik replik mikroslužby, které se budou automaticky přepínat, pokud dojde k pádu jedné z nich. (Toto chování není vhodné cíleně využívat, chyby by měly být opravovány a ne ignorovány.)

3.4 Nevýhody

Použití mikroservisní architektury však nepřináší pouze výhody. Jednou velkou nevýhodou je fakt, že rozdělením monolitu na mikroslužby vytváříme vlastně distribuovaný systém. Vývojáři se tak musí naučit řešit problémy, které se v distribuovaných systémech vyskytují. Zároveň narůstají nároky na složitost a komplexnost infrastruktury. [5, 20, 23, 24]

Komunikace

V monoliticky navržené aplikaci voláme pro přístup do jiné komponenty funkce či metody, a tedy komunikaci za nás řeší jazyk sám. Při rozdělení aplikace na více mikroslužeb musíme komunikaci řešit za pomoci různých front, REST API apod.

Transakce

Při přístupu do databáze používáme transakce, abychom provedli buď celý zápis nebo při chybě doposud změněná data vrátili do původního stavu (zajistili ACID vlastnosti). Při rozdělení aplikace na více mikroslužeb musí tyto mechanismy napříč aplikací řešit vývojáři.

Závislosti při nasazení

Za ideálních podmínek bychom pro chod mikroslužby neměli být závislí na jiné mikroslužbě. To je ovšem v realitě prakticky nesplnitelné. V těchto případech je nutné dbát na pořadí nasazení jednotlivých mikroslužeb.

Testování celé aplikace

Rozdělením aplikace na jednotlivé mikroslužby zjednodušíme testování samotných mikroslužeb, zároveň se pro nás však stane obtížné testování napříč aplikací. Pro běh takových testů je nutné nasadit i závislé mikroslužby nebo mikroslužby nahrazovat mocky a stuby. Nasazení závislých mikroslužeb nebo přímo celé aplikace se může zdát jako jednoduché východisko, problém ovšem přijde při neúspěšném testu a zjištění, že pro nalezení chyby nám nestačí pouze stacktrace vypsaný v konzoli. V takovém případě musíme zjistit, která mikroslužba způsobila problém a prozkoumat, co se v ní odehrálo. Výsledkem takového testu je tedy značně komplexní a složitý stacktrace, který je obtížně zpracovatelný. Dalším problém při testování je obtížné restatování aplikace za účelem zachování konzistentního stavu.

Zdroje pro chod

Tento problém můžeme potkat například při volbě programovacího jazyka Java. (Java pro spuštění programu potřebuje platformu JVM.) Jelikož mikroslužby běží zvlášť, platí, že máme spuštěných právě tolik JVM jako je spuštěných mikroslužeb. Zároveň však můžeme hovořit i o zdrojích použitých pro virtualizaci, nebo statických datech nutných pro běh aplikace.

3.5 Kdy mikroslužby použít a kdy se naopak použití vyvarovat

V této sekci si ukážeme dva reálné případy použití mikroslužeb, kde jeden případ je úspěšný a druhý naopak neúspěšný.

3.5.1 Úspěšný případ

Úspěšným případem implementace mikroslužeb je například dekompozice monolitní aplikace na mikroslužby u jedné české anonymní společnosti. Tato společnost se potýkala s následujícími problémy:

- **Byznys logika na FE** - Značné množství byznys logiky je implementováno na FE.
- **Zakázkové frameworky** - Obsahuje na míru vytvořené frameworky pro mapování objektů.
- **Staré technologie** - Obsahuje staré technologie, které již nejsou podporovány.

- **Servery** - Dva funkčně se překrývající servery pracující nad jednou databází. (První server sloužil pro správu dat a integraci s ESB, druhý server měl původně sloužit pro implementaci BFF logiky, postupem času ale došlo k degradaci záměru tohoto serveru a k duplicitě funkcionalit u těchto serverů.)
- **Nasazení** - Aplikaci je nutné nasadit jako celek (bez možnosti snadné dekompozice).

Výsledným návrhem bylo kompletní přepsaní FE do nové technologie, přesunutí byznys logiky z FE na BE, separace zakázkových frameworků a dekompozice aplikace dle domén. Dekomponované mikroslužby jsou dvou typů:

- **Dekompozice A** - Plná dekompozice, kdy každá mikroslužba má svou vlastní databázi. Výhodou je plná izolace dopadu změny v jedné mikroslužbě na ostatní, možnost vyšší paralelizace vývoje i testování. Nevýhodou je nutnost řešit konzistenci dat napříč mikroslužbami, nutnost zásahu do modelu původní databáze a všech částí aplikace, které s touto částí modelu pracují.
- **Dekompozice B** - Mikroslužby tohoto typu přistupují do sdílené databáze. Mikroslužba má oprávnění zapisovat do omezené části databázového modelu, ale oprávnění číst má pro celý databázový model. Výhodou tohoto přístupu je jednoduchost dekompozice z důvodu absence nutnosti zásahu do modelu původní databáze a absence nutnosti řešit konzistenci dat napříč mikroslužbami. Nevýhodou je menší izolace dopadu změny oproti předchozímu případu dekompozice a nutnost vyšší kontroly domén mikroslužeb z důvodu pouhého logického oddělení.

Dekompozice A byla aplikována pro domény, které mohly být snadno plně osamostatněny. Dekompozice B byla aplikována pro domény, které nebylo možné snadno osamostatnit z důvodu nutného komplexního zásahu do databázového modelu (zejména domény pracující s rozsáhlými datovými strukturami).

Tímto způsobem bylo dosaženo řešení zmíněných problémů a zároveň bylo dosaženo výhod mikroslužeb.

3.5.2 Neúspěšný případ

Neúspěšným případem implementace mikroservisní architektury je aplikace od společnosti Amazon Prime Video. Úkolem aplikace je monitorování a nahlašování kvalitativních problémů streamu médií.

Původním řešením byla mikroservisní architektura obsahující tři hlavní komponenty:

- **Konvertor** - Komponenta konvertující stream médií do snímků videa nebo do zvukových vzorků.

- **Detektor** - Komponenta detekující defekty v konvertovaných snímcích videa nebo zvukových vzorcích.
- **Orchestrator** - Komponenta řídící předchozí dvě komponenty.

Řešení bylo zvoleno z důvodu rychlého vývoje a nasazení pomocí cloudových služeb společnosti Amazon. Toto řešení se ale ukázalo jako nevhodné z následujících důvodů:

- **Škálování** - Tvrdý limit škálování dosahoval pouze 5% oproti očekávání.
- **Náklady** - Provozování orchestrace bylo velmi nákladné.
- **Přenos dat** - Komponenty mezi sebou přenášely velké množství dat pomocí úložiště S3[1].

Na základě těchto důvodů bylo rozhodnuto přepsat aplikaci do monolitické architektury. Tímto způsobem byla vyřešena orchestrace i přenos dat mezi komponentami. Myšlenka horizontálního škálování pomocí mikroslužeb byla nahrazena vertikálním škálováním a tvořením kopií aplikace. Každá kopie aplikace obsahovala, dle parametrického nastavení, jiné detektory.

Tímto způsobem bylo dosaženo snížení nákladů za provoz aplikace o 90%. Více informací o tomto případu je možné nalézt ve zdroji [27].

[1] Amazon Simple Storage Service

Kapitola 4

Návrhové vzory specifické pro mikroslužby

V této kapitole si představíme návrhové vzory specifické pro mikroservisní architekturu a řekneme si, v jakých situacích je vhodné jejich použití.

Narozdíl od obecných návrhových vzorů nemají návrhové vzory specifické pro mikroslužby žádné oficiální kategorizování. Přesto je však lze rozdělit na základě principů, které svým popisem řeší, do následujících kategorií. [28]

4.1 Dekompoziční návrhové vzory

Při přechodu z monolitu na mikroservisní architekturu máme mnoho možností, jakými můžeme dělbu provést. Tato kategorie se těmito možnostmi zabývá. Neznamená to ovšem, že vzory využijeme pouze při dekompozici monolitu. I při návrhu aplikace ze zelené louky nám mohou pomoci aplikaci správně dekomponovat.

4.1.1 Strangler

Návrhový vzor strangler se řadí do tzv. blownfield vývoje (více o brownfield vývoji [29]) a je ho vhodné využít v případě, kdy máme monolitickou aplikaci a chceme ji převést na mikroservisní architekturou. Jedná se o postup, kdy postavíme dvě aplikace vedle sebe, postupně implementujeme jednotlivé mikroslužby a následně přesouváme provoz z monolitu do mikroslužeb. Při dokončení implementace můžeme monolit zcela vypnout. [28]

4.1.2 Anti-corruption layer

Návrhový vzor anti-corruption layer navazuje na předchozí návrhový vzor strangler. Jeho využití je příhodné v situaci, kdy monolitická aplikace neobsahuje technologie a nástroje, které chceme využít v mikroslužbách a zároveň nechceme do mikroslužeb zanášet staré datové struktury, technologie, API apod. V takovém případě je vhodné vytvořit vrstvu mezi monolitem a mikroslužbami, která bude data z monolitu převádět do formátu, který bude mikroslužbám vyhovovat. [30]

4.1.3 Bulkhead

Bulkhead návrhový vzor se zabývá izolací jednotlivých mikroslužeb. Jedná se o přístup, kdy každá mikroslužba běží na vlastním fyzickém stroji nebo například v logicky odděleném kontejneru. V případě, že dojde k pádu jedné mikroslužby, ostatní mikroslužby by neměly být zasaženy. Tento přístup je vhodný zejména v případě, že je jedna z mikroslužeb výkonnostně náročná nebo se v ní nachází funkcionalita náchylná na pád. [28]

4.1.4 Sidecar

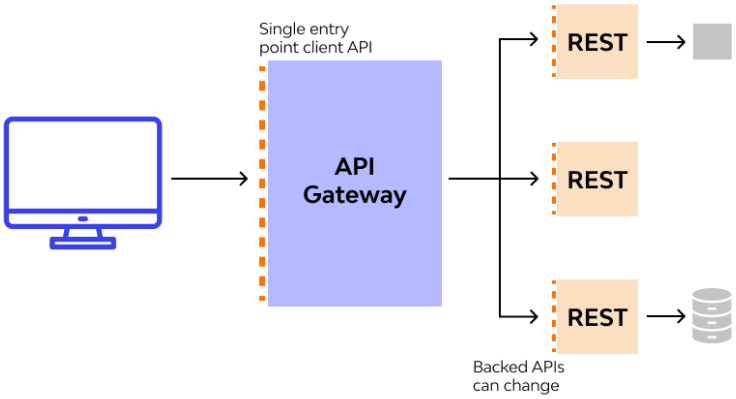
Návrhový vzor sidecar může být použit pro oddělení hlavních funkcí mikroslužby od vedlejších funkcí. Jedná se o přístup, kdy vedle hlavní mikroslužby běží další mikroslužba, která zajišťuje například logování, monitorování, šifrování komunikace apod. [28]

4.2 Integrační návrhové vzory

Integrace nebo také ucelení, sjednocení, ale i začlenění. Jedná se o návrhové vzory zajišťující například jednotný bod přístupu do distribuované aplikace. Rovněž v této kategorii můžeme nalézt vzory pro komunikaci mezi jednotlivými mikroslužbami.

4.2.1 API Gateway

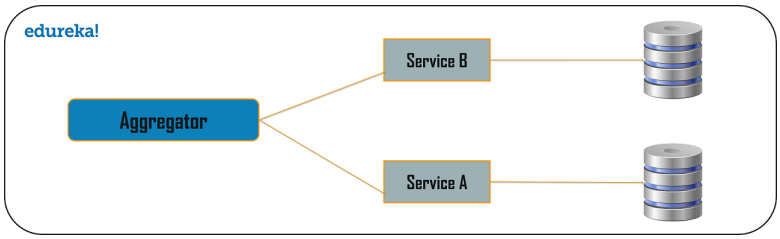
Jedná se o návrhový vzor popisující mikroslužbu, která zajišťuje přístup k jednotlivým mikroslužbám za pomoci jednotného rozhraní, viz. obrázek 4.1. Zároveň může zajišťovat i další funkce, jako například autentizace, autorizace, logování, správu HTTP hlaviček požadavků, caching, rozkládání zátěže (load balancing) apod. [6, 7]



Obrázek 4.1: Příklad API Gateway [6]

4.2.2 Agregace

Tento návrhový vzor využijeme především v případě, že pro získání dat je třeba více než jedna mikroslužba nebo obecně více zdrojů. Náplní agregátoru je pak získat data z jednotlivých zdrojů a jejich sjednocení nebo dokonce transformování do požadovaného formátu. Nejzákladnější příklad je možné vidět na obrázku 4.2. [7]



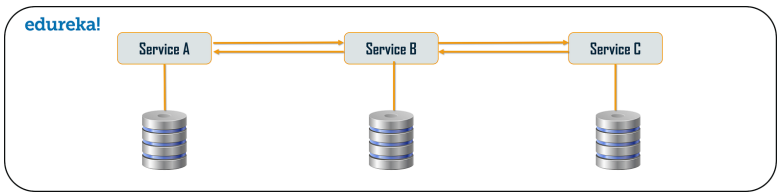
Obrázek 4.2: Příklad agregátoru [7]

Agregaci dat můžeme provádět několika dalšími způsoby, které jsou popsány v následujících podkapitolách.

Zřetězení mikroslužeb

Jedná se o případ, kdy jednotlivé mikroslužby jsou za sebou zřetězené, a kde výstup jedné mikroslužby je vstupem pro další. Výhodou je jednoduchost implementace, nevýhodou je neefektivita řešení. Jedná se o přístup, který nelze paralelizovat a výsledný výkon je omezen výkonem nejpomalejší mikroslužby. Zároveň pokud jedna z mikroslužeb selže, selže celý proces a není tedy možné získat ani částečný výsledek. [7, 28]

Ilustraci tohoto přístupu lze vidět na obrázku 4.3.



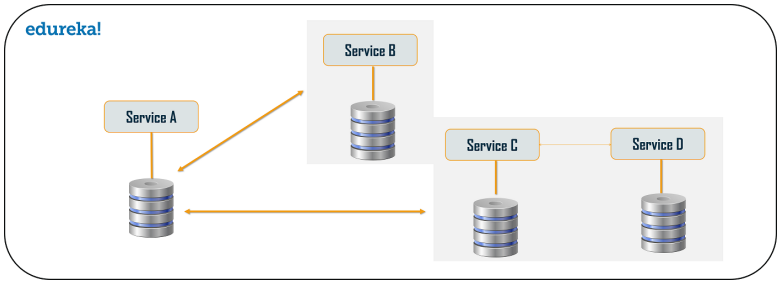
Obrázek 4.3: Příklad zřetězení mikroslužeb [7]

Branch

Branch návrchový vzor říká, že agregaci můžeme provést pomocí volání více zřetězených mikroslužeb, včetně řetězů, které jsou tvořeny pouze jednou mikroslužbou, viz. obrázek 4.4. [7]

4.2.3 Orchestrace

Návrhový vzor orchestrace se na první pohled může podobat agregaci, narozdíl od agregátoru ale nemanipuluje s daty. Orchestrace pouze řídí tok dat mezi



Obrázek 4.4: Příklad návrhového vzoru branch [7]

jednotlivými mikroslužbami. Tento přístup je vhodný zejména v situacích, kdy chceme zamezit komunikaci mezi jednotlivými mikroslužbami. Namísto sdílení API mezi mikroslužbami, které by mohlo vést k nežádoucí komunikaci, nebo v případě aktualizace API vést k aktualizizaci i všech zasažených mikroslužeb, je orchestrátor jediným bodem, který s mikroslužbami komunikuje. [31]

4.3 Datové návrhové vzory

Tato kategorie sepisuje návrhové vzory předepisující jakým způsobem zpracovávat a ukládat data. Například zda má každá mikroslužba disponovat svým vlastním datovým úložištěm nebo ho sdílet.

4.3.1 Databáze pro každou mikroslužbu

U mikroservisní architektury cílíme na nízkou provázanost, aby každá mikroslužba byla ideálně plně nezávislá na ostatních. V takovém případě by každá mikroslužba měla disponovat vlastním datovým úložištěm. Mezi další důvody, proč bychom mohli chtít každé mikroslužbě přidělit vlastní databázi je například nekonzistence dat, rozložení databázové zátěže či technologická nezávislost. [7]

4.3.2 Sdílená databáze

Sdílené databázové prostředky nejsou u mikroservisní architektury často se vyskytující, jedná se spíše o anti-pattern[1]. Kde nebo spíše kdy můžeme tento návrhový vzor dočasně vyžadovat je například při přechodu z monolitické architekrury. [7]

4.3.3 Materializovaný pohled

Materializovaný pohled je tvořen kopií originálních dat, kde jejich struktura je více vhodná pro časté čtení z důvodu, že čtecí operace jsou mnohem častější, jak operace pro zápis. Například v případě, že máme databázi obsahující informace o uživatelích, můžeme vytvořit materializovaný pohled, který bude

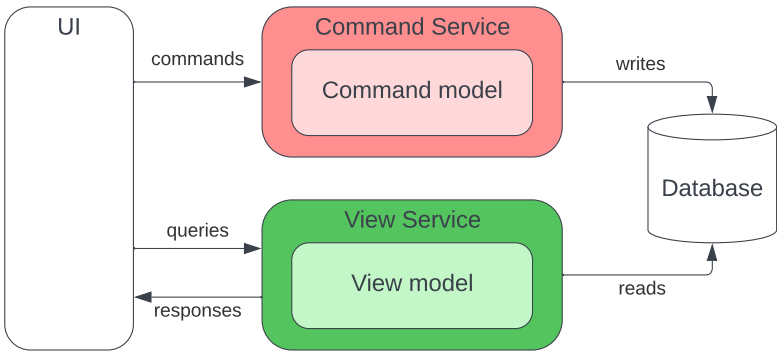
[1] návrhový vzor představující nevhodnou implementaci

obsahovat pouze informace o uživatelích, kteří jsou aktivní. Tímto způsobem můžeme urychlit čtení dat, protože již není třeba prohledávat skrze celou databázi, ale pouze její podmnožinu. Navíc, při tvorbě materializovaného pohledu se nemusíme omezovat pouze na jednu tabulku, ale můžeme při změně dat jednorázově předpřipravit drahé sloučení dvou tabulek namísto opětovného slučování při každém čtení. [32]

4.3.4 Command and query responsibility segregation

Command and query responsibility segregation (CQRS) je návrhový vzor, který můžeme ocenit ve velkých aplikacích. Jeho cílem je oddělit operace pro čtení a zápis. Operace CREATE, UPDATE a DELETE reprezentují v tomto návrhovém vzoru tzv. commands a operace READ představuje tzv. queries, viz. obrázek 4.5. Tímto rozdělením můžeme docílit zaprvé rozdělení service vrstvy na dvě části, kde každá může být spravovaná nezávisle na sobě a zadruhé, queries můžeme provádět efektivněji skrze materializovaný pohled [7, 28].

Nevýhodou je zvýšená složitost implementace, v případě použití materializovaného pohledu především synchronizace hlavní databáze s vedlejší.



Obrázek 4.5: Příklad návrhového vzoru command and query responsibility segregation

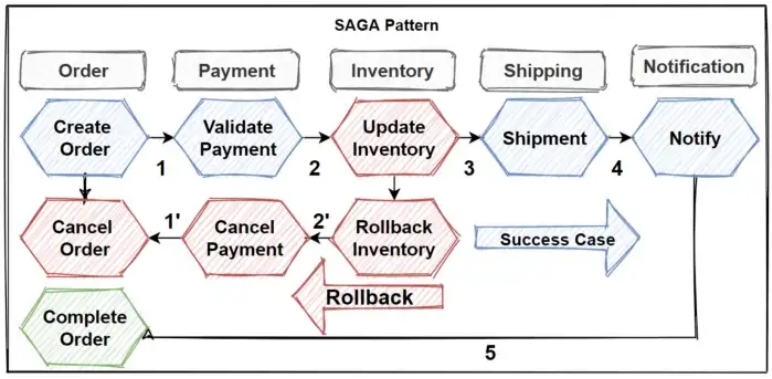
4.3.5 Event sourcing

Návrhový vzor event sourcing si můžeme představit jako frontu stavů aplikace, kde každý stav aplikace je zaznamenán pomocí události (eventu). Jednotlivé události jsou ukládány do tzv. event store a následně vykonávány v chronologickém pořadí. Výhodou tohoto přístupu je možnost vrácení aplikace do jakéhokoli stavu v minulosti pouhou inverzí pořadí událostí a provedení jejich inverzní operace. Výhodou je také možnost sledování změn v aplikaci, které mohou být použity například pro audit. [7, 28]

4.3.6 Sága

Návrhový vzor sága řeší problém transakcí v distribuovaném sytému. Jejím principem je rozdělit transakci na několik malých, které již mohou být prová-

děny samostatně vyhovujícími mikroslužbami, viz. obrázek 4.6. Existují dvě varianty ságy, které se liší způsobem řízení:

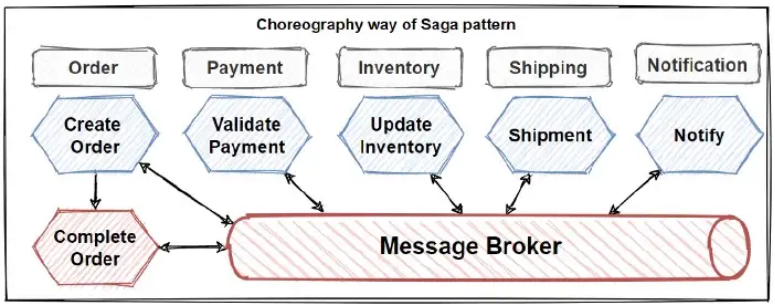


Obrázek 4.6: Princip návrhového vzoru sága [8]

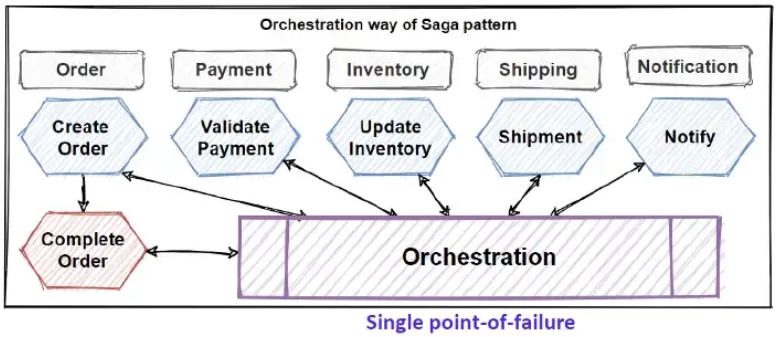
1. **Choerografikcá sága** znázorněna na obrázku 4.7 je založena na komunikaci pomocí událostí. Při úspěšném dokončení lokální transakce mikroslužbou je zaslána událost, kterou je informována další mikroslužba, která opět zpracuje transakci. Takto proces pokračuje dál až do plného zpracování hlavní transakce. Pokud při vykonávání transakce nastane chyba, je vyvolán rollback, který postupně vykoná každá ze zúčastněných mirkoslužeb.

2. **Orchestrační sága** znázorněna na obrázku 4.8 funguje na stejném principu jako sága choreografická, s tím rozdílem, že jednotlivé lokální transakce jsou vykonávány sekvenčně a jsou řízeny orchestrační komponentou, která zároveň vyobrazuje single point of failure.

Po celou dobu vykonávání transakce je aplikace ve stavu eventuální konzistence, kdy ačkoli lokální transakce v jedné dané mikroslužbě již v pořádku proběhla, celý proces dokončen být nemusí. [8, 28]



Obrázek 4.7: Obrázek návrhového vzoru sága ve variantě choerografie [8]



Obrázek 4.8: Obrázek návrhového vzoru sága ve variantě orchestrace [8]

4.4 Cross-cutting concern návrhové vzory

V první řadě je nutné porozumět pojmu cross-cutting concern. Jedná se o problém, který je nutné řešit napříč celou aplikací, ale zároveň není součástí její business logiky. Jedná se například o logování, autentizaci, autorizaci, caching apod.

4.4.1 Monitorování

V této sekci nebudeme hovořit o jediném návrhovém vzoru, ale celé skupině. V mikroservisní architektuře je nutné sledovat jednotlivé mikroslužby, ale také celou aplikaci jako celek. K tomu slouží následující návrhové vzory:

- **Agregace logů** - agregace logů z jednotlivých mikroslužeb do jednoho centrálního systému
- **Výkonnostní metriky** - získávání výkonnostních metrik z jednotlivých mikroslužeb
- **Distribuované trasování** - sledování průchodu požadavku v rámci aplikace přidáním jedinečného identifikátoru do hlavičky HTTP požadavku
- **Health check** - zjišťování stavu/dostupnosti jednotlivých mikroslužeb

4.4.2 Externí konfigurace

Každá mikroslužba potřebuje pro svůj chod konfiguraci, nehledě na to, zda se jedná o konfiguraci databáze nebo logovací severity. Navíc je často nutné spravovat větší množství takových konfigurací z důvodu samostatných profilů pro produkci, testování, vývoj apod. Spravovat takové konfigurace v rámci celé aplikace může být náročné z důvodu nutného zásahu do každé mikroslužby zvlášť. Řešením tohoto problému je externí konfigurace, kterou si aplikace sama načte během startu nebo aktualizuje při změně. [28]

4.4.3 Service discovery

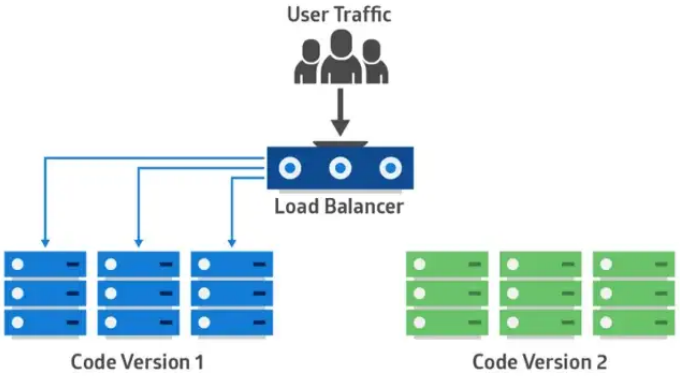
Pro komunikaci mezi mikroslužbami potřebují jednotlivé mikroslužby znát IP/URL adresy ostatních mikroslužeb. Tento problém řeší návrhový vzor service discovery, který jednotlivé mikroslužby registruje. Mikroslužbě pak stačí znát pouze adresu daného discovery serveru a například název mikroslužby, se kterou chce komunikovat. Service discovery je zároveň základem pro implementaci load balancingu. [28]

4.4.4 Circuit breaker

Při vysokém zatížení některé mikroslužby může nastat její pád. Pokračovat v zasílání požadavků takové mikroslužbě by mohlo vést k zanášení komunikačního kanálu nebo i pádu celé aplikace. Pomocí návrhového vzoru circuit breaker můžeme regulovat délku časového limitu (timeout) požadavku nebo i počet požadavků, které může daná mikroslužba vykonávat. Pokud je daný limit překročen, circuit breaker se přepne do tzv. open stavu. Circuit breaker také umožňuje definovat, jak dlouho bude mikroslužba v open stavu, než se přepne do tzv. half-open stavu, kdy je možné opět zasílat požadavky. Pokud je mikroslužba v half-open stavu a požadavky proběhnou úspěšně, circuit breaker se přepne do tzv. closed stavu, a tedy obnoví svoji plnou funkčnost. Pokud se však požadavky nezdaří, circuit breaker se znovu přepne do open stavu. [7, 28]

4.4.5 Blue-green deployment

Při rozsáhlé aktualizaci aplikace může být nutné aplikaci restartovat. V případě, že bychom aplikaci vypli a poté nastartovali její novou verzi, by byla aplikace po celou dobu procesu nedostupná. Řešením tohoto problému je návrhový vzor blue-green deployment, který umožňuje provádět aktualizace aplikace bez jejího restartu, nebo spíše není restart zaznamenatelný. V rámci blue-green deploymentu je aplikace rozdělena do dvou částí, tzv. blue a green, viz. obrázek 4.9. Vedle běžící blue aplikace se vybuduje aktualizovaná a plně funkční green aplikace. Následně se přepne provoz z blue na green aplikaci. Po dokončení přepnutí se stará blue aplikace smaže. V případě, že by se při přepnutí na green aplikaci objevily problémy, je možné přepnout zpět na blue aplikaci. [28]



Obrázek 4.9: Popis návrhového vzoru blue-green deployment [7]

Kapitola 5

Návrh příkladové knihovny

V této kapitole si odůvodníme výběr návrhových vzorů, které jsou použity v příkladové knihovně. Zároveň u těchto návrhových vzorů zmíníme problematiku aplikace v praxi a popíšeme jejich možná řešení. Následně představíme návrh příkladové knihovny, ve kterém klíčové části aplikace popíšeme a vysvětlíme jejich význam.

5.1 Výběr návrhového vzoru

Jako příkladně navržený a následně implementovaný návrhový vzor mikroservisní architektury je zvolen command and query responsibility segregation (CQRS), viz. sekce 4.3.4, v kombinaci s materializovaným pohledem, viz. sekce 4.3.3. Tato kombinace návrhových vzorů je zvolena na základě doporučení od mého vědoucího práce Ing. Davida Kadlečka, Ph.D., a také na základě jejich problematické aplikaci v praxi.

5.1.1 Problematika aplikace návrhových vzorů

Při aplikaci návrhového vzoru CQRS a materializovaného pohledu narazíme na problém se synchronizací vedlejší databáze (obsahující transformovaná data pro čtení) s hlavní databází (obsahující skutečná data - single source of truth). Tento problém je možné řešit následujícími způsoby.

Synchronizace pomocí aplikačních událostí

Jedná se o přístup, který spravuje synchronizaci na úrovni mikroslužby. Vyžaduje, aby mikroslužba aplikující změny v hlavní databázi vyvolávala události, které jsou následně zachyceny a zpracovány mikroslužbou synchronizující vedlejší databázi.

Problémem tohoto přístupu je vysoká náročnost na vývojáře, který musí zajistit:

1. **Vyvolání událostí** - Aby byly na všechny změny vyvolány události.

2. **Správné pořadí událostí** - Aby byly změny vyvolány v pořadí, ve kterém je lze zpracovat (např. vytvoření košíku před přidáním produktu do košíku).
3. **Správné zacházení s transakcemi** - Aby byly události vyvolány až po úspěšném uložení dat do hlavní databáze.

Zajištění takového chování vede k nutné znalosti celé aplikace a jejího chování. Vývojář musí znát všechny možné scénáře, které mohou nastat a zajistit jejich správné zpracování. Navíc tímto přístupem zatěžujeme i mikroslužbu samotnou. Mikroslužba musí vyvolávat události, které nejsou pro její běh potřebné. Tím se zvyšuje náročnost na výkon, ale i komplexita mikroslužby.

Synchronizace pomocí databázových událostí

Druhým přístupem je synchronizace databázovými událostmi za pomoci procesu CDC, viz. sekce 2.4. Tento přístup dokáže plně odstínit vývojáře od synchronizace vedlejší databáze. Nevýhodou tohoto řešení je nutnost použití databáze, která podporuje CDC. Zároveň je řešení náročnější na správu při migraci databázového schéma a konfiguraci.

V mém případě je zvolen tento způsob. Důvodem je snaha o co největší odstínění vývojáře od synchronizace vedlejší databáze, a o co nejmenší náročnost na výkon a komplexitu mikroslužby.

5.1.2 Kontext příkladové knihovny

Příkladová knihovna je zasazena do kontextu e-shopu. Jedná se o velmi jednoduchý model, který obsahuje pouze základní entity, které jsou potřebné pro vysvětlení návrhu a implementace.

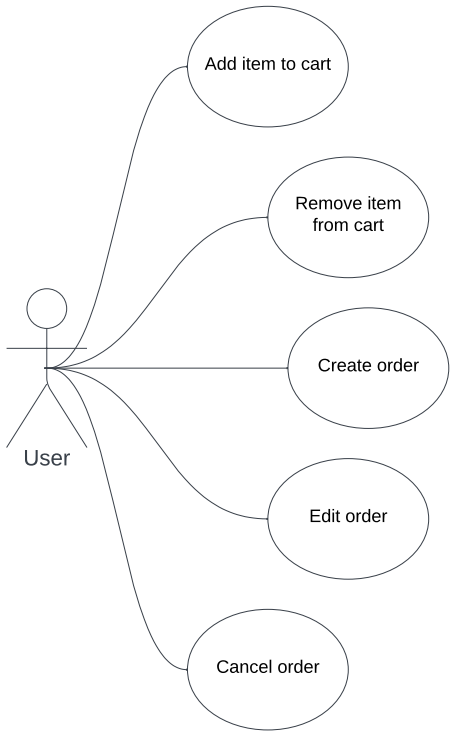
5.2 Zabezpečení

Zabezpečení není cílem příkladové knihovny, proto je zvoleno jednoduché řešení pomocí UUID přenášeném v hlavičce HTTP požadavku. Jediným důvodem tohoto řešení je rozpoznání uživatele, který vytvořil požadavek. V případě, že by bylo nutné zabezpečit přístup k datům, bylo by třeba zvolit jiné řešení.

5.3 Případy užití

Aplikace obsahuje pouze jednoho aktéra (Uživatele) jehož případy užití jsou rozdělené do dvou diagramů. První diagram 5.1 obsahuje případy užití, které jsou zodpovědné za zápis dat do databáze (commands).

- **Add product to cart/přidat produkt do košíku** - Uživatel může přidat produkt do košíku.

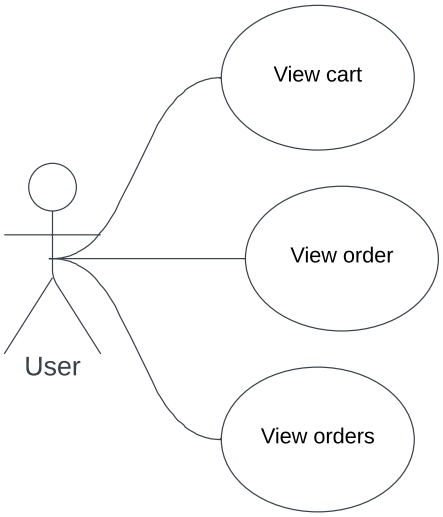


Obrázek 5.1: Diagram případů užití Uživatele (User) ovlivňující data

- **Remove product from cart/odebrat produkt z košíku** - Uživatel může odebrat produkt z košíku.
- **Create order/vytvořit objednávku** - Uživatel může vytvořit objednávku.
- **Edit order/upravit objednávku** - Uživatel může upravit objednávku.
- **Cancel order/zrušit objednávku** - Uživatel může zrušit objednávku.

Druhý diagram 5.2 obsahuje případy užití, které jsou zodpovědné za získání dat z databáze (queries).

- **View cart/zobrazit košík** - Uživatel si může zobrazit obsah svého košíku.
- **View order/zobrazit objednávku** - Uživatel si může zobrazit svoji objednávku.
- **View orders/zobrazit objednávky** - Uživatel si může zobrazit své objednávky.



Obrázek 5.2: Diagram případů užití Uživatele (User) neovlivňující data

5.4 Diagramy tříd

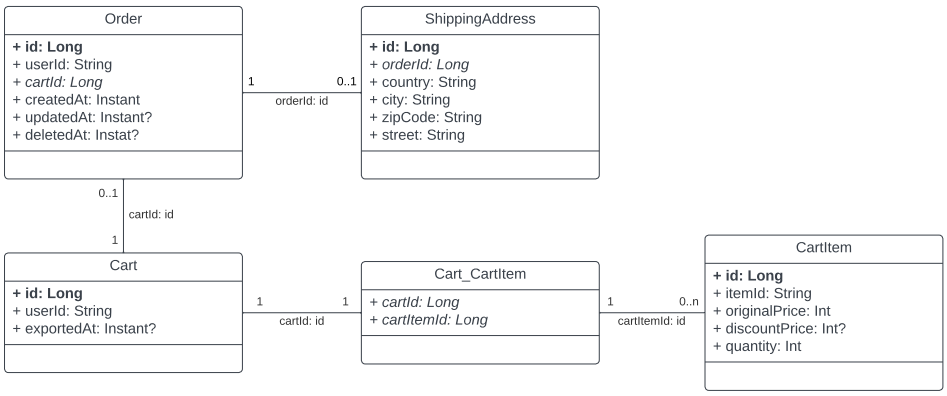
Aplikace disponuje dvěma diagramy tříd, které mají vyšší míru popisu a soustředěnosti na databázi, než je u návrhu obvyklé. Důvodem je zvýraznění vztahů mezi jednotlivými třídami za účelem snazšího pochopení problematik návrhu materializovaného pohledu.

První diagram 5.3 reprezentuje byznys třídy, které jsou použity v hlavní databázi (commands).

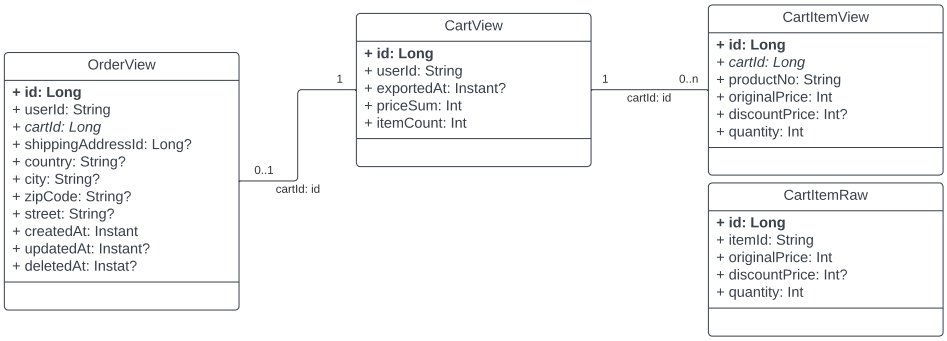
- **CartItem/položka košíku** - Reprezentuje produkt v košíku.
- **Cart/košík** - Reprezentuje košík, do kterého lze vkládat položky.
- **Cart_CartItem** - Reprezentuje uměle vytvořenou vazební tabulku mezi košíkem a položkou košíku. Její význam bude vysvětlen v sekci 5.5.2.
- **Order/objednávka** - Reprezentuje objednávku.
- **ShippingAddress/adresa doručení** - Reprezentuje adresu doručení objednávky (v případě, že se liší od adresy uživatele).

Druhý diagram 5.4 reprezentuje třídy, které ve vedlejší databází tvoří materializovaný pohled nad daty z hlavní databáze.

- **CartItemView/pohled položky košíku** - Reprezentuje třídu materializovaného pohledu, která je tvořena agregací byznys tříd "CartItem" a "Cart_CartItem".



Obrázek 5.3: Diagram tříd byznys modelu

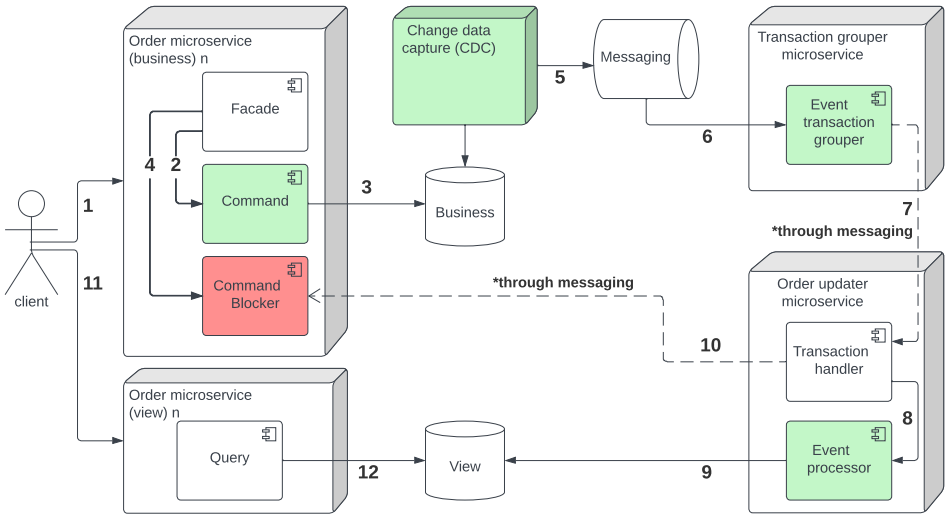


Obrázek 5.4: Diagram tříd materializovaného pohledu

- **CartView/košík** - Reprezentuje třídu materializovaného pohledu, která obsahuje byznys třídu "Cart" obohacena o počet položek v košíku a jejich celkovou cenu.
- **CartItemRaw** - Reprezentuje třídu materializovaného pohledu, která je kopií byznys třídy "CartItem". Její význam bude vysvětlen v sekci 5.5.2.
- **OrderView** - Reprezentuje třídu materializovaného pohledu, která je tvořena agregací byznys tříd "Order" a "ShippingAddress".

5.5 Mechanismus synchronizace

Mechanismus synchronizace je nejdůležitější částí návrhu. Jeho úkolem je zajistit, aby data v materializovaném pohledu byla vždy aktuální. Na následujícím obrázku 5.5 je naznačen průběh synchronizace vedlejší databáze na základě provedení požadavku uživatelem. Body 1-10 reprezentují zpracování požadavku typu command. Body 11-12 reprezentují zpracování požadaveku typu query.



Obrázek 5.5: Popis průběhu zpracování požadavků a synchronizace vedlejší databáze

1. Uživatel provede požadavek typu command (např. přidání produktu do košíku).

2. Komponenta “Facade” aktivuje transakci a deleguje požadavek do komponenty “Command”.

3. Komponenta “Command” požadavek zpracuje. Při návratu do komponenty “Facade” je na pozadí proveden proces získání a uložení unikátního identifikátoru transakce, transakce je následně potvrzena (commit).

4. Komponenta “Facade” předá unikátní identifikátor transakce komponentě “Command Blocker”, která zablokuje vlákno zpracovávající požadavek po dobu procesu synchronizace nebo vypršení časového limitu (timeout).

5. Proces CDC zaznamená změny v hlavní databázi a pro každou vytvoří událost. Události jsou následně zaslány do systému zpráv.

6. Komponenta “Event Transaction Grouper” shromáždí všechny události, které byly vytvořeny v rámci jedné transakce.

7. Komponenta “Event Transaction Grouper” vytvoří novou událost obsahující shromážděné události a následně je zašle do systému zpráv.

8. Komponenta “Transaction Handler” přijme událost a namapuje podporované databázové události na události aplikační. Následně události deleguje do komponenty “Event Processor”.

9. Komponenta “Event Processor” zpracuje události v pořadí, ve kterém byly vytvořeny a navrátí se do komponenty “Transaction Handler”.

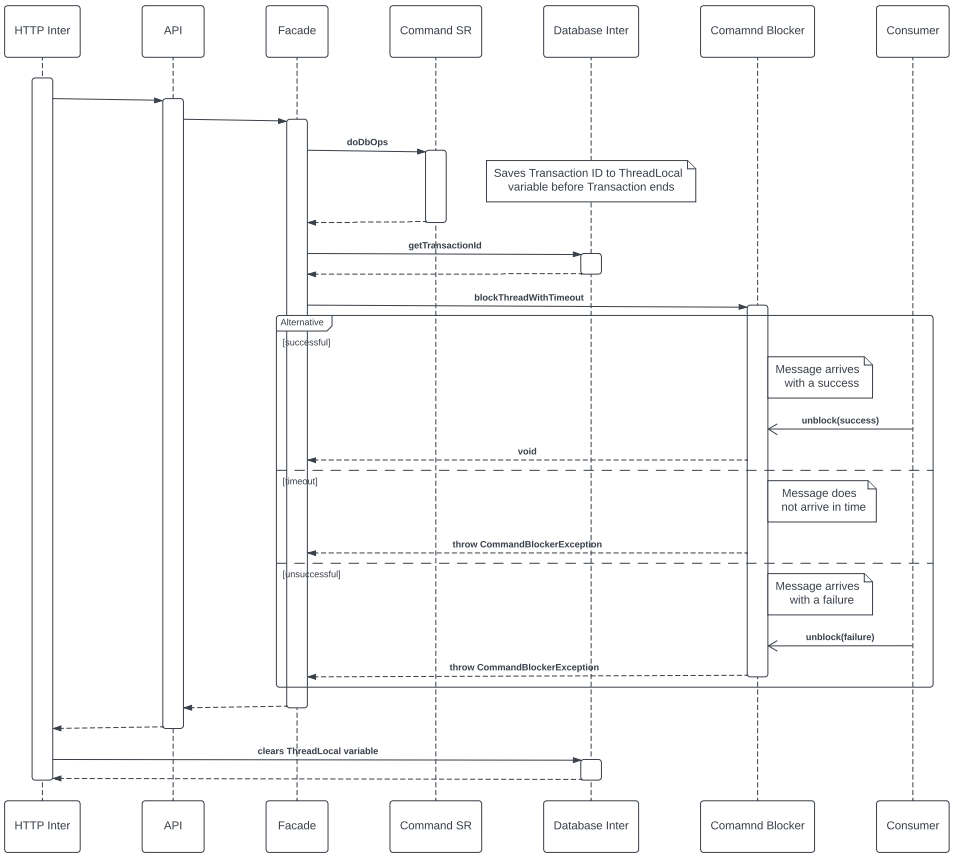
10. Komponenta “Transaction Handler” vytvoří událost o úspěchu/neúspěchu zpracování události a zašle ji do systému zpráv. Následně komponenta “Command Blocker” přijme zprávu a přiřadí na základě unikátního identifikátoru transakce k vláknu spjatému s požadavkem. V případě úspěchu zpracování události odblokuje vlákno, které bylo zablokováno v kroku 4. V případě neúspěchu zpracování události vyhodí výjimku.

Detailněji je proces popsán v následujících podsekcích.

5.5.1 Blokace požadavků

Blokace požadavků uživatele je nezbytné pro zajištění konzistence dat. Pokud by nebyl požadavek zablokován, mohlo by dojít k situaci, kdy by uživatel viděl data, která nejsou aktuální. Například by přidal produkt do košíku a při následném zobrazení košíku by produkt chyběl. Tento problém je řešen pomocí blokace požadavku po dobu synchronizace vedlejší databáze (materializovaného pohledu). V případě, že by synchronizace trvala příliš dlouho, je požadavek odblokován (timeout). V takovém případě může být uživateli zobrazena chybová hláška s informací, že data nemusí být aktuální.

Detailněji je proces blokace znázorněn sekvenčním diagramem 5.6.



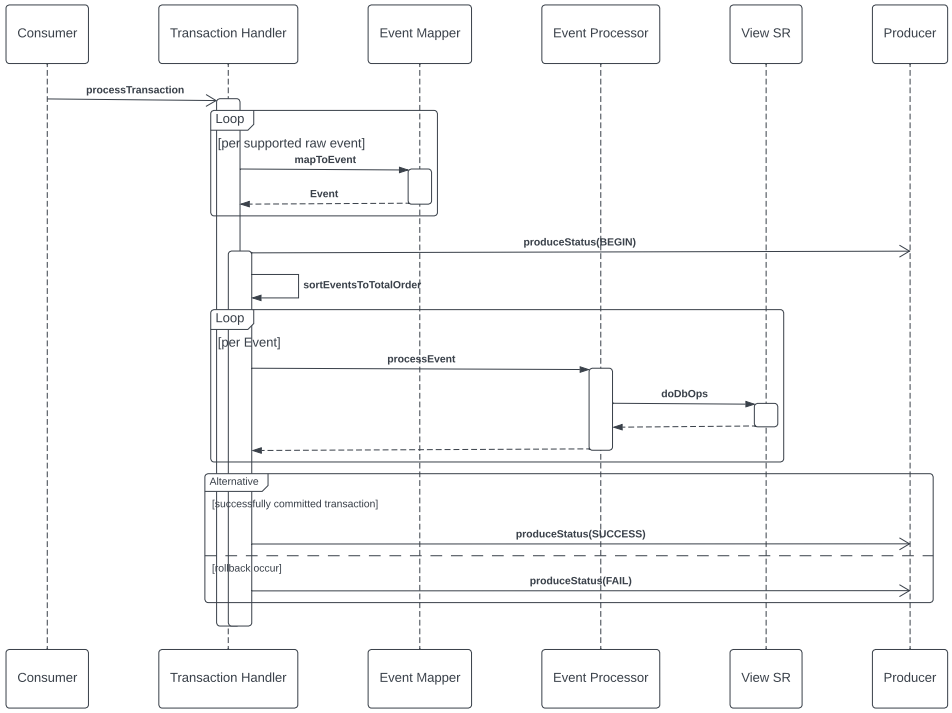
Obrázek 5.6: Sekvenční diagram blokace požadavku

- **HTTP Inter** - HTTP interceptor zajišťující čištění proměnných spjatých s vláknem požadavku.
- **API** - API vrstva.
- **Facade** - Vrstva zajišťující komunikaci s ostatními komponentami.
- **Command SR** - Servisní a repozitní vrstvy pro zpracování požadavků typu command.
- **Database Inter** - Databázový interceptor zajišťující uložení unikátního identifikátoru transakce do proměnné spjaté s vláknem požadavku před potvrzením transakce (commit).
- **Command Blocker** - Komponenta zajišťující mechanismus blokace požadavku.
- **Consumer** - Konzument asynchronních událostí.

5.5.2 Zpracování událostí

Pro zachování jednoduchosti a přehlednosti logiky transformace dat do materializovaného pohledu je zvolen mechanismus, který zajišťuje, že pořadí událostí je vždy stejné. Jedná se o pořadí v jakém jsou události vytvořeny v rámci jedné transakce, tzn. dle jejich závislostí v relační databázi. Tímto způsobem je možné zaručit, že se například při přidání produktu do košíku (košík zatím nebyl vytvořen) nejprve provede tvorba košíku a pak teprve následné vložení produktu do košíku. V opačném případě by mohlo dojít k situaci, kdy by se zboží vložilo do neexistujícího košíku a tím došlo k porušení integritního omezení. Mechanismus je popsán diagramem 5.7.

- **Consumer** - Konzument seskupených databázových událostí dle transakčního identifikátoru.
- **Transaction Handler** - Komponenta zajišťující zpracování seskupených databázových událostí.
- **Event Mapper** - Komponenta zajišťující mapování podporovaných databázových událostí na události aplikační.
- **Event Processor** - Komponenta zajišťující zpracování událostí aplikační vrstvou.
- **View SR** - Servisní a repozitní vrstvy zajišťující aktualizaci dat ve vedlejší databázi.
- **Producer** - Producent asynchronních událostí o stavu zpracování transakce.



Obrázek 5.7: Sekvenční diagram zpracování databázových událostí

Problémem tohoto řešení je situace, kdy agregujeme třídy, např. “CartItemView”. V tomto případě agregujeme vazební tabulku reprezentovanou třídou “Cart_CartItem” s třídou “CartItem”. Situaci si můžeme představit na zmíněném příkladu s přidáním produktu do košíku. V tomto případě se vygenerují události v následujícím pořadí:

1. Vytvoření košíku (“Cart”).
2. Vytvoření položky košíku (“CartItem”).
3. Vytvoření vazby mezi košíkem a položkou košíku (“Cart_CartItem”).

Pro vytvoření agregace “CartItemView” potřebujeme, aby se událost č.1 zpracovala první a následně události č.2 a 3 zároveň. To je však v rozporu se zpracováním událostí v pořadí, ve kterém byly vytvořeny. Tento problém je řešen pomocí pomocné třídy “CartItemRaw”, která slouží jako mezistupeň pro vytvoření třídy “CartItemView”. Navržený mechanismus se zachová následovně:

1. Obdrží událost o vytvoření košíku (“Cart”) - vytvoří “CartView”.
2. Obdrží událost o vytvoření položky košíku (“CartItem”) - vytvoří “CartItemRaw”.
3. Obdrží událost o vytvoření vazby mezi košíkem a položkou košíku (“Cart_CartItem”) - načte “CartItemRaw” a vytvoří “CartItemView”.

Poznámka: Stejný problém by nastal i v situaci agregace tříd s vazbou m:n. Pro jednoduchost je zvolen uměle vytvořený příklad s vazbou 1:n.

Kapitola 6

Implementace příkladové knihovny

V této kapitole vybereme technologie a výběr odůvodníme. Následně si představíme implementaci příkladové knihovny v podobě diagramu komponent a jeho popisu. Dále detailněji popíšeme implementaci klíčových mechanismů aplikace a navrhneme jejich možné vylepšení. Na závěr kapitoly si představíme, jakým způsobem je možné aplikaci nasadit, škálovat a migrovat její databáze.

6.1 Výběr technologií

Výběr technologií je subjektivně zaměřen především na technologie, které jsou v současné době využívány v praxi. Zároveň jsou vybrány technologie, které jsou vývojáři dobře známé a umožňují rychlý vývoj.

6.1.1 Kotlin

Kotlin je open source programovací jazyk založený na objektově orientovaném jazyce Java a jeho spouštěcí platformě JVM. Jedná se o jazyk snažící se využít výhod Javy a obohatit o několik dalších. Některé z nich jsou:

- **Null safety** - Možnost explicitně definovat proměnnou jako *non-null*.
- **Zkrácení syntaxe** - Zkrácení syntaxe pro vytváření tříd, přístup k proměnným (automatické generování metod get a set u veřejných proměnných) apod.
- **Vyšší podpora funcionálního programování** - Disponuje vyšší podporou funkcionálního programování. Například možnost definovat funkce jako metody jiných tříd apod.

6.1.2 Gradle

Gradle je open source nástroj automatizující sestavování programů a správu závislostí původně pro programovací jazyk Java. Jedná se o novější nástroj stavící na zkušenostech z předchozích nástrojů Apache Ant a Apache Maven.

6.1.3 Spring Boot

Spring Boot je open source framework pro vytváření aplikací založených na jazycích Java nebo Kotlin. Jedná se o framework, který využívá Spring Framework a jeho moduly. Výhody použití Spring Bootu:

- **Nastavení** - Umožňuje rychlé nastavení aplikace pomocí konfiguračních souborů, navíc je za nás velké množství konfigurací již předem nastaveno.
- **Aplikační server** - Disponuje přednastaveným servlet kontejnerem Apache Tomcat, který nám dovoluje spustit aplikaci bez nutnosti instalace externího aplikačního serveru.
- **Moduly** - Poskytuje mnoho modulů, které nám umožňují rychle rozšiřovat aplikaci o další funkcionality bez nutnosti vlastní implementace, například Spring JPA, Spring Security apod.

6.1.4 Kafka

Kafka je open source distribuovaný systém sloužící pro asynchronní komunikaci mezi komponentami. Je založen na produkování zpráv do topiků (unikátní kategorie zpráv) a následnou konzumaci. Tímto přístupem je zaručena nízká provázanost jednotlivých komponent. Zprávy v jednotlivých topikách se zároveň ukládají (obecně zaručuje ACID vlastnosti) - tím je zaručeno, že se i při výpadku zprávy neztratí.

Při implementaci této knihovny je použita Kafka od společnosti Confluent[1], která disponuje několika rozšířeními, které nám umožní jednodušší použití Kafka systému (např. schema registry). Využití software od společnosti Confluent je bezplatné za podmínek nekomerčního využití a vývoje s maximálním použitím jedné broker nody [33].

6.1.5 Schema registry

Schema registry je externí server fungující jako databáze schémat jednotlivých zpráv. Pokud schema registry podporuje námi zvolený datový formát, v tomto případě JSON, můžeme pro jednotlivé topiky uchovávat schéma hodnot (values) i klíčů (keys) a dokonce je i verzovat. Tyto informace jsou uchovávány v systémovém topiku. Schema registry se dá použít jako pouhá knihovna, ve které se snadno dozvíme schéma zpráv, nebo může sloužit i jako validátor kompatibility při zápisu do topiku. [34]

V tomto případě je schema registry využita i jako validátor.

6.1.6 Kafka connect

Kafka connect je open source komponenta, která lze použít ke komunikaci mezi Kafkou a ostatními datovými systémy. Komponentu můžeme použít buď samostatně nebo distribuovaně (vytvořit Kafka connect cluster) a následně

[1] https://www.confluent.io

propojit s naším existujícím Kafka clusterem. Pokud takto připravený Kafka connect disponuje rozšířením pro datový systém, se kterým chceme komunikovat, stačí nám vytvořit konfigurační soubor a zaregistrovat skrze Kafka connect REST API. [35]

V tomto případě je použito rozšíření Debezium pro databázi PostgreSQL, které nám umožní sledovat změny v databázi a zároveň je zapisovat do Kafka topiků.

6.1.7 Debezium

Debezium je open source platforma poskytující knihovny, tzv. konektory, pro sledování změn v databázích. Platforma je často používána pro synchronizaci dat z databáze do jiné databáze nebo zcela odlišných datových systémů. Proces odposlouchávání změnových událostí se nazývá CDC, viz sekce 2.4. [36, 37]

V případě příkladové knihovny se jedná o konektor pro databázi PostgreSQL, který můžeme nahrát jako rozšíření do komponenty Kafka connect a následně jej zaregistrovat. Pro registraci konektoru je nutné nastavit konfigurační soubor, který nám umožní nastavit např. zdrojovou databázi, cílový Kafka topik apod.

6.1.8 PostgreSQL

Jedná se o open source databázový systém, který je založen na objektově relačním modelu. Tento databázový systém je zvolen na základě jednoduchosti použití a též na skutečnosti existence bezplatného Debezium konektoru, který nám umožní sledovat změny v databázi a zároveň je zapisovat do Kafka topiků.

6.1.9 Flyway

Flyway je open source nástroj, který nám umožňuje migrovat databázové schéma při spuštění aplikace. Tento nástroj je zvolen na základě jednoduchosti použití a kompatibility s Java aplikacemi. V případě, že je potřeba změnit databázové schéma, stačí vytvořit nový SQL skript a přidat jej do projektu. Při následném spuštění aplikace se skript automaticky spustí a provede změny v databázi.

6.1.10 Docker

Docker je platforma umožňující rychlé a snadné nasazování komponent v izolovaných kontejnerech. Díky tomuto řešení je možné jednotlivé komponenty spustit na jakémkoli operačním systému, který Docker podporuje, s použitím jediné konfigurace. Kontejner si všechny soubory a nastavení drží ve svém kontextu, a tedy nijak neovlivňuje svého hostitele[2]. Další výhodou je, že při spuštění kontejneru je možné definovat, jaké prostředky (CPU, RAM apod.)

[2] fyzický nebo virtualizovaný server, na kterém je kontejner spuštěn

má k dispozici. Zároveň je nám zaručeno stažení všech závislostí, které jsme definovali v konfiguračním souboru (Dockerfile).

V případě příkladové knihovny je navíc použit nástroj docker-compose, který nám pomocí konfiguračního souboru (docker-compose.yml) umožní definovat jednotlivé kontejnery, jejich vazby a další parametry.

6.2 Komponent diagram

Aplikace se skládá z několika uzlů a komponent, viz. diagram komponent 6.2. V diagramu je pro přehlednost vynechána komponenta schema registry, která by byla nepojena na Kafku a všechny ostatní komponenty využívající Kafku.

- **Order microservice (business)** - Mikroslužba zodpovědná za zpracování zápisové (command) logiky v kontextu objednávek.
- **Order microservice (view)** - Mikroslužba zodpovědná za zpracování čtecí (query) logiky v kontextu objednávek.
- **Transaction grouper microservice** - Obecná mikroslužba zodpovědná za seskupování událostí provedených v rámci jedné databázové transakce. Je závislá pouze na technologiích Kafka a Debezium. Nerozumí databázovým schématům.
- **Order updater microservice** - Mikroslužba zodpovědná za zpracování seskupených transakcí v kontextu objednávek.
- **Kafka** - Distribuovaný systém zpráv.
- **Kafka Connect** - Systém zajišťující komunikaci mezi Kafkou a ostatními datovými systémy.
- **Business PostgreSQL** - Databáze zodpovědná za ukládání byznys dat v kontextu objednávek.
- **View PostgreSQL** - Databáze zodpovědná za ukládání dat tvořících materializový pohled nad byznys daty.
- **Command Service a View Service** - Vrstvy obsahující byznys logiku.
- **Command Repository a View Repository** - Vrstvy obsahující logiku pro práci s databází.
- **Kafka Consumer a Kafka Producer** - Komponenty zajišťující komunikaci s Kafkou.
- **HTTP Interceptor** - Interceptor zodpovědný za mazání proměnných spjatých s vláknem požadavku.
- **Facade** - Fasáda zodpovědná za řízení jednotlivých komponent při zpracování požadavku.

- **Hibernate Interceptor** - Interceptor zodpovědný za získání a uložení unikátního identifikátoru databázové transakce při potvrzení transakce (commit).
- **Command Blocker** - Mechanismus zodpovědný za blokaci vlákna požadavku během procesu synchronizace dat mezi databázemi. Podrobněji je implementace vysvětlena v sekci 6.2.1.
- **Event transaction service** - Zprostředkovává seskupení událostí provedených v rámci jedné databázové transakce. Podrobněji je implementace vysvětlena v sekci 6.2.2.
- **Transaction handler** - Komponenta zodpovědná za řízení jednotlivých komponent při zpracování seskupených databázových událostí.
- **Event Mapper** - Komponenta zodpovědná za mapování podporovaných událostí z databázového schématu do aplikačních událostí. Podrobněji je implementace vysvětlena v sekci 6.2.3.
- **Event processor** - Komponenta zodpovědná za zpracování aplikačních událostí pomocí návrhového vzoru visitor. Podrobněji je implementace vysvětlena v sekci 6.2.4.

6.2.1 Blokace vlákna požadavku

Blokace vlákna je realizována pomocí mapy, kde klíčem je unikátní identifikátor transakce a hodnotou je třída “CompletableFuture”.

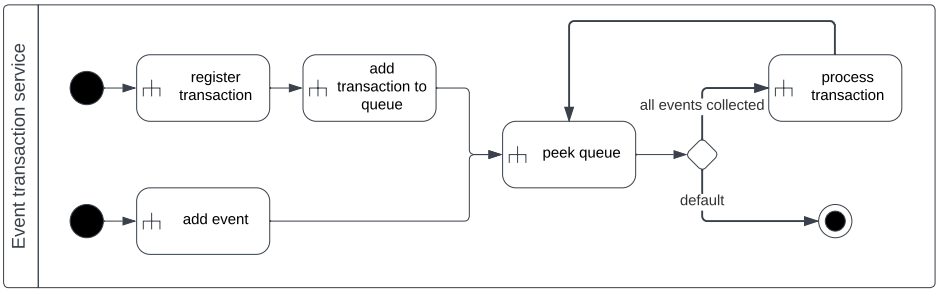
Při zahájení blokace je vytvořena instance třídy “CompletableFuture”, která se uloží do mapy. Vlákno požadavku je následně zablokováno pomocí metody “join” na instanci třídy “CompletableFuture”. Při přijmutí události o dokončení transakce je z mapy vyhledána instance třídy “CompletableFuture”, která je z mapy odstraněna a následně zavolána její metoda “complete” v případě úspěchu nebo “completeExceptionally” v případě neúspěchu. Tím je vlákno požadavku odblokováno.

Pokud je vlákno požadavku blokováno déle, než je definováno (vyprší timeout), je z mapy instance třídy “CompletableFuture” odstraněna a vlákno požadavku odblokováno. Tím je zabráněno nekonečně dlouhé blokaci vlákna požadavku.

6.2.2 Transakční seskupení

Seskupení událostí je realizováno samostatnou mikroslužbou z důvodu možné obecné implementace. Mikroslužba je zodpovědná pouze za seskupení událostí a neobsahuje žádnou byznys logiku. Je závislá pouze na technologiích Kafka a Debezium. Díky tomu je možné tuto mikroslužbu použít pro seskupení událostí i z jiných databázových systémů, které jsou podporovány technologií Debezium. Při použití jiné databáze může být nutné implementovat datové třídy popisující události, které jsou produkovány Debeziem.

Debezium produkuje následující události (bližší popis [37]):



Obrázek 6.1: Diagram aktivit transakčního seskupovače

1. **Změnová událost tabulky** - Událost obsahující informace o změně v tabulce (obsahuje například typ události, hodnoty před a po změně, unikátní identifikátor transakce, pořadí události v transakci). Pro každou tabulku jsou události produkovány do samostatného topiku. Topikům lze definovat prefix, díky kterému je možné události konzumovat jediným konzumentem.

2. **Transakční událost** - Událost obsahující metadata o transakci (obsahuje například unikátní identifikátor transakce, počet událostí produkovaných v transakci). Událost je produkována do samostatného topiku.

Tyto události je nutné asynchronně konzumovat a párovat. Následně je nutné takto seskupené události produkovat do topiku ve stejném pořadí, jako byly provedeny transakce. Tím je zajištěna konzistence dat mezi databázemi v případě výchozí izolační úrovně *read committed*.

Pořadí transakcí je zajištěno topikem obsahující transakční události. Z tohoto důvodu je nutné tento topik konzumovat pouze jedním vláknem a jeho partition rozdělení je nutné nastavit na pouze jednu partition. V opačném případě by nebylo možné zajistit pořadí těchto událostí.

Párování událostí je naznačeno diagramem aktivit 6.1. Před jeho popisem je nutné definovat následující pojmy:

- **Seskupení** - Jedná se o třídu obsahující transakční událost a pole změnových událostí tabulek.
- **Kolekce událostí** - Jedná se o mapu, kde klíčem je unikátní identifikátor transakce a hodnotou je třída Seskupení.
- **Fronta transakcí** - Jedná se o frontu (queue) obsahující unikátní identifikátory transakcí.
- **Kompletnost seskupení** - Seskupení obsahuje transakční událost a všechny změnové události se stejným unikátním identifikátorem transakce.

Diagram aktivit 6.1 obsahuje následující průchody:

- **Průchod A** - Průchod je aktivován příjmem změnové události tabulky.
  1. Přidání změnové události tabulky do *kolekce událostí*.
  2. Nahlédnutí do *fronty transakcí*.
  3. Pokud je *seskupení kompletní*, je unikátní identifikátor z fronty odebrán a výsledné *seskupení* je produkováno do výstupního topiku.
- **Průchod B** - Průchod je aktivován příjmem transakční události.
  1. Registrace transakční události do *kolekce událostí*.
  2. Přidání unikátního identifikátoru transakce do *fronty transakcí*.
  3. Nahlédnutí do *fronty transakcí*.
  4. Pokud je *seskupení kompletní*, je unikátní identifikátor z fronty odebrán a výsledné *seskupení* je produkováno do výstupního topiku.

Aktivita *nahlédnutí do fronty transakcí* je opakována, dokud na vrcholu *fronty transakcí* není unikátní identifikátor transakce takový, pro který neplatí, že *seskupení* v *kolekci událostí* je *kompletní*.

Možné vylepšení

Aktuální implementace nezahrnuje situaci pádu aplikace během seskupování událostí. Při pádu aplikace jsou ztraceny všechny nezpracované události z paměti a zároveň jsou tyto události již zkonzumované z Kafka topiků.

Tento problém by bylo možné vyřešit ukládáním *kolekce událostí* a *fronty transakcí* do databáze. V případě pádu aplikace by bylo možné pokračovat v seskupování událostí od posledního uloženého stavu.

Dalším možným řešením je vytvořit mechanismus zaznamenávající offsety topiků naposledy zpracovaného seskupení. V případě pádu aplikace by bylo možné resetovat offsety topiků na poslední zpracované seskupení a pokračovat v seskupování událostí od tohoto bodu.

6.2.3 Mapování událostí

Mapování databázových změnových událostí je prováděno v mikroslužbě starající se o propagaci seskupených událostí do vedlejší databáze (materializovaného pohledu). Je založeno na mechanismu, kde je pro každou tabulku vytvořena třída implementující interface “RawEventMapper” předepisující následující metody:

- **supports** - Metoda přijímá jako parametr událost a vrací boolean hodnotu. Touto metodou je možné určit, zda je událost implementovanou třídou podporována.
- **mapToEvent** - Metoda přijímá jako parametr událost a vrací aplikační událost. Touto metodou je možné událost mapovat na aplikační událost.

Každá takto implementovaná třída je beanou Spring IoC kontejneru. Při procesu mapování je vyhledána první třída podporující událost a následně zavolána metoda "mapToEvent". Pokud není nalezena žádná třída podporující událost, nic se mimo logování nestane. Toto chování je zvoleno z důvodu, že ne všechny databázové změnové události musí být pro aplikaci relevantní.

6.2.4 Zpracování aplikačních událostí

Tato sekce navazuje na sekci předchozí. Nyní jsme v situaci, kdy máme aplikační události, které je nutné zpracovat. Před zpracováním jsou události seřazeny do pořadí, ve kterém byly v transakci vykonány. Následně je zahájena transakce, ve které skrze události iterujeme a voláme jejich metodu "process" (alternativa k "visit" u návrhového vzoru visitor). Tato metoda přijímá jako parametr interface "EventProcessor", ve kterém je uschována jednoduchá logika volající servisní vrstvu aplikace.

Pokud je zpracování všech událostí úspěšné, je transakce potvrzena a vytvořena stavová událost typu "SUCCESS". V opačném případě je transakce zrušena a vytvořena stavová událost typu "FAILURE". Tyto zprávy jsou konzumovány "Order microservice (business)", která na základě stavové události odblokuje vlákno požadavku.

6.3 Škálování

Návrh aplikace je založen na mikroservisní architektuře. Každá mikroslužba je samostatná komponenta aplikace, kterou lze škálovat nezávisle na ostatních mikroslužbách. V případě, že je mikroslužba zatížena, je možné její počet instancí zvýšit. V případě, že instance mikroslužby nejsou využívány, je možné jejich počet snížit. Mikroslužby jsou navrženy tak, aby bylo možné jejich počet instancí měnit za běhu aplikace.

Takovým způsobem lze škálovat následující mikroslužby:

- **Order microservice (business)**
- **Order microservice (view)**

Problematické škálování nastává u mikroslužeb:

- **Transaction grouper microservice**
- **Order updater microservice**

V případě škálování *transaction grouper* mikroslužby by bylo nutné zajistit distribuci změnových událostí tabulek do více topiků například operací *unikátní identifikátor transakce mod počet instancí mikroslužby*. Tímto by bylo zajištěno, že všechny události jedné transakce budou zpracovány v rámci jedné instance mikroslužby. Následně by bylo nutné zajistit, aby byly takto

distribuovány i transakční události a zároveň existovala společná fronta transakcí pro zachování pořadí zpracování. Toto řešení je velmi komplexní a není implementováno. Zároveň se jedná pouze o teoretickou optimalizaci.

U mikroslužby *order updater* nastává podobný problém. Propagace dat do vedlejší databáze je nutné provádět sériově z důvodu zajištění zpracování událostí v deterministickém pořadí. V opačném případě by bylo nutné rekonstruovat závislosti mezi jednotlivými událostmi komplexní logikou (stejnou jakou používá zdrojová databáze, navíc se znalostí izolační úrovně transakcí). Z tohoto důvodu se domnívám, že je vhodné tuto mikroslužbu škálovat pouze vertikálně. Tedy zvýšit výpočetní výkon jediné instance mikroslužby.

6.4 Migrace databázového schéma

Databázové schéma se s časem mění. Jeho migrace je v této implementaci umožněna následujícím způsobem:

1. Odstavení mikroslužby *order (business)*. Tímto je zajištěno, že proces CDC nebude vytvářet nové změnové události.
2. Vyčkání na zpracování všech změnových událostí.
3. Odstavení mikroslužeb *order updater* a *order (view)*.
4. Migrace databázového schématu vedlejší databáze.
5. Spuštění mikroslužeb *order updater* a *order (view)*.
6. Migrace databázového schématu hlavní databáze.
7. Spuštění mikroslužby *order (business)*.

Migraci databázového schématu zajišťuje technologie flyway. O migraci databázového schématu hlavní databáze se stará mikroslužba *order business*. O migraci databázového schématu vedlejší databáze se stará mikroslužba *order updater*.

Mikroslužba *order updater* může být upravena dvěma způsoby:

- **Zpětně kompatibilní změna** - Pokud se jedná o zpětně kompatibilní změnu, je možné provést pouze snadné úpravy jako jsou například přidání aplikačních událostí a přidání volitelných atributů.
- **Nekompatibilní změna** - Pokud se jedná o nekompatibilní změnu, je možné se vydat dvěmi směry.
  1. **Porušení zpětné kompatibility** - Upravit mikroslužbu tak, že dokáže zpracovávat pouze nově přicházející událostí.
  2. **Zachování zpětné kompatibility** - Vytvořit novou třídu implementující interface “RawEventMapper” a upravit starou implementaci tak, aby byly zachovány obě verze. Tímto je zajištěno, že mikroslužba dokáže zpracovávat jak nové, tak staré události.

6.5 Migrace dat

V současné implementaci není podporována propagace dat z hlavní databáze do vedlejší jiným způsobem, než pomocí provádění změn dat v hlavní databázi po registraci Debezium konektoru. Tyto databázové změnové události jsou typu CREATE, UPDATE a DELETE. Jedním z možných rozšíření je zahrnutí podpory tzv. snapshotů.

Debezium snapshoty podporuje pomocí JDBC ovladače (driver). Při aktivaci snapshotu je pro každý řádek vytvořena změnová událost typu READ, která dosavadní implementací není podporována z důvodu absence transakčních metadat. Transakční metadata jsou v nynější implementaci důležitá z důvodu zajištění zpracování změnových událostní v deterministickém pořadí, tzn. že je například při vytvoření vazby *jedna k jedné* bez vazební tabulky je nejprve vytvořena událost CREATE pro tabulku A a až poté pro tabulku B obsahující cizí klíč ukazující na primární klíč tabulky A. V případě snapshotu nelze pořadí tímto způsobem zaručit. Zpracování snapshotů by bylo možné podporovat například následujícím způsobem:

1. Mikroslužba "Order updater" by při svém spuštění iterovala v přesně definovaném pořadí topiky, do kterých jsou produkovány databázové změnové události tabulek.
2. Pro každý topik by konzumovala události typu READ až do doby nalezení události jíného typu nebo do doby vypršení časového limitu (timeout) zahájeného poslední přijatou událostí typu READ/počátkem konzumace.
3. Událost jiného typu, než je READ nebo vypršení časového limitu by znamenalo, že se je možné posunout na další topik v pořadí.
4. Při dokončení této iterace by se mikroslužba "Order updater" přepnula do výchozího režimu.

6.6 Nasazení aplikace

Nasazení je zajištěno pomocí technologie Docker a nástroje docker-compose. Pro chod aplikace je nutné spustit kontejnery "Order microservice (business)", "Order microservice (view)", "Order updater microservice", "Transaction grouper microservice", "Kafka Connect", "Zookeeper", "Kafka broker", "PostgreSQL" a "Schema registry". Veškeré konfigurace jsou uvedeny v souboru *docker-compose.yml*. Tento soubor obsahuje nastavení jednotlivých komponent včetně jejich komunikace mezi sebou pomocí Docker virtualizované sítě. Návod, jak aplikaci spustit je k nalezení v souboru *README.md*, kde je rovněž možné dohledat cesty k REST API specifikacím vytvořeným dle standardu OpenAPI[3]. Oba zmíněné soubory jsou přiloženy v kořenovém adresáři příkladové knihovny.

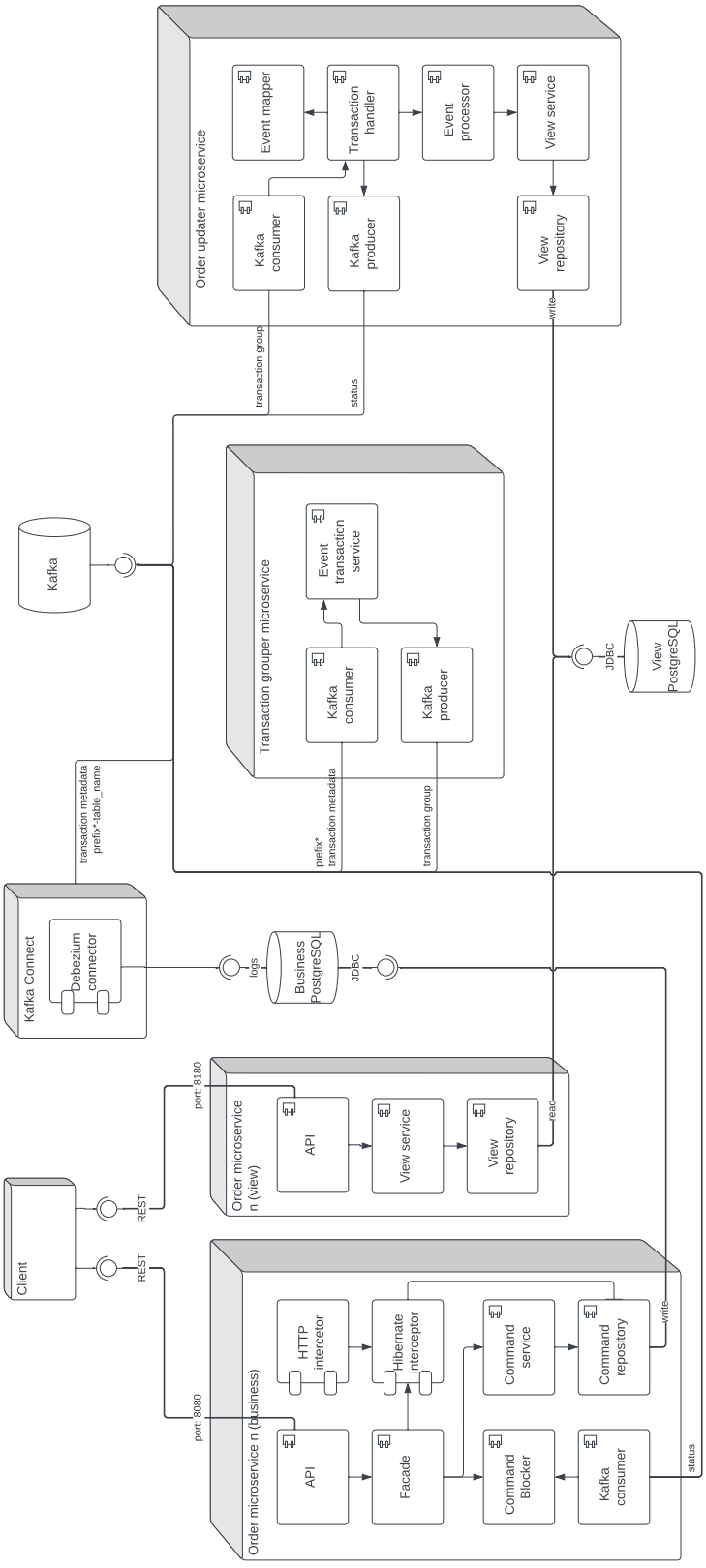
Z Docker virtualizované sítě jsou vystaveny následující porty:

[3] https://www.openapis.org

- **8080** - REST API mikroslužby “Order microservice (business)”.
- **8180** - REST API mikroslužby “Order microservice (view)”.
- **8083** - REST API Kafka Connect. Tento port je vystaven pouze z důvodu snadného použití nástroje curl při vytváření Debezium konektoru. V reálném nasazení by bylo vhodné tento port z bezpečnostních důvodů nevystavovat.

Diagram nasazení není vytvořen z důvodu jeho komplexnosti a především variabilitě. V příkladovém případě se jedná o nasazení na jediný server (na počítač vývojáře), ale v reálném nasazení by se jednalo o nasazení na více serverů např. pomocí technologie Kubernetes[4]. Navíc například technologie Kafka nebývá nasazena na stejném serveru jako aplikace, ale na samostatném clusteru. Z tohoto důvodu je upřednostněn podrobnější diagram komponent 6.2, na kterém je jsou znázorněny i topiky, do/z kterých dané komponenty produkují/konzumují zprávy.

[4]software řídící virtualizaci na úrovní OS s podporou horizontální rozšiřitelnosti (https://kubernetes.io)



Obrázek 6.2: Diagram komponent

Kapitola 7
Testování

Pro testování příkladové aplikace byl zvolen způsob manuálního volání REST API endpointů pomocí nástroje Insomnia[1] s očekáváním úspěšné odpovědi. Export konfigurace z tohoto nástroje je k nalezení v kořenovém adresáři přiložené příkladové knihovny pod názvem *insomnia_export.json*. Tento způsob byl zvolen z důvodu složité automatizace *end-to-end (integračních)* testů kvůli nutnosti nasazení celé aplikace. Všechny testy byly prováděny na lokálním prostředí, na kterém byly spuštěny všechny potřebné služby.

7.1 Jednotkové testy

Jednotkovými testy byla podrobena komponenta "Event transaction service", ve které probíhá seskupování databázových změnových událostí tabulek a transakčních událostí dle unikátního identifikátoru transakce. Tento mechanismus je rizikovou částí aplikace vzhledem k přetečení paměti, záměny pořadí transakcí, špatně implementované vícevláknové obsluze apod. Testy jsou implementovány pomocí knihoven JUnit 5 a Mockito.

Jednotkové testy pokrývající byznys logiku nebyly vytvořeny z důvodu jejího příkladového charakteru a nízkého rozsahu.

7.2 Testování výkonu

Pro testování výkonu byla použita knihovna ApacheBench[2]. Tato knihovna umožňuje vytvořit požadavky na zadanou IP/URL adresu a zobrazit statistiky o průběhu testu. Jedná se celkově o 18 testů, které byly provedeny na endpointech pro přidání zboží do košíku, získání obsahu košíku a speciálně vytvořeným endpointem reprezentující získání obsahu košíku bez využití materializovaného pohledu. Testy byly provedeny v rámci stejné Docker sítě pomocí operačního systému Ubuntu. Příkazy, výsledky testů a pomocnou třídu "ReadBench" je možné nalézt v přiložených souborech.

[1] https://insomnia.rest
[2] https://httpd.apache.org/docs/2.4/programs/ab.html

- **T1 přidání zboží do košíku** - Náplní testů je otestovat výkon synchronizace dat mezi mikroslužbami. Výsledky testů jsou zaneseny v tabulce 7.1.
- **T2 získání obsahu košíku** - Náplní testů je otestovat výkon čtení z materializovaného pohledu. Výsledky testů jsou zaneseny v tabulce 7.2.
- **T3 získání obsahu košíku** - Náplní testů je otestovat výkon čtení bez materializovaného pohledu. Pro tento účel byl vytvořen speciální endpoint v mikroslužbě "Order microservice (business)" odpovídající endpointu pro získání obsahu košíku mikroslužby "Order microservice (view)". Výsledky testů jsou zaneseny v tabulce 7.3.

počet požadavků	počet vláken	požadavek /sekunda	čas požadavku (průměr) [ms]
100	1	1,90	525,796
100	5	9,02	554,265
1000	100	181,77	550,145
10000	200	363,18	550,685
12000	300	355,06	844,929
12000	400	359.24	1113,449

Tabulka 7.1: Tabulka zaznamenaných výsledků testů T1

počet požadavků	počet vláken	požadavek /sekunda	čas požadavku (průměr) [ms]
100	1	1394,53	0,717
100	5	2530,81	1,976
1000	100	12359,41	8,091
10000	200	19166,78	10,435
12000	300	19250,36	15,584
12000	400	18549,18	21,564

Tabulka 7.2: Tabulka zaznamenaných výsledků testů T2

počet požadavků	počet vláken	požadavek /sekunda	čas požadavku (průměr) [ms]
100	1	1008,77	0,991
100	5	2204,49	2.268
1000	100	8050,49	12,422
10000	200	11130,79	17,968
12000	300	11058,15	27,129
12000	400	10963,34	36.485

Tabulka 7.3: Tabulka zaznamenaných výsledků testů T3

Při bližším ohledání výsledků testů je možné pozorovat, že výkon všech typů testů se zastavuje při použití 200 a více vláken. Toto chování může být

způsobeno dosažením limitu aplikace, ale také limitu hardware, na kterém byly testy prováděny. Ze srovnatelného bohu zastavení růstu výkonu lze vyhodnotit, že synchronizace dat mezi mikroslužbami není v rámci těchto testů omezující.

U výsledků testů T2 a T3 je možné pozorovat, že výkon čtení z materializovaného pohledu je výrazně vyšší než přímé čtení původních dat. Toto chování je způsobeno tím, že materializovaný pohled obsahuje data připravená pro čtení. V případě přímého čtení původních dat je nutné provést výpočet celkové ceny košíku a počtu položek košíku při každém požadavku. Tento výpočet je v rámci testování zanedbatelný, protože se jedná o jednoduchou operaci sčítání. Hlavním důvodem vyššího výkonu je zde absence jedné z join operací při čtení z materializovaného pohledu. V případě složitějších modelů je možné očekávat výraznější rozdíly ve výkonu.

Z očekávaných výsledků testů lze vyhodnotit, že implementace návrhových vzorů CQRS a materializovaného pohledu byla úspěšná.

Kapitola 8
Závěr

Práci je možné strukturovat do tří hlavních částí. První část (kapitoly 2, 3 a 4) se věnuje teoretickému základu. Druhá část (kapitoly 5 a 6) využívá tuto teorii k navržení a implementaci příkladové knihovny. Poslední část (kapitola 7) hodnotí úspěšnost implementace příkladové knihovny prostřednictvím testování.

V první části práce jsou vysvětleny klíčové pojmy, které je nezbytné správně chápat pro snazší porozumění mikroservisní architektury a jejím cílům. Současně byly v této části popsány některé další architektury, které jsou často s mikroslužbami srovnávány nebo dokonce zaměňovány. Následně je detailně popsána samotná mikroservisní architektura spolu s jejími výhodami a nevýhodami. Kromě toho jsou uvedeny některé příklady situací, ve kterých je vhodné tuto architekturu použít, a naopak kdy je vhodné se jejímu použití vyvarovat. Poslední kapitola této části prezentuje návrhové vzory charakteristické pro mikroslužby. Návrhové vzory jsou stručně představeny prostřednictvím popisu, který zahrnuje jejich účel a možné konkrétní využití. V této kapitole jsou také popsány návrhové vzory CQRS a materializovaný pohled, které byly vybrány pro implementaci v rámci příkladové knihovny.

Druhá část práce se zaměřuje na návrh a implementaci příkladové knihovny. Nejprve je představen návrh knihovny pomocí diagramu případů užití a diagramu tříd. Zároveň je zde pomocí zjednodušeného obrázku diagramu komponent popsán průchod dat napříč komponenty. Následně je popsána implementace příkladové knihovny. Jsou vybrány technologie použité při implementaci a popsán postup implementace jednotlivých komponent včetně možných zlepšení. Kromě toho je také popsán způsob škálování, migrace a nasazení příkladové knihovny.

Poslední část práce se věnuje testování příkladové knihovny. Bylo zvoleno manuální end-to-end testování pomocí nástroje Insomnia, jednotkové testování kritických částí příkladové knihovny s využitím nástrojů JUnit 5 a Mockito a testování výkonu pomocí nástroje ApacheBench. Výsledky testů výkonu potvrdily zvýšení výkonu operace čtení, a tedy správnost implementace návrhových vzorů.

Závěrem práce lze konstatovat, že bylo dosaženo všech cílů stanovených v úvodu této práce. Příkladovou knihovnu lze navíc použít jako šablonu pro vývoj mikroslužeb, u kterých je vyžadován vysoký výkon operací zprostřed-

kovávajících čtení na úkor delšího zpracování operací zprostředkovávajících zápis z důvodu blokace vlákna požadavku během synchronizace databází.

Literatura

[1] S. Randive, "Why product development and design needs cohesion-coupling?" 2022, accessed 24.12.2022. [Online]. Available: https://www.haptik.ai/tech/why-product-development-and-design-needs-cohesion-coupling

[2] A. Suschevich, "The advantages of microservices vs monolithic architectures," 2022, accessed 24.12.2022. [Online]. Available: https://levelup.gitconnected.com/the-advantages-of-microservices-vs-monolithic-architectures-94ce25ae3fd

[3] The Code Reaper, "Layered architecture pattern in software engineering," 2020, accessed 24.12.2022. [Online]. Available: https://thecodereaper.com/2020/08/22/layered-architecture-pattern-in-software-engineering

[4] Altexsoft, "Event-driven architecture and pub/sub pattern explained," 2021, accessed 24.12.2022. [Online]. Available: https://www.altexsoft.com/blog/event-driven-architecture-pub-sub

[5] Maveric Systems, "Microservices i - microservices vs soa," 2018, accessed 24.12.2022. [Online]. Available: https://maveric-systems.com/blog/microservices-i-microservices-vs-soa

[6] Wallarm, "The concept of an api gateway," 2021, accessed 24.12.2022. [Online]. Available: https://www.wallarm.com/what/the-concept-of-an-api-gateway

[7] S. Kappagantula, "Everything you need to know about microservices design patterns," 2022, accessed 24.12.2022. [Online]. Available: https://www.edureka.co/blog/microservices-design-patterns

[8] M. Ozkaya, "Saga pattern for microservices distributed transactions," 2021, accessed 24.12.2022. [Online]. Available: https://medium.com/design-microservices-architecture-with-patterns/saga-pattern-for-microservices-distributed-transactions-7e95d0613345

[9] L. Singla, "What's a software design pattern? (+7 most popular patterns)," 2022, accessed 24.12.2022. [Online]. Available: https://www.netsolutions.com/insights/software-design-pattern

[10] Asper Brothers, "Effective application scaling. proven techniques and methods," 2022, accessed 24.12.2022. [Online]. Available: https://asperbrothers.com/blog/scaling-applications

[11] M. Tanner, "What is change data capture? everything you need to know," 2022, accessed 4.5.2023. [Online]. Available: https://www.arcion.io/learn/change-data-capture

[12] Red Hat, "What is an application architecture?" 2020, accessed 24.12.2022. [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture

[13] Lithmee, "Difference between peer to peer and client server network," 2018, accessed 24.12.2022. [Online]. Available: https://pediaa.com/difference-between-peer-to-peer-and-client-server-network

[14] Shivang, "Monolithic architecture - explained in simple words," accessed 24.12.2022. [Online]. Available: https://scaleyourapp.com/monolithic-architecture

[15] IBM, "What is an esb?" accessed 1.5.2023. [Online]. Available: https://www.ibm.com/topics/esb

[16] K. Clark, "The fate of the esb," 2018, accessed 1.5.2023. [Online]. Available: https://developer.ibm.com/articles/cl-lightweight-integration-1

[17] IBM, "What is soa?" accessed 1.5.2023. [Online]. Available: https://www.ibm.com/topics/soa

[18] IBM Cloud Team, "Soa vs microservices: What's the difference?" 2021, accessed 1.5.2023. [Online]. Available: https://www.ibm.com/cloud/blog/soa-vs-microservices

[19] G. Alvarenga, "Soa vs microservices: What's the difference?" 2022, accessed 24.12.2022. [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/cloud-security/soa-vs-microservices

[20] K. Arsov, "Microservices vs. soa - is there any difference at all?" 2017, accessed 24.12.2022. [Online]. Available: https://medium.com/microtica/microservices-vs-soa-is-there-any-difference-at-all-2a1e3b66e1be

[21] Samarpit, "Microservices vs soa: What's the difference," 2020, accessed 24.12.2022. [Online]. Available: https://www.edureka.co/blog/microservices-vs-soa

[22] Red Hat, "What is event-driven architecture?" 2019, accessed 24.12.2022. [Online]. Available: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture

[23] J. Nemer, "Advantages and disadvantages of microservices architecture," 2019, accessed 24.12.2022. [Online]. Available: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback

[24] Middleware, "What are microservices? how microservices architecture works," 2022, accessed 24.12.2022. [Online]. Available: https://middleware.io/blog/microservices-architecture

[25] S. S. Mike Loukides, "Microservices adoption in 2020," 2020, accessed 24.12.2022. [Online]. Available: https://www.oreilly.com/radar/microservices-adoption-in-2020

[26] G. Plantrou, "Microservices are becoming the default application architecture choice - is it time to jump in?" 2022, accessed 24.12.2022. [Online]. Available: https://blog.scaleway.com/microservices-are-becoming-the-default-application-architecture-choice-is-it-time-to-jump-in

[27] M. Kolny, "Scaling up the prime video audio/video monitoring service and reducing costs by 90%," 2023, accessed 13.5.2023. [Online]. Available: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90

[28] M. Udantha, "Microservice architecture and design patterns for microservices," 2019, accessed 24.12.2022. [Online]. Available: https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a

[29] J. Wade, "Brownfield vs. greenfield development: What's the difference in software?" 2018, accessed 4.5.2023. [Online]. Available: https://synoptek.com/insights/it-blogs/greenfield-vs-brownfield-software-development

[30] Microsoft, "Anti-corruption layer pattern," accessed 4.5.2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer

[31] B. Crema, "Orchestration pattern," 2021, accessed 4.5.2023. [Online]. Available: https://medium.com/gbtech/orchestration-pattern-3d8f5abc3be3

[32] M. Ozkaya, "Materialized view pattern," 2021, accessed 4.5.2023. [Online]. Available: https://medium.com/design-microservices-architecture-with-patterns/materialized-view-pattern-f29ea249f8f8

[33] Confluent, "Confluent platform licenses," accessed 24.12.2022. [Online]. Available: https://docs.confluent.io/platform/current/installation/license.html

[34] ——, "Schema registry overview," accessed 24.12.2022. [Online]. Available: https://docs.confluent.io/platform/current/schema-registry/index.html

[35] ——, "Kafka connect," accessed 24.12.2022. [Online]. Available: https://docs.confluent.io/platform/current/connect/index.html

[36] Baeldung, “Introduction to debezium,” 2021, accessed 24.12.2022. [Online]. Available: https://www.baeldung.com/debezium-intro

[37] Debezium, “Debezium,” 2021, accessed 6.5.2023. [Online]. Available: https://debezium.io

Příloha A

Přiložené soubory

Přiložené soubory:

`app` - zdrojové kódy implementované příkladové knihovny

`tests` - soubory obsahující příkazy použité při testování a jejich výstupy

`text/bachelor.pdf` - text práce ve formátu PDF

`text/latex` - zdrojové soubory dokumentu ve formátu LaTeX

Zdrojové kódy příkladové knihonvy jsou také volně přístupné prostřednictvím služby GitHub - `https://github.com/palivtom/cqrs-materializedview-combo-pattern/commit/c9f1208469422b60038ccb75d2fa587e4aaf4daf` - na tomto odkazu je zachycen stav repozitáře v době odevzdání práce.