F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

# Event Driven Architektura

**Victor Remel**
Otevřená Informatika, Software

02 2023, 05 2023
Vedoucí práce: Ing. Martin Komárek

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Remel** Jméno: **Victor** Osobní číslo: **502498**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Event Driven Architektura**

Název bakalářské práce anglicky:

**Event Driven Architecture**

Pokyny pro vypracování:

Seznamte se s Event Driven Architecture[2] a souvisejícími technologiemi/vzory, jako jsou Event Sourcing, Event Storming, Transactional Log Tailing, Apache Kafka a RabbitMQ. Konkrétně:
1. Definujte a popište Event Driven Architecture a její výhody oproti tradičním architekturám.
2. Detailně se seznamte s Event Sourcingem, Event Stormingem a Transactional Log Tailingem. Popište jejich výhody a omezení a ukažte příklady aplikací.
3. Implementujte Transactional Log Tailing nad ukázkovou SQL DB.
4. Představte technologie Apache Kafka a RabbitMQ, popište rozdíly mezi nimi, ukažte situace, kdy je vhodné použít jedno nebo druhé řešení.
5. Implementujte a porovnejte v praxi několik ukázek aplikací s použitím Apache Kafka a RabbitMQ. Zhodnoťte jejich výhody a nevýhody.
6. Aplikujte získané znalosti a doplňte do existující ukázkové cloud-native aplikace “Rezervace jízdenek”[1] novou mikroslužbu pro notifikaci uživatelů.
Při vývoji používejte relevantní mikroservisní vzory a postupujte iterativně. Vše průběžné testujte, nasazujte na Value Stream Delivery Platformu CodeNOW[3] a dokumentujte.

Seznam doporučené literatury:

[1] Vávra Robin. Vývoj cloud native aplikací v praxi. B.S. thesis, České vysoké učení technické v Praze. Výpočetní a informační centrum., 2022. Dostupné z: http://hdl.handle.net/10467/101026.
[2] Chris Richardson. Microservices patterns: with examples in Java. Manning Publications, 2018. Dostupné z: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/. [3] CodeNOW. CodeNOW Documentation, 2022. Dostupné z: https://docs.codenow.com/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Martin Komárek kabinet výuky informatiky FEL**

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

Datum zadání bakalářské práce: **10.02.2023** Termín odevzdání bakalářské práce: _______________

Platnost zadání bakalářské práce: **22.09.2024**

Ing. Martin Komárek
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

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

## III. PŘEVZETÍ ZADÁNÍ

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

.

Datum převzetí zadání

Podpis studenta

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

# Poděkování / Prohlášení

Chtěl bych poděkovat svému vedoucímu práce panu Ing. Martinu Komárkovi za jeho pravidelné konzultace a za zajímavé téma bakalářské práce.

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

V Praze dne 22. 05. 2023

........................................

# Abstrakt / Abstract

Cílem této práce je představit architekturu řízenou událostmi, způsob její implementace a technologie, které se při implementaci architektury řízené událostmi nejčastěji používají. Představit vzory, jako je Transaction Log Tailing, Event Sourcing. Ukázat implementaci Transaction Log Tailing. Porovnat dva message brokery RabbitMQ a Apache Kafka a ukázat příklady jejich implementace. Ukázat, jak implementovat vlastní službu pro odesílání e-mailů do existující aplikace CodeNow. Všechny kódy budou napsány v programovacím jazyce Java s využitím frameworku SpringBoot.

**Klíčová slova:** Event driven architecture, Microservice, Apache Kafka, Mailgun, CodeNow, RabbitMQ, Debezium, Docker, Asynchronní komunikace

The main goal of this work is to introduce event-driven architecture, how it is implemented and the technologies that are most commonly used in implementing event-driven architecture. To introduce patterns such as Transaction Log Tailing, Event Sourcing. Demonstrate the implementation of Transaction Log Tailing. Compare two message brokers RabbitMQ and Apache Kafka and show examples of their implementation. Demonstrate how to implement a custom email sending service into an existing CodeNow application. All codes will be written in Java programming language using SpringBoot framework.

**Keywords:** Event driven architecture, Microservice, Apache Kafka, Mailgun, CodeNow, RabbitMQ, Debezium, Docker, Asynchronous communication

# Obsah /


**1 Úvod** **1**

**2 Event Driven Architecture** **2**

2.1 Event Driven Architecture . . . . 2

2.1.1 Hlavní princip architektury řízené událostmi . . . 2

2.1.2 Typy přenosu událostí v architektuře řízené událostmi . . . . . . . . . . . 2

2.1.3 Výhody a nevýhody architektury řízené událostmi . . . . . . . . . . . . . 3

2.2 Výhody a nevýhody oproti tradičním architekturám. . . . . 3

2.2.1 Výhody architektury řízené událostmi oproti monolitické architektuře . . . 4

2.2.2 Nevýhody architektury řízené událostmi oproti monolitické architektuře . . . 4

2.2.3 Výhody a nevýhody architektury řízené událostmi oproti architektuře mikroslužeb . . . . . . . 4

2.3 Použití patternů při projektování architektury řízené událostmi . . . . . . . . . . . . . 5

2.3.1 Patterny používané v architektuře řízené událostmi . . . . . . . . . . . 5

**3 Event Sourcing, Event Storming, Transaction Log Tailing** **6**

3.1 Event Sourcing . . . . . . . . . . 6

3.1.1 Výhody Event Sourcingu . . 6

3.1.2 Nevýhody Event Sourcingu . 6

3.2 Event Storming . . . . . . . . . . 6

3.2.1 Výhody Event Stormingu . . 7

3.2.2 Nevýhody Event Stormingu . 7

3.3 Transactional Outbox, Polling Publisher, Transaction Log Tailing . . . . . . . . . . . . 7

3.3.1 Polling Publisher . . . . . . . 8

3.3.2 Transaction Log Tailing . . . 8

3.4 Implementace Transaction Log Tailing . . . . . . . . . . . . 9

3.4.1 Konfigurace před spuštěním aplikace . . . . . . . . 9

3.4.2 Implementace komponenty Rezervace a ukládání dat do Postgres. . 12

3.4.3 Závěr o Transaction Log Tailing . . . . . . . . . 17

**4 Porovnání Apache Kafka a RabbitMQ** **18**

4.1 Message broker . . . . . . . . . 18

4.1.1 Jak funguje message broker . . . . . . . . . . . 18

4.2 Apache Kafka . . . . . . . . . . 18

4.2.1 Jak funguje Apache Kafka 19

4.3 Ukazka Apache Kafka code . . 20

4.3.1 Testování Apache Kafka . . 23

4.4 RabbitMQ . . . . . . . . . . . 24

4.4.1 Jak funguje RabbitMQ . . 25

4.5 Ukazka RabbitMQ code . . . . 25

4.5.1 Jednoduchá komunikace . . 25

4.5.2 Spolehlivé publikování . . . 28

4.5.3 Testování RabbitMQ . . . 31

4.6 Porovnání Apache Kafka a RabbitMQ . . . . . . . . . . . 32

4.6.1 V jakých případech je lepší použít Apache Kafka a v jakých případech RabbitMQ . . . . . 32

4.6.2 Výhody a nevýhody Apache Kafka . . . . . . . 33

4.6.3 Výhody a nevýhody RabbitMQ . . . . . . . . . 33

4.7 Závěry . . . . . . . . . . . . . . 34

**5 Závěr** **35**

**Literatura** **37**

**A Přiložené soubory** **41**

# / Obrázky


**3.1** Konečný výsledek Event Storming .......................... 7
**3.2** TransactionLogMining ........... 9
**3.3** dockerCompose1 ............... 10
**3.4** dockerCompose2 ............... 10
**3.5** schemaSQL .................... 11
**3.6** StartDebeziumConnector ...... 11
**3.7** debeziumJSON1 ................ 11
**3.8** debeziumJSON2 ................ 12
**3.9** ResultDebeziumConnector ..... 12
**3.10** propertiesForProducentTLT ... 13
**3.11** propertiesNotification .......... 13
**3.12** NewReservationsParameters ... 14
**3.13** SaveCreatedReservation ....... 14
**3.14** SaveEventToOutbox ........... 15
**3.15** SaveReservationLog ............ 15
**3.16** GetReservationEventLog ...... 15
**3.17** HandleMessage ................. 15
**3.18** ConsumeMessage ............... 16
**3.19** MailgunMessage ................ 16
**3.20** MailgunGoogle ................. 16
**4.1** Diagram fungování služby Apache Kafka .................. 19
**4.2** Ukládání zpráv v Apache Kafka ............................. 19
**4.3** ChooseApiBackend ............. 20
**4.4** SendResponseAPI .............. 21
**4.5** SeeLogs ......................... 21
**4.6** MailSendLogCodeNow ......... 21
**4.7** MailNotificationConfig ......... 22
**4.8** MailConsumer .................. 22
**4.9** SendEmail ...................... 23
**4.10** TestKafkaPart1 ................ 24
**4.11** TestKafkaPart2 ................ 24
**4.12** TestKafkaResult ............... 24
**4.13** Diagram Producent-Queue-Consumer RabbitMQ .......... 25
**4.14** MessagingConfigForRabbitMQ ......................... 26
**4.15** RabbitMQProducer ............ 27
**4.16** SendDataPostman ............. 27
**4.17** MessageInRabbitMQ ........... 27
**4.18** ConsumerRabbitMQ ........... 28
**4.19** ConsumerRabbitMQOutPut ... 28
**4.20** RPCRabbitMQ ................. 28
**4.21** RPCProducerConf1 ............ 29

**4.22** RPCProducerConf2 . . . . . . . . . . . . 29
**4.23** RPCProducer . . . . . . . . . . . . . . . . . . 30
**4.24** RPCProducerOut . . . . . . . . . . . . . . 30
**4.25** RPCConsumerOut . . . . . . . . . . . . . 30
**4.26** RPCConsumer . . . . . . . . . . . . . . . . . 31
**4.27** TestRabbtiMQ . . . . . . . . . . . . . . . . . 31
**4.28** TestRabbtiMQResult . . . . . . . . . . 32

# Kapitola 1
# Úvod

Stále častěji se setkáváme s aplikacemi, které musí být schopny zpracovávat velké množství dat, pracovat s vysokým výkonem a být spolehlivé. V takovém případě je použití architektury řízené událostmi velmi výhodné [1]. Architektura řízená událostmi je architektonický přístup založený na zpracování a komunikaci událostí v systému. Díky tomuto přístupu může systém reagovat na události, které se v systému vyskytují, a rozhodovat se na základě typu událostí.

Cílem této práce je prozkoumat architekturu řízenou událostmi. Budou diskutovány výhody a nevýhody této architektury ve srovnání s klasickými architekturami, jako je architektura mikroslužeb a monolitická architektura. Probereme také vzory, pomocí nichž lze architekturu řízenou událostmi implementovat.

Nedílnou součástí architektury řízené událostmi je asynchronní komunikace a s ní i message brokery. Proto budou podrobně rozebrány a vzájemně porovnány dva message brokery RabbitMQ a Apache Kafka.

Závěrečnou částí bude implementace vlastní služby, která bude uživatele informovat o nové rezervaci. Služba bude napojena na stávající službu na platformě CodeNow.

Studium architektury řízení událostí je důležité pro vytváření škálovatelných, efektivních a spolehlivých systémů, které jsou schopny zpracovávat velké množství dat. Pochopení základního principu architektury řízené událostmi pomůže vývojářům vybrat pro jejich aplikace nejvhodnější architekturu a nástroje.

# Kapitola 2
# Event Driven Architecture

V této kapitole se seznámíme s architekturou řízenou událostmi, vysvětlíme její výhody a nevýhody a její výhody oproti tradičním architekturám.

## 2.1 Event Driven Architecture

**Architektura řízená událostmi (Even Driven Architecture)** je softwarová architektura, v rámci které jednotlivé aplikace nebo nezávislé části jedné aplikace komunikují mezi sebou prostřednictvím odesílání a přijímání událostí. Architektura řízená událostmi umožňuje reagovat na změny v aplikaci v reálném čase [2].

Taková architektura se skládá z producenta událostí a konzumenta událostí. Producent detekuje událost a prezentuje ji jako zprávu. Producent nezná konzumenta ani důsledky události. Jakmile je událost detekována, producent ji *asynchronně* (viz. 4.1) předá konzumentovi. Konzument může událost zpracovat nebo změnit své chování v závislosti na události.

Architektura řízená událostmi maximalizuje potenciál cloudových aplikací a umožňuje využívat výkonné aplikační technologie, jako je například analýza v reálném čase [3].

### 2.1.1 Hlavní princip architektury řízené událostmi

V takové architektuře jsou jednotlivé části aplikace často producenty a konzumenty. Producent předává událost prostřednictvím *message brokera*(viz. 4.1) ve formě zprávy, kde je zachováno chronologické pořadí událostí. V reálném čase nebo kdykoli jindy konzument přijme událost a zpracuje ji za účelem změny chování programu nebo spuštění jiné události.

### 2.1.2 Typy přenosu událostí v architektuře řízené událostmi

V architektuře řízené událostmi existují dva základní modely přenosu událostí [4].

**Event messaging or publish/subscribe.** Konzumenti se přihlásí k odběru jedné nebo více tříd zpráv poskytovaných producenty. Po přečtení budou události smazány.

**Přenos událostí**. V modelu streamování událostí poskytovatelé událostí publikují streamy událostí brokerovi. Odběratelé událostí se k těmto proudům *přihlásí*[1], ale namísto přijímání a konzumace každé události v okamžiku jejího zveřejnění se mohou odběratelé kdykoli připojit ke každému proudu a přijímat pouze ty události, které chtějí

[1] U dvou (RedHat, Microsoft) ze tří zdrojů je napsáno, že se konzument nepřihlašuje k odběru proudu událostí, ale u třetího zdroje (IBM) je napsáno, že se k odběru proudu událostí přihlašuje. 1. zdroj:https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture, 2.zdroj: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven, 3.zdroj: https://www.ibm.com/topics/event-driven-architecture

přijímat. Klíčový rozdíl spočívá v tom, že události jsou uchovávány zprostředkovatelem i poté, co je odběratelé přijali.

V dalších kapitolách se seznámíme s message brokery, jako jsou Apache Kafka (viz. 4.2) a RabbitMQ (viz. 4.4), probereme jejich výhody a nevýhody a ukážeme si, jak fungují v praxi.

### 2.1.3 Výhody a nevýhody architektury řízené událostmi

Hlavní výhody použití architektury řízené událostmi jsou následující [5]:

- **Nahrazení dotazování a webhooků událostmi.** Architektury řízené událostmi nahrazují neefektivní metody komunikace, jako je dotazování a zpětná volání, komunikací mezi komponentami pomocí událostí, čímž snižují latenci, zjednodušují integraci a škálování[2], usnadňují návrh systému téměř v reálném čase a odstraňují zpracování dat v dávkách. Tato architektura umožňuje, aby každá mikroslužba zpracovávala v daném okamžiku pouze jednu událost a škálovala se zvyšováním počtu spuštěných instancí.
- **Snížení složitosti.** Mikroslužby umožňují rozdělit složité podnikové procesy na nezávislé komponenty, které spolu komunikují asynchronně pomocí událostí. To usnadňuje implementaci a údržbu byznys procesů, integraci služeb s různou rychlostí zpracování dat a efektivní zpracování velkých objemů informací, například přijímání objednávek a zpracování plateb v internetovém obchodě.
- **Zlepšení škálovatelnosti a rozšiřitelnosti.** Mikroslužby odesílají vzniklé události do služeb pro zasílání zpráv, což umožňuje škálovatelnost. Aplikace založené na architektuře řízené událostmi se snadněji škálují a rozšiřují, protože vývojáři mohou přidávat vlastnosti a funkce, aniž by ovlivnili již existující mikroslužby.

Hlavní nevýhody použití architektury řízené událostmi jsou následující [7]:

- **Duplicitní události.** Bez vhodného plánování a sladění může jediná událost vyvolat několik duplicitních zpráv v různých službách, čímž vzniknou mezery v komunikaci.
- **Zmatek v pojmenování.** Při manipulaci s mnoha publikujícími a odběrateli je snadné mít duplicitní názvy nebo chyby v pojmenování, zejména v případě více týmů, kteří spolu nepravidelně komunikují.
- **Nedostatek jasného pořadí pracovního postupu.** Každý krok v rámci pracovního postupu je spuštěn, když je služba nebo broker upozorněn na konkrétní událost. Pokud přijímající komponenta není správně nastavena pro reakci na událost, vzniknou chyby a způsobí se kaskádové problémy v pracovním postupu.
- **Řešení chyb.** Aplikace může obsahovat obrovské množství message brokerů a služeb, vyhledávání chyb v takových aplikacích je časově náročné. Proto taková architektura vyžaduje komplexní sadu monitorovacích nástrojů a technik, aby vývojáři jasně viděli, jak aplikace funguje, a aby vyhledávání chyb trvalo co nejméně času.

## 2.2 Výhody a nevýhody oproti tradičním architekturám.

Nejprve se podíváme na definici monolitické a mikroservisní architektury.

[2] *Škálovatelnost* je schopnost systému zvýšit svůj výkon, když se zvýší zatížení aplikace. Další podrobnosti naleznete zde [6].

**Monolitická architektura** je tradiční model softwarového programu, který je vytvořen jako jednotný celek, který je samostatný a nezávislý na ostatních aplikacích [8].

**Architektura mikroslužeb** je typ aplikační architektury, kdy je aplikace vyvíjena jako soubor služeb. Poskytuje rámec pro nezávislý vývoj, nasazení a údržbu diagramů architektury mikroslužeb a jejích služeb [9].

### 2.2.1 Výhody architektury řízené událostmi oproti monolitické architektuře

- **Škálovatelnost.** Události v architektuře řízené událostmi jsou obvykle publikovány do služeb pro zasílání zpráv, které fungují jako pružná vyrovnávací paměť mezi mikroslužbami a pomáhají řešit škálovatelnost. Události lze také odesílat do směrovací služby, která může filtrovat a směrovat zprávy na základě obsahu události. Výsledkem je, že aplikace založené na událostech mohou být škálovatelnější a poskytují větší redundanci než monolitické aplikace [10].
- **Rychlé zapojení do projektu .** Architektura řízená událostmi podporuje dekompozici monolitických systémů na menší doménové modely. Vývojáři se tak mohou dostat do tempa s menší kognitivní zátěží a rychleji se stávají produktivními. Když jsou kritické funkce odděleny, je také menší riziko při nasazování aktualizací a nových funkcí [10].

### 2.2.2 Nevýhody architektury řízené událostmi oproti monolitické architektuře

- **Variabilní latence.** Na rozdíl od monolitických aplikací, které mohou zpracovávat vše v rámci stejné paměťové oblasti na jednom zařízení, událostmi řízené aplikace komunikují přes sítě. Tento návrh přináší variabilní latenci. I když monolitické aplikace mohou mít nižší nebo méně variabilní latenci, obecně to přichází na úkor škálovatelnosti a dostupnosti [10].
- **Ladění.** Ladění systémů řízených událostmi se liší od ladění monolitických aplikací. Při použití mikroslužeb a přenosu událostí mezi různými systémy je obtížné zaznamenat a reprodukovat přesný stav více služeb při výskytu chyby, kvůli odděleným logovacím souborům pro každou službu a volání funkce [10].

Architektura řízená událostmi by měla být použita, pokud aplikace vyžaduje vysokou odolnost proti chybám, škálovatelnost a odezvu aplikace v reálném čase. Vývoj aplikace pomocí takové architektury však komplikuje proces vývoje a vyžaduje zkušené vývojáře.

Monolitická architektura se používá při vývoji malé aplikace s omezenou funkčností, kterou lze rychle nasadit. Je důležité si uvědomit, že monolitická architektura se obtížně škáluje a těžko reaguje na zásadní změny.

### 2.2.3 Výhody a nevýhody architektury řízené událostmi oproti architektuře mikroslužeb

Událostmi řízená aplikace je založena na architektuře mikroslužeb. Proto nelze diskutovat o výhodách či nevýhodách architektury řízené událostmi oproti architektuře mikroslužeb. Můžeme však říci, že aplikace využívající architekturu řízenou událostmi má více výhod než architektura mikroslužeb neimplementující architekturu řízenou událostmi, kdy je nutné neustále reagovat na změny v aplikaci [11].

## 2.3 Použití patternů při projektování architektury řízené událostmi

Používání *vzorů* [3] při návrhu architektury řízené událostmi je dobrou praxí a má mnoho výhod.

- Vzory zjednodušuje vývoj, protože určité vzory poskytují určitou strukturu. Takovou strukturu je snazší implementovat, než ji od začátku vymýšlet samostatně. Použitím vzorů se také zkracuje doba návrhu.
- Vzory pomáhají minimalizovat rizika, která mohou vzniknout během vývoje, protože vzory byly vyzkoušeny lidmi a časem.
- Vzory dělají aplikaci srozumitelnější a strukturovanější, čímž pomáhají vývojářům snížit složitost aplikace.

### 2.3.1 Patterny používané v architektuře řízené událostmi

Následující seznam patternů je převzat ze stránek IBM [12]:

- **Event Sourcing.** Tento vzor ukládá všechny změny stavu jako události. To pomáhá sledovat historii změn stavu aplikace, protože obvyklá *databáze* [4] ukládá pouze poslední aktualizovaný stav. Tento pattern detailněji popíšu později (viz. 3.1).
- **Transactional Outbox.** Tento vzor zajišťuje, že události nebo zprávy v rámci jedné databázové transakce[5] jsou uloženy v databázi v tabulce Outbox předtím, než jsou přeneseny do message brokeru. To je užitečné, když je třeba dodržovat automatické akce v databázi [15].
- **Saga.** Tento vzor rozděluje jednu velkou transakci na několik menších transakcí, z nichž každá se týká konkrétní služby. Pokud jedna z lokálních transakcí není úspěšně dokončena, mohou být zrušeny všechny ostatní lokální transakce [16].
- **CQRS (Command Query Responsibility Segregation).** Tento pattern slouží k vytvoření oddělených částí aplikace pro čtení z databáze a pro zápis dat do databáze. To pomáhá vyhnout se vysokému zatížení databáze. Také lze snadno optimalizovat stranu čtení systému odděleně od strany zápisu, což umožňuje škálovat každou z nich jinak podle zatížení dané strany [17].

[3] Vzory jsou osvědčeným způsobem řešení častých problémů při návrhu a programování.
[4] Databáze je organizovaná kolekce strukturovaných dat, která jsou obvykle uložena a používána s pomocí specializovaného softwaru [13].
[5] Transakce je související sekvence operací s databází, která musí být provedena jako celek [14].

# Kapitola 3
# Event Sourcing, Event Storming, Transaction Log Tailing

V této kapitole se blíže podíváme na Event Sourcing. Projedeme si, co je Event Storming a Transaction Log Tailing a podíváme se na implementaci Transaction Log Tailing.

## 3.1 Event Sourcing

**Event Sourcing.** Tento pattern ukládá všechny změny stavu aplikace jako události. Tyto události představují fakta o tom, co se v systému stalo v minulosti. To pomáhá sledovat historii změn stavu aplikace, protože obvyklá databáze ukládá pouze poslední aktualizovaný stav. Když aplikace obdrží příkaz od uživatele nebo ze systému, je v aplikaci bude vygenerována událost popisující danou akci, která se uloží do databáze.

### 3.1.1 Výhody Event Sourcingu

Hlavní výhody event sourcingu jsou následující [15]:

- U každé události lze uložit ID uživatele, který změnu provedl, což umožňuje vést zaručeně správný záznam všech událostí, které nastaly.
- Protože jsou všechny změny stavu aplikace ukládány v pořadí, je vždy možné vrátit aplikaci do určitého stavu. To je užitečné pro opravu chyb.

### 3.1.2 Nevýhody Event Sourcingu

Hlavní nevýhody event sourcingu jsou následující [15]:

- Integrace event sourcingu do stávající aplikace často vyžaduje přepsání business logiky této aplikace.
- Implementace event Sourcingu může být složitá, vzhledem ke specifické správě událostí. Například jak uspořádané ukládání událostí do databáze.
- U aplikací založených na zprávách může být event sourcing problémem, pokud zpracovatelé událostí znovu přijímají a zpracovávají událost, což vede k duplikaci nebo nežádoucím efektům. To lze vyřešit pomocí monotónně rostoucího identifikátoru pro každou událost.

## 3.2 Event Storming

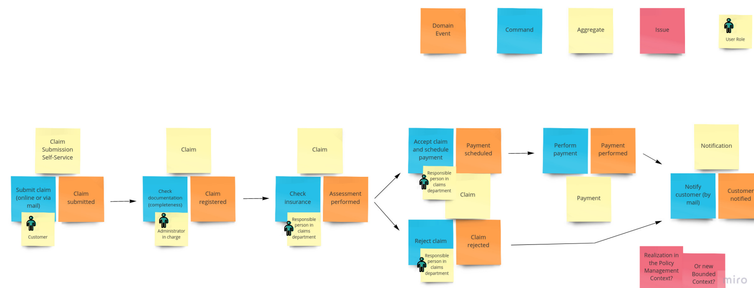
**Event Storming** je formou schůzky, na které se setkávají odborníci na danou problematiku, aby společně modelovali a navrhovali aplikaci. Hlavním cílem Event Stormingu je rychle vytvořit model domény. Event Storming je rozdělen do tří fází.

První fáze spočívá v intenzivním zvýraznění událostí, které se vyskytují v rámci byznys procesů.

Ve druhé fázi odborníci identifikují příčiny jednotlivých událostí. Příčinou může být: uživatelská akce, vnější systém, jiná doménová událost, uplynutí času. Akce uživatele je příkaz.

Třetí fáze spočívá v odhalení jednotek, které spotřebovávají příkazy a generují události.

Výsledek Event Stormingu je znázorněn na obrázku (viz. Obrázek 3.1)




**Obrázek 3.1.** Event Storming Results in Context [18]

### 3.2.1 Výhody Event Stormingu

Hlavní výhody event stormingu jsou následující [19]:

- Event Storming zkracuje dobu potřebnou k vytvoření komplexního modelu podnikové domény. Pomocí této techniky můžete zmapovat podnikový proces během několika hodin během jediného workshopu.
- Místo použití složitých diagramů UML rozvádí Event Storming proces do jednoduchých pojmů, kterým rozumí jak technické, tak netechnické zainteresované strany.
- Event storming je poutavý a zábavný způsob vytváření modelu domény. Díky praktickému přístupu je každý účastník vyzván, aby se podělil o své znalosti domény. To vede ke skvělé interakci a výsledkem je úplnější pochopení procesu.

### 3.2.2 Nevýhody Event Stormingu

Hlavní nevýhody event stormingu jsou následující [20]:

- Jelikož je event storming méně strukturovaný než tradiční brainstorming, je také možné, že se členové týmu nesoustředí na správné problémy.
- Jelikož je extrémní brainstorming méně strukturovaný než tradiční, je pravděpodobné, že bude časově náročnější. To může být problém, pokud jste pod časovým tlakem.

## 3.3 Transactional Outbox, Polling Publisher, Transaction Log Tailing

Předtím, než se podíváme na Transaction Log Tailing, se seznámíme se patterny Transactional Outbox a Polling Publisher.

Služby často musí publikovat zprávy jako součást transakce, která aktualizuje databázi. Tyto dvě operace: odeslání zprávy a aktualizace databáze probíhají v rámci jedné transakce a musí být provedeny atomicky, jinak existuje vysoká pravděpodobnost nekonzistentního systému, a tedy nesprávného fungování celé aplikace.

Toto řešení se nazývá transakční řešení, které využívá transakce zahrnující databázi a message brokery.

Transakční řešení někdy nevyhovuje většině moderních aplikací a není podporováno v mnoha moderních brokerech, jako je například Apache Kafka(viz. 4.2). Místo ***transakčního řešení***[1] můžeme použít Transactional Outbox. O Transactional Outbox jsem už dříve psal (viz. 2.3.1). A jedním z problému tohoto vzoru je nalezení objektů obsahujících události a jejich publikování do message brokerů. Tento problém lze vyřešit pomocí Polling Publisher nebo druhého složitějšího patternu Transaction Log Tailing.

### 3.3.1 Polling Publisher

**Polling Publisher** je vzor, který slouží k dotazování tabulky OUTBOX publisherem. Publisher pravidelně vybírá nepublikované záznamy z tabulky. Poté je odešle do message brokeru a nakonec je z tabulky odstraní.

Tento přístup je samozřejmě jednoduchý na implementaci a použití Polling Publisher umožňuje řídit zatížení systému, takže vývojáři mohou nastavit časové intervaly, kdy publisher kontroluje záznamy z tabulky.

Nevýhodou je, že tento přístup lze použít pouze u databází ***NoSQL***[2]. Aplikace se totiž může dotazovat na obchodní objekty místo na tabulku OUTBOX a dotazování na obchodní objekty často není efektivní [15].

Místo Polling Publisher můžeme použít složitější, ale efektivnější přístup Transaction Log Tailing.

### 3.3.2 Transaction Log Tailing

**Transaction Log Tailing** je vzor, který publikuje změny v databázi na základě sledovaných událostí v transakčním logu. Každá zaznamenaná aktualizace provedená aplikací je uvedena jako záznam v transakčním logu. Transaction Log Miner tento log čte a publikuje zprávu do Apache Kafka.

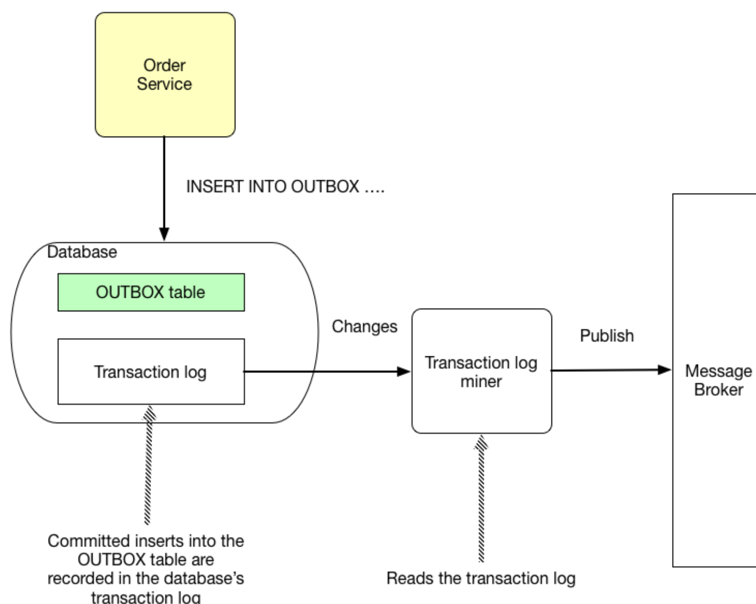
Transaction Log Tailing je vhodný pro publikování databázových zpráv založených na SQL a NoSQL.

Hlavním problémem tohoto přístupu je, že vyžaduje určité úsilí vývojářů, například napsání nízkoúrovňového kódu pro volání rozhraní API databáze nebo použití Debezium pro publikování změn do Apache Kafka z databáze [15].

Schéma Transaction Log Tailing na Obrázku 3.2:

[1] Transakční řešení je přístup, který zaručuje atomicitu(Atomicity), konzistenci(Consistency), izolaci(Isolation) a trvanlivost(Durability) (ACID).

[2] NoSQL je přístup k ukládání a zpracování dat, která jsou uložena jako dokumenty, grafy a klíče-hodnoty.




**Obrázek 3.2.** Pattern: Transaction log tailing [21]

## 3.4 Implementace Transaction Log Tailing

Transaction Log Tailing implementuji pomocí nástroje ***Debezium*** [22], který slouží jako nástroj pro realizaci *Change Data Capture* [23].

Moje služba `Rezervace` používá dva *koncové body*[3] pro „novou registraci" a pro „získání seznamu rezervací". Používám databázi *Postgres* [24] se dvěma tabulkami reservation a msg_outbox. Dále používám službu `sendNotification`. Tato služba je konzumentem. Přijímá zprávy ze služby Kafka, získá z nich e-mail, na který bude odeslán nový e-mail týkající se rezervace. E-mail bude odeslán prostřednictvím služby *Mailgun* [25].

Používám také Kafka Connect a Debezium. Kafka Connect slouží jako konektor pro připojení Apache Kafka k různým databázím. K sledování změn v databázi používám Debezium.

Debezium přebírá záznamy z tabulky msg_outbox a publikuje události podle témat.

Change Data Capture umožňuje sledovat změny v databázi analýzou protokolu změn v databázi. To nám pomáhá reagovat na změny v reálném čase.

Používám také *Docker* [26], ve kterém běží databáze Postgres, message broker Apache Kafka, *Zookeeper* [27] a Debezium.

### 3.4.1 Konfigurace před spuštěním aplikace

Před spuštěním aplikace musíme nastavit Docker, který spustí všechny potřebné komponenty.

[3] Koncový bod je unikátní adresa, prostřednictvím které lze v síti přistupovat ke službě.

```yaml
services:
  postgres:
    image: postgres:12
    ports:
      - 5432:5432
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./docker/postgres/0.schema.sql:/docker-entrypoint-initdb.d/0.schema.sql
    environment:
      POSTGRES_PASSWORD: changeit
    command: postgres -c wal_level=logical

  zookeeper:
    image: confluentinc/cp-zookeeper:5.5.1
    volumes:
      - zookeeper_data:/var/lib/zookeeper/data
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      KAFKA_HEAP_OPTS: -Xmx64m

  kafka:
    image: confluentinc/cp-kafka:5.5.1
    ports:
      - 9092:9092
    volumes:
      - kafka_data:/var/lib/kafka/data
    environment:
      KAFKA_LISTENERS: LC://kafka:29092,LX://kafka:9092
      KAFKA_ADVERTISED_LISTENERS: LC://kafka:29092,LX://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: LC:PLAINTEXT,LX:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: LC
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_HEAP_OPTS: -Xmx192m
    depends_on:
      - zookeeper
```

**Obrázek 3.3.** Příklad toho, jak může vypadat docker-compose. Část 1.

```yaml
  debezium-connect:
    image: debezium/connect:1.3.0.Final
    ports:
      - 8083:8083
    environment:
      GROUP_ID: 1
      CONFIG_STORAGE_TOPIC: debezium_connect_config
      OFFSET_STORAGE_TOPIC: debezium_connect_offsets
      STATUS_STORAGE_TOPIC: debezium_connect_status
      BOOTSTRAP_SERVERS: kafka:29092
    depends_on:
      - kafka
      - postgres

volumes:
  postgres_data:
    driver: local
  zookeeper_data:
    driver: local
  kafka_data:
    driver: local
```

**Obrázek 3.4.** Příklad toho, jak může vypadat docker-compose. Část 2.

Docker spustí databázi Postgres a provede skript `0.schema.sql`, který inicializuje tabulky v databázi.

```
-- Conforms to the default Debezium structure
-- https://debezium.io/documentation/reference/1.2/configuration/outbox-event-router.html#basic-o

CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE msg_outbox (
  id UUID primary key default uuid_generate_v4(),
  aggregatetype text not null,
  aggregateid UUID not null,
  type text not null,
  payload text not null
);
```

**Obrázek 3.5.** Příklad toho, jak vypadá skript 0.schema.sql

Dále Docker spustí Zookeeper pro koordinaci Apache Kafka a spustí Apache Kafka. Apache Kafka používá dva *porty* 29092 a 9092. První port slouží k interní komunikaci mezi Debezium a Apache Kafka. Druhý port slouží ke komunikaci mezi aplikací a Apache Kafka.

Poslední komponentou bude Debezium. Debezium používá port 8083 pro vnější komunikaci.

Teď je potřeba vytvořit konektor Debezium k databázi Postgres. To provedeme příkazem v konzoli (viz. Obrázek 3.6).

```
remelvic@nvremel:/mnt/c/Users/vremel$ curl -X POST http://localhost:8083/connectors/
-H 'Content-Type: application/json' -d @debezium.json
```

**Obrázek 3.6.** Připojení konektoru Debezium k systému Postgres

Konifguraci pro Debezium odešleme přes port 8083. Konfigurace je uložena ve formátu *JSON* [28] (viz. Obrázek 3.7 a Obrázek 3.8).

```
{
  "name": "transactional-outbox-final",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "tasks.max": "1",
    "plugin.name": "pgoutput",
    "slot.name": "transactional_outbox",

    "database.hostname": "postgres",
    "database.port": "5432",
    "database.dbname": "postgres",
    "database.user": "postgres",
    "database.password": "changeit",
    "database.server.name": "transactional-outbox",

    "schema.whitelist": "public",
    "table.whitelist": "public.msg_outbox",
    "tombstones.on.delete": "false",

    "transforms": "outbox",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.fields.additional.placement": "type:header",

    "key.converter" : "org.apache.kafka.connect.storage.StringConverter",
```

**Obrázek 3.7.** Příklad toho, jak vypadá konfigurační soubor pro Debezium. Část 1.

```
    "value.converter" : "io.debezium.converters.ByteBufferConverter",
    "value.converter.delegate.converter.type": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.delegate.converter.type.schemas.enable": "false"
  }
}
```

**Obrázek 3.8.** Příklad toho, jak vypadá konfigurační soubor pro Debezium. Část 2.

Základní nastavení této konfigurace zahrnuje:

- **connector.class** obsahuje třídu konektoru, pomocí které se má připojit k databázi
- parametry **database.hostname**, **database.port**, **database.dbname**, **database.user** a **database.password** pro připojení k databázi.
- **schema.whitelist** a **table.whitelist** určují, která schémata a tabulky budou sledovány a jejich změny budou předávány do Apache Kafka
- **transforms** a **transforms.outbox.type** slouží ke zpracování a konverzi dat, která mají být předána do Apache Kafka.
- **key.converter** a **value.converter** definují formát klíčů a hodnot, které mají být zapsány do Apache Kafka.

Pokud jsou všechny parametry v konfiguračním souboru správné a pokud je k konektoru navázáno spojení, vrátí dříve spuštěný příkaz odpověď (viz. Obrázek 3.9).

```
remelvic@nvremel:/mnt/c/Users/vremel/studium/bachelor_work/TransactionLogTailing/transactionlogtailingexample/docker/debezium$ ./debezium.sh
```
{"name":"transactional-outbox-final","config":{"connector.class":"io.debezium.connector.postgresql.PostgresConnector","tasks.max":"1","plugin.name":"pgoutput","slot.name":"transactional_outbox","database.hostname":"postgres","database.port":"5432","database.dbname":"postgres","database.user":"postgres","database.password":"changeit","database.server.name":"transactional-outbox","schema.whitelist":"public","table.whitelist":"public.msg_outbox","tombstones.on.delete":"false","transforms":"outbox","transforms.outbox.type":"io.debezium.transforms.outbox.EventRouter","transforms.outbox.table.fields.additional.placement":"type:header","key.converter":"org.apache.kafka.connect.storage.StringConverter","value.converter":"io.debezium.converters.ByteBufferConverter","value.converter.delegate.converter.type":"org.apache.kafka.connect.json.JsonConverter","value.converter.delegate.converter.type.schemas.enable":"false","name":"transactional-outbox-final"},"tasks":[],"type":"source"}remelvic@nvremel:/mnt/c/Users/v

**Obrázek 3.9.** Vrácená odpověď z Debezium po připojení konektoru k databázi Postgres

### 3.4.2 Implementace komponenty Rezervace a ukládání dat do Postgres.

Nejprve ukážu, jak vypadají soubory `application.properties` pro služby „Reservation“ a „SendNotification“.

Ze souboru `application.properties` služby Reservation je vidět, že se aplikace připojuje pouze k databázi Postgres (viz. Obrázek 3.10).

Ze souboru application.properties služby sendReservation je vidět, že se aplikace připojuje pouze ke službě Apache Kafka a nemá žádné připojení k databázi (viz. Obrázek 3.11).

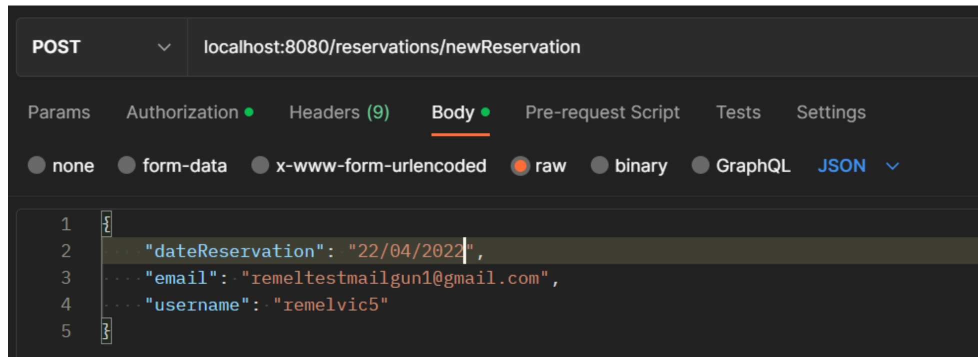
```yaml
server:
  port: ${SERVER_PORT:8080}
spring:
  main:
    banner-mode: off
  zipkin:
    enabled: false
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres
    username: postgres  # todo: change if push to codeNow
    password: changeit  # todo: change if push to codeNow
    databaseName: test
    driver-class-name: org.postgresql.Driver
    validation-query: SELECT 1
  jpa:
#    generate-ddl: true
    database-platform: org.hibernate.dialect.PostgreSQL95Dialect
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
  sleuth:
    propagation:
      type: B3
management:
  endpoints:
    web:
      exposure:
        include: health, prometheus
```

**Obrázek 3.10.** Příklad toho, jak vypadá soubor `application.properties` pro službu Rezervace

```yaml
server:
  port: ${SERVER_PORT:8081}
spring:
  main:
    banner-mode: off
  zipkin:
    enabled: false
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: remel.bachelor.work.outbox
      auto-offset-reset: earliest
    listener:
      ack-mode: MANUAL_IMMEDIATE
  sleuth:
    propagation:
      type: B3
management:
  endpoints:
    web:
      exposure:
        include: health, prometheus
```

**Obrázek 3.11.** Příklad toho, jak vypadá soubor `application.properties` pro službu sendNotification

Pro komunikaci se službou, která vytvoří rezervaci a uloží ji do databáze, používám dva koncové body („getReservations“ a „reservations/newReservation“). „GetReservations“ vrátí seznam všech rezervací a „reservation/newReservations“ přijímá parametry pro vytvoření registrace.




**Obrázek 3.12.** Ukázka, v jaké podobě a v jakém formátu jsou přijímány parametry pro vytvoření registrace

Po získání informací o rezervaci uložím data do databáze do tabulky reservation voláním metody `reservationRepository.save(reservation)`. Po uložení rezervace do databáze vytvořím událost s názvem „NewReservationEvent“. NewReservationEvent přebírá parametry ***username***, ***email***, ***status*** (stav rezervace), ***dateReservation***, ***CreatedOn***(kdy byla událost vytvořena).

Celý tento proces můžete vidět na Obrázku 3.13.

```java
@Override
@Transactional
public String createReservation(NewReservationDTO reservationDTO) throws Exception {
    log.info("Create new reservation details for reservationId: {}", reservationDTO);
    try {
        Reservation reservation = new Reservation();
        reservation.setDateReservation(reservationDTO.getDateReservation());
        reservation.setEmail(reservationDTO.getEmail());
        reservation.setUsername(reservationDTO.getUsername());
        reservation.setStatus("CREATED");

        reservationRepository.save(reservation);
        log.info("Reservation saved successfully");

        //Publish the event
        eventPublisher.publish(reservation, new NewReservationEvent()
                .setUsername(reservation.getUsername())
                .setEmail(reservation.getEmail())
                .setStatus(reservation.getStatus())
                .setDateReservation(reservation.getDateReservation())
                .setCreatedOn(OffsetDateTime.now()));

        log.info("New reservation created");
        return "New reservation created";
```

**Obrázek 3.13.** Ukázka uložení nové rezervace a vytvoření události

Poté událost publikuji do databáze v tabulce msg_outbox pomocí příkazu eventPublisher.publish(...).

Událost je takto vytvořena záměrně, protože konzument bude dostávat informace od message brokeru a pomocí typu události může konzument snadno třídit události a provádět různé operace pro jednotlivé typy událostí.

```
public void publish(@NonNull AbstractPersistable<UUID> aggregate, @NonNull Event event){
    UUID id = UUID.randomUUID();
    jdbcTemplate.update( sql: "insert into msg_outbox(id, aggregateid, aggregatetype, type, payload) values (?, ?, ?, ?, ?)",
            id, aggregate.getId(), aggregate.getClass().getName(), event.getClass().getSimpleName(), objectMapper.writeValueAsString(event));
    log.info("Domain event saved to outbox: eventType: {}, aggregateType: {}, aggregateId: {}",
            event.getClass().getSimpleName(), aggregate.getClass().getName(), aggregate.getId());
    jdbcTemplate.update( sql: "delete from msg_outbox where id = ?", id);
}
```

**Obrázek 3.14.** Ukázka uložení události do tabulky msg_outbox

Tento kód nám ukazuje, že uloží událost do tabulky msg_outbox, vypíše zprávu, že událost uložil do tabulky, a poté událost z tabulky okamžitě odstraní. To nám ukazuje, že Debezium reaguje na změny v databázi v reálném čase. (viz. 3.4.3)

Při uložení rezervace dostaneme zprávu, že rezervace byla uložena a do databáze byla uložena také událost NewReservation.

```
2023-04-29 16:48:42.373  INFO 8476 --- [nio-8080-exec-1] r.b.work.service.ReservationServiceImpl  : Create new reservation details for
 reservationId: NewReservationDTO(dateReservation=Sat Apr 29 16:48:42 CEST 2023, email=remeltestmailgun1@gmail.com, username=remelvic6)
2023-04-29 16:48:42.407  INFO 8476 --- [nio-8080-exec-1] r.b.work.service.ReservationServiceImpl  : Reservation saved successfully
2023-04-29 16:48:42.427  INFO 8476 --- [nio-8080-exec-1] r.bachelor.work.service.EventPublisher   : Domain event saved to outbox: eventType:
 NewReservationEvent, aggregateType: remel.bachelor.work.entity.Reservation, aggregateId: 661e9a89-4c0e-4e16-8e61-dfa704dea707
2023-04-29 16:48:42.429  INFO 8476 --- [nio-8080-exec-1] r.b.work.service.ReservationServiceImpl  : New reservation created
```

**Obrázek 3.15.** Zprava, že rezervace byla uložena do databáze

V tuto chvíli druhá služba SendNotification přijímá zprávy od Apache Kafka, které Apache Kafka obdržela od Debezium.

```
2023-04-29 16:48:42.802  INFO 8456 --- [ntainer#0-0-C-1] r.b.work.config.KafkaConfiguration       : Event consumer received message
2023-04-29 16:48:42.826  INFO 8456 --- [ntainer#0-0-C-1] r.bachelor.work.service.EventConsumer    : Received event
 payload={\\\"username\\\":\\\"remelvic6\\\",\\\"email\\\":\\\"remeltestmailgun1@gmail.com\\\",\\\"status\\\":\\\"CREATED\\\",
 \\\"dateReservation\\\":\\\"2023-04-29T14:48:42.360+00:00\\\",\\\"createdOn\\\":\\\"2023-04-29T16:48:42.4125861+02:00\\\"} with
 aggregateId=661e9a89-4c0e-4e16-8e61-dfa704dea707 aggregateType=outbox.event.remel.bachelor.work.entity.Reservation created
 on=+55295-03-02T03:52:02Z eventId=e564b669-5242-42b2-96d9-9f1c48b61364 eventType=NewReservationEvent
2023-04-29 16:48:42.827  INFO 8456 --- [ntainer#0-0-C-1] r.bachelor.work.service.EventConsumer    : Found username with email -->
 remeltestmailgun1@gmail.com
2023-04-29 16:48:43.893  INFO 8456 --- [ntainer#0-0-C-1] remel.bachelor.work.email.SendEmail      : Mailgun send email about reservation to
 email ->  remeltestmailgun1@gmail.com
2023-04-29 16:48:43.896  INFO 8456 --- [ntainer#0-0-C-1] r.bachelor.work.service.EventConsumer    : Result of sending email -->
 {"id":"<20230429144844.526c7b4a177c2709@sandbox4d26efa5916c46df9994bf47d11ddf6a.mailgun.org>","message":"Queued. Thank you."}
```

**Obrázek 3.16.** Zpráva, že služba sendNotification obdržela zprávu z Apache Kafka

Služba SendNotification čte téma s názvem *outbox.event.** (v našem případě se téma jmenuje ). Přijímá všechny zprávy ve formě řetězce a tato zpráva je uložena v argumentu eventPayload. Ostatní argumenty jsou přijímány jako hlavičky Apache Kafka. Poté zavolá funkci `eventConsumer.consume()`, která zpracuje přijatou zprávu.

```
@KafkaListener(topicPattern = "outbox.event.*")
public void handleInboxMessage(@Payload String eventPayload,
                               @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String aggregateId,
                               @Header(KafkaHeaders.RECEIVED_TOPIC) String aggregateType,
                               @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long createdOn,
                               @Header(EventConsumer.HEADER_EVENT_ID) String eventId,
                               @Header(EventConsumer.HEADER_EVENT_TYPE) String eventType,
                               Acknowledgment ack){
    log.info("Event consumer received message");
    eventConsumer.consume(StringEscapeUtils.escapeJson(eventPayload.substring(1, eventPayload.length() -1)),
            UUID.fromString(aggregateId),
            aggregateType,
            OffsetDateTime.ofInstant(Instant.ofEpochSecond(createdOn), ZoneOffset.UTC),
            UUID.fromString(eventId),
            eventType.replace( target: "outbox.event", replacement: ""));
    ack.acknowledge();
}
```

**Obrázek 3.17.** Ukázka obdržení zprávy z aplikace Apache Kafka

Metoda `eventConsumer.consume()` zpracuje zprávu, která přišla z Apache Kafka. Metoda převezme e-mail z řetězce a zavolá `SendEmail.sendSimpleEmail()`, aby odeslala e-mail o nové registraci.

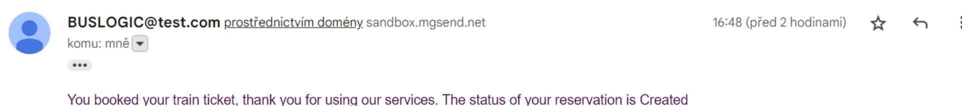
```
@Transactional
public void consume(String eventPayloadJson,
                    UUID aggregateId,
                    String aggregateType,
                    OffsetDateTime createdOn,
                    UUID eventId,
                    String eventType) {
    log.info("Received event payload=" + eventPayloadJson + " with aggregateId=" + aggregateId + " aggregateType=" +
            aggregateType + " created on=" + createdOn + " eventId=" + eventId + " eventType=" + eventType);

    String status = "?";
    Pattern patternEmail = Pattern.compile("\\w+@\\w+\\.\\w+");
    Matcher matcherEmail = patternEmail.matcher(eventPayloadJson);
    try {
        if (matcherEmail.find()) {
            String email = matcherEmail.group(0);
            log.info("Found username with email --> " + email);
            if (Objects.equals(eventType, "NewReservationEvent")) {
                status = "Created";
            }
            log.info("Result of sending email --> " + SendEmail.sendSimpleEmail(email, status).toString());
        } else {
            log.info("Username doesn't have with email");
        }
    } catch (UnirestException e) {
        log.error("Error sending email");
    }
```

**Obrázek 3.18.** Ukázka zpracování zprávy

Metoda `sendSimpleEmail()` odesílá e-maily pomocí služby Mailgun. Jako parametry získá e-mail uživatele, na který bude e-mail odeslán, a stav rezervace. Metoda vrátí zprávu ve formátu JSON, pokud byl e-mail úspěšně odeslán. Používají se také konstanty DOMAIN_NAME a API_KEY. DOMAIN_NAME se používá pro odesílání pošty prostřednictvím služby Mailgun. API_KEY se používá pro *autentifikaci*[4] při odesílání e-mailů přes Mailgun.

```
public static JsonNode sendSimpleEmail(String email, String status) throws UnirestException {
    HttpResponse<JsonNode> request = Unirest.post("https://api.mailgun.net/v3/" + DOMAIN_NAME + "/messages")
            .basicAuth("api", MailgunKey.API_KEY)
            .queryString("from", "BUSLOGIC@test.com")
            .queryString("to", email)
            .queryString("subject", "Create reservation")
            .queryString("text", "You booked your train ticket, thank you for using our services. " +
                    "The status of your reservation is "+status)
            .asJson();
    log.info("Mailgun send email about reservation to email ->  "+email);
    return request.getBody();
}
```

**Obrázek 3.19.** Příklad odeslání e-mailu pomocí služby Mailgun




**Obrázek 3.20.** Příklad zprávy přijaté prostřednictvím služby Mailgun

[4] Autentifikace je proces ověřování identity uživatele.

### 3.4.3 Závěr o Transaction Log Tailing

Podívali jsme se na Transaction Log Tailing a ověřili jsme si, že je **složitější** než Polling Publisher. Hlavní problém může nastat při konfiguraci Debezium. Polling Publisher se snadno implementuje a snadno používá. Nevýhodou Polling Publisher je, že vyžaduje pravidelné dotazování databáze [15], což může být zbytečné, pokud nedochází k častým změnám v databázi.

Výhodou Transaction Log Tailing je, že umožňuje reagovat na změny v databázi v reálném čase. [5] Také Transaction Log Tailing má dvě velké nevýhody:

- Transaction Log Tailing nelze použít pro všechny typy databází.
- Pokud aplikace nezpracovává data dostatečně rychle, hrozí ztráta dat. Protože změny v transakčním protokolu mohou nastat rychleji, než je aplikace zvládne zpracovat.

[5] Podívejte se na Obrázek 3.15, na obrázek 3.16 a na obrázek 3.19. Věnujte pozornost tomu, kdy jsou vypsány logy a kdy zpráva dorazila do Google Mail. Všechny události se stali během několika sekund.

# Kapitola 4
# Porovnání Apache Kafka a RabbitMQ

V této kapitole se seznámíme s message brokery a podíváme se na rozdíly mezi message brokery, jako jsou Apache Kafka a RabbitMQ. Podíváme se také na to, ve kterých případech je nejlépe použít Apache Kafka nebo RabbitMQ.

## 4.1 Message broker

Existují dva modely komunikace mezi mikroslužbami. Jedná se o synchronní a asynchronní komunikaci. **Synchronní komunikace** představuje typ komunikace, kdy jedna funkce volá jinou funkci a čeká na odpověď druhé funkce v realném čase [29].

**Asynchronní komunikací** představuje typ komunikace, kdy jedna funkce volá jinou funkci, ale nečeká na její odpověď a pokračuje ve své práci [30]. Uživatel například odešle asynchronní požadavek a místo čekání na odpověď může pokračovat v prohlížení webu.

Abychom zajistili, že druhá služba při asynchronní komunikaci požadavek obdrží, používáme message broker. *Message broker* je mezivrstva mezi odesílatelem a příjemcem zpráv. Message broker zpracovává zprávy, přesměrovává je a spravuje jejich distribuci, čímž zajišťuje spolehlivou výměnu zpráv mezi producenty a konzumenty.

### 4.1.1 Jak funguje message broker

Funkčnost je založena na následujících krocích:

- Odesílatel odešle zprávu do message brokeru. Zpráva může být v různých formátech, například JSON a *XML* [31].
- Message broker zprávu přijme a provede její směrování k příjemci. Pravidla směrování jsou definována předem. Podle těchto pravidel jsou zprávy odesílány do určité fronty(viz. 4.4) nebo do určitého topiku(viz. 4.2.1).
- Příjemce obdrží zprávu od message brokeru a zpracuje ji.

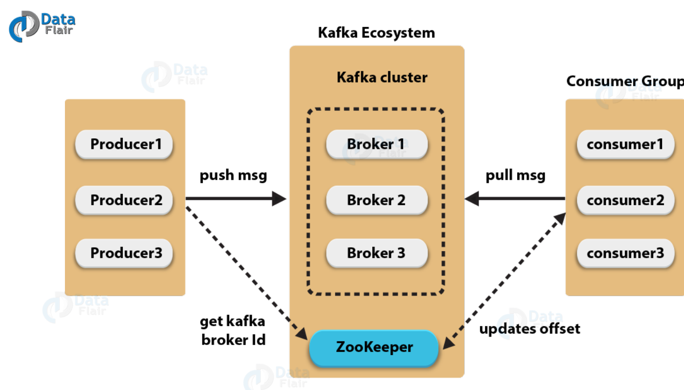
Message broker může také ukládat zprávy do databáze, hlásit chyby a spravovat fronty zpráv.

## 4.2 Apache Kafka

Apache Kafka je distribuovaný systém pro zasílání zpráv typu publish-subscribe, který přijímá data z různých zdrojových systémů a zpřístupňuje je cílovým systémům v reálném čase [32].

### 4.2.1 Jak funguje Apache Kafka

Producenti posílají zprávy do brokeru a konzumenti přijímají data od brokera. Může existovat jeden nebo více brokerů. Zprávy jsou rozděleny podle *témat* (topics), která označují zprávy podle jejich typu [33].



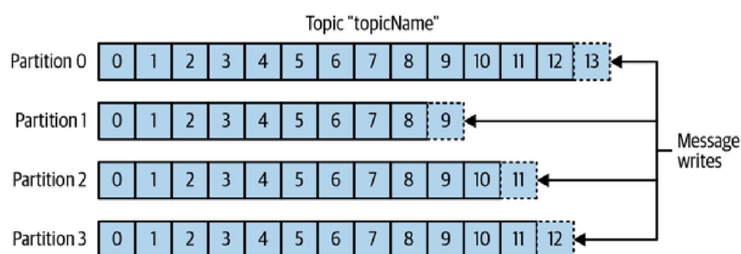

**Obrázek 4.1.** Jak Apache Kafka komunikuje s producenty a konzumenty.(*Kafka Architecture and Its Fundamental Concepts* [34])

*Téma* (topic) je logické rozlišení, které definuje typ údajů od producenta. Například údaje o objednávce produktu a údaje o platební kartě z internetového obchodu jsou dvě různá témata.

*Oddíly* (partitions) jsou kontejnery, ve kterých jsou uloženy sady dat z určitého tématu. Zprávy se přenášejí mezi všemi oddíly. Standardně se k přenosu zpráv mezi oddíly používá princip Round-Robin [35], ale algoritmus lze změnit.

*Zprávy* (message) obsahují klíč (key), hodnotu (value) a pole posunu (offset). Zprávy v sobě mají název tématu, aby broker mohl určit, kam má zprávy přidělit.

*Offset* je celé postupné číslo, které Kafka ukládá pro každou zprávu. Toto číslo pomáhá řadit zprávy v rámci oddílů.




**Obrázek 4.2.** Jak vypadá topic, partitions a způsob záznamu zpráv.(*Kafka The Definitive Guide* [33])

*Brokery* si lze představit jako servery, které ukládají témata a jejich oddíly. Jak již bylo zmíněno, brokerů může být více než jeden. V takových systémech může být jeden oddíl rozdělen mezi ostatní brokery. Počet brokerů, mezi které jsou oddíly rozděleny, je určen faktorem replikace (ve standardním nastavení je 3). V takových systémech je jeden broker vedoucím a provádí takové funkce, jako je přidání a odstranění zprávy. Ostatní brokery jsou sledovatelé a mohou se synchronizovat s vedoucím brokerem.

Zjednodušeně lze říci, že komunikace v Kafce funguje následovně:

Producenti odesílají data do Kafky ve formě zprávy. Zprávy obsahují Topic ID, s nímž Kafka odešle zprávu hlavnímu brokeru pro dané téma. Když je zpráva na dané téma zapsána do oddílu, ukládá Zookeeper číslo offsetu. Když chce konzument zprávu přečíst,

odešle požadavek obsahující offset poslední přečtené zprávy. Zookeeper odesílá zprávy počínaje číslem offsetu přijatým od konzumenta. Zookeeper může posílat i několik zpráv najednou. Počet odeslaných zpráv určuje konzument.

## 4.3 Ukazka Apache Kafka code

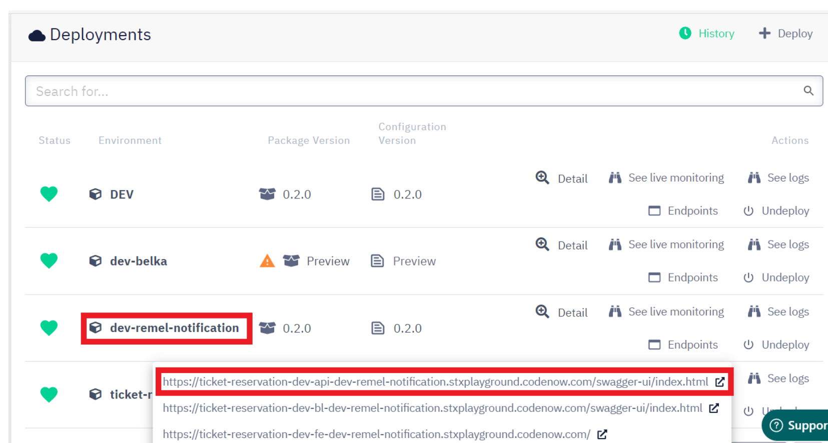
Jeden příklad implementace Apache Kafka byl uveden při implementaci Transaction Log Tailing (viz. 3.4.2).

Druhý příklad implementace Apache Kafka ukážu na implementaci služby notifikací uživatelů na CodeNow [36].

Na platformě CodeNow je k dispozici aplikace `ticket-reservation-dev` pro rezervaci vstupenek. Z této aplikace používám tři mikroslužby ticket-`reservation-dev-api` (komunikace s backendem), `ticket-reservation-dev-bl` (business logika, zároveň backend), `ticket-reservation-dev-fe` (frontend). Přes backend provedeme rezervaci, která se uloží do databáze, a rezervační zpráva se odešle do Kafky.

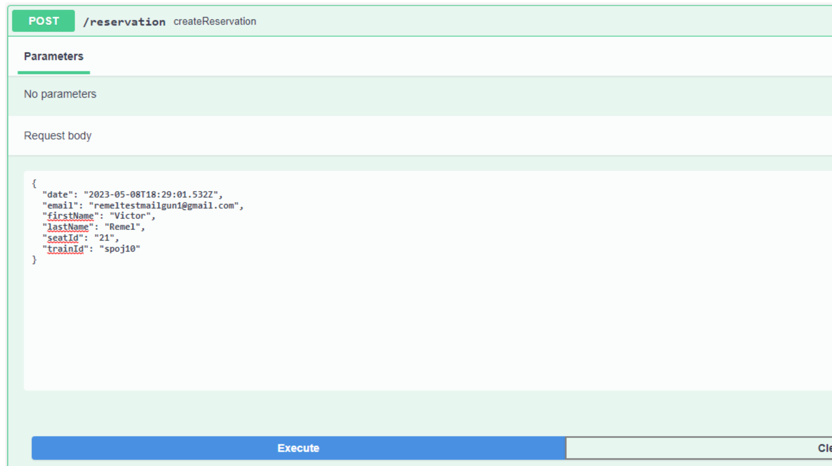
Na platformě CodeNow vybereme aplikaci ticket-reservation-dev, vyhledáme část *Deployment*[1] a vybereme *Environment*[2] dev-remel-notification, vybereme koncový bod ticket-reservation-dev-api-dev-remel-notification a poté prostřednictvím nástroje Swagger vytvoříme novou rezervaci.

Vzhledem k tomu, že používám Mailgun a používám bezplatný účet, lze e-maily odesílat pouze na potvrzené e-maily. [3] Je možné provést rezervaci pomocí jiného e-mailu a rezervace se uloží a moje služba obdrží zprávu od Kafky, ale e-mail se neodešle.



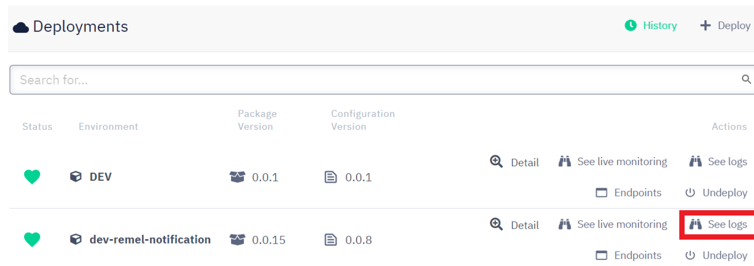

**Obrázek 4.3.** Příklad výběru koncového bodu pro vytvoření rezervace

[1] Deployment je proces publikování nebo nahrávání aplikace na server.
[2] Environment je prostředí, ve kterém je aplikace vyvinuta a testována. Prostředí má svá vlastní nastavení, jako je například databáze
[3] E-maily lze potvrdit na účtu Mailgun osobně nebo si lze zakoupit předplatné Mailgun, aby bylo možné e-maily odesílat [37].



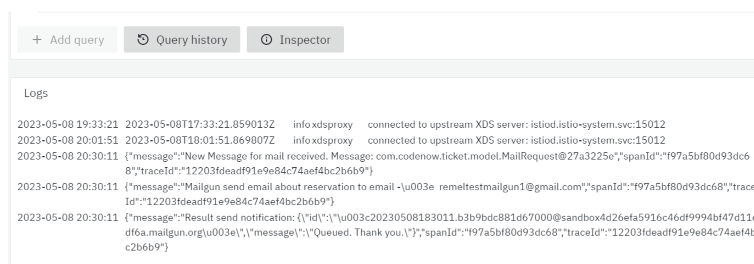

**Obrázek 4.4.** Ukázka předávání parametrů prostřednictvím nástroje Swagger pro vytvoření rezervace

Poté stačí kliknout na tlačítko `Execute` a rezervace se uloží do databáze. Zpráva o rezervaci bude odeslána do Kafky. Nyní je potřeba najít aplikaci ticket-notification-dev. V části Deployment zvolit Environment `dev-remel-notification` a kliknout na `See logs`.




**Obrázek 4.5.** Příklad jak číst logy aplikace v CodeNow

V logách najdeme zprávu, že byla přijata zpráva z Kafky, a zprávu, že byl odeslán e-mail o rezervaci na adresu remeltestmailgun1@gmail.com.




**Obrázek 4.6.** Příklad toho, jak vypadají logy odeslaného e-mailu s informacemi o rezervaci.

Konfigurace MailNotificationConfiguration pro čtení zpráv ze služby Apache Kafka je následující.

```
1 usage
private final String bootstrapServers = "sharedkafka-kafka-bootstrap.demo-env:9092";
1 usage
private final String groupId = "mail-sender-id";
1 usage  Victor Remel *
@Bean
public ConsumerFactory<String, MailRequestDTO> mailNotificationConsumerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
    props.put(ConsumerConfig.CLIENT_ID_CONFIG, UUID.randomUUID().toString());
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ErrorHandlingDeserializer.class);
    props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS, JsonDeserializer.class);
    props.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
    return new DefaultKafkaConsumerFactory<>(props,new StringDeserializer(),
            new JsonDeserializer<>(MailRequestDTO.class));
}
1 usage  Victor Remel
@Bean
public ConcurrentKafkaListenerContainerFactory<String, MailRequestDTO> mailNotificationKafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, MailRequestDTO> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(mailNotificationConsumerFactory());
    factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
    return factory;
```

**Obrázek 4.7.** Ukázka konfigurace pro čtení zpráv z Apache Kafka.

`MailNotificatioConsumer` přijímá zprávu prostřednictvím tématu reservation-mail-notification a přijímá z něj `MailRequest`. Vypíše log s informací o tom, zda byla zpráva odeslána, nebo ne, prostřednictvím `Mailgun`.

```
@Service
@Slf4j
public class MailNotificationConsumer {

    no usages
    @Autowired
    private SendEmailService sendEmailService;

    // listen to new messages(mail requests) from kafka
    no usages  Victor Remel *
    @KafkaListener(topics = "reservation-mail-notification", containerFactory="mailNotificationKafkaListenerContainerFactory"
    public void mailListener(@Payload MailRequest mailRequest, Acknowledgment ack) throws InterruptedException {
        log.info("New Message for mail received. Message: "+ mailRequest);
        try {
            log.info("Result send notification: " + SendEmailService.sendSimpleEmail(mailRequest));
        } catch (UnirestException e) {
            throw new RuntimeException(e);
        }
        ack.acknowledge();
    }
```

**Obrázek 4.8.** Ukázka jak vypadá konzument Apache Kafka

Odeslání e-mailu pomocí Mailgun vypadá následovně: metoda `sendSimpleEmail` obdrží parametr `MailRequest`, který obsahuje informace o tom, komu má být e-mail odeslán, předmět e-mailu a text zprávy. Jak Mailgun odesílá e-maily jsem psal dříve (viz. 3.4.2)

```java
1 usage
private static final String DOMAIN_NAME = "sandbox4d26efa5916c46df9994bf47d11ddf6a.mailgun.org";

/**
 * The method gets the recipient's email, configures the email, and sends the email
 * @param email Who we're sending the letter to
 * @return the result if the email is sent or returns an exception
 */
1 usage   Victor Remel *
public static JsonNode sendSimpleEmail(MailRequest mailRequest) throws UnirestException {
    HttpResponse<JsonNode> request = Unirest.post( url: "https://api.mailgun.net/v3/" + DOMAIN_NAME + "/messages")
            .basicAuth( username: "api", MailgunKey.API_KEY)
            .queryString( name: "from",  value: "remelNotification@test.com")
            .queryString( name: "to", mailRequest.getTo())
            .queryString( name: "subject", mailRequest.getSubject())
            .queryString( name: "text", mailRequest.getBody())
            .asJson();
    log.info("Mailgun send email about reservation to email ->  "+mailRequest.getTo());
    return request.getBody();
}
```

**Obrázek 4.9.** Ukázka jak vypadá odeslání e-mailu pomocí Mailgun

### 4.3.1 Testování Apache Kafka

Protože všechny aplikace, které obsahují Apache Kafka (kromě Transaction Log Tailing[4]) musí běžet v CodeNow a byly napsány tak, aby běžely v CodeNow, proto budou testy napsány pro jiné aplikace. Tato čast víceméně ukáže, jak testovat aplikace, které obsahují Apache Kafka.

Testoval jsem pomocí JUnit a tento test ukazuje, zda je zpráva správně odeslána od producenta ke konzumentovi.Aby test fungoval správně, musí být spuštěny aplikace Apache Kafka a Zookeeper.

K odesílání a přijímání zpráv používám instance KafkaProducer a KafkaConsumer. Zprávy jsou odesílány přes téma test-topic.

Pomocí anotace @Before před testem provedu konfiguraci tak, aby producent a konzument byli správně připojeni k Apache Kafka. Pomocí anotace @After po skončení testu uvolním zdroje, tedy KafkaProducer a KafkaConsumer.

Samotný test s KafkaProducer odešle zprávu přes téma test-topic. KafkaConsumer přijímá zprávu z tohoto tématu. Poté pomocí metody Assert.assserEquals() porovná přijatou zprávu s odeslanou zprávou, pokud se neshodují, test selže.

[4] Po dohodě s vedoucím jsme se rozhodli, že nebudu psát testy pro Transaction Log Tailing, protože je to nadprůměrně složité.

```
2 usages
private static final String BOOTSTRAP_SERVERS = "localhost:9092";
2 usages
private static final String TOPIC_NAME = "test-topic";
2 usages
private static final String MESSAGE_CONTENT = "Hello, Kafka!";
3 usages
private KafkaProducer<String, String> producer;
4 usages
private Consumer<String, String> consumer;
no usages   new *
@Before
public void setup() {
    Properties producerProps = new Properties();
    producerProps.setProperty("bootstrap.servers", BOOTSTRAP_SERVERS);
    producerProps.setProperty("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    producerProps.setProperty("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    producer = new KafkaProducer<>(producerProps);
    Properties consumerProps = new Properties();
    consumerProps.setProperty("bootstrap.servers", BOOTSTRAP_SERVERS);
    consumerProps.setProperty("group.id", UUID.randomUUID().toString());
    consumerProps.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    consumerProps.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    consumerProps.setProperty("auto.offset.reset", "earliest");
    consumer = new KafkaConsumer<>(consumerProps);
    consumer.subscribe(Collections.singleton(TOPIC_NAME));
}
```

**Obrázek 4.10.** Ukázka jak vypadá testování Apache Kafka. Část 1.

```
no usages   new *
@After
public void cleanup() {
    producer.close();
    consumer.close();
}
no usages   new *
@Test
public void testKafkaProducerAndConsumer() {
    ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC_NAME, MESSAGE_CONTENT);
    producer.send(record);
    log.info("Message send");
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    Assert.assertFalse(records.isEmpty());
    ConsumerRecord<String, String> received = records.iterator().next();
    log.info("received message: " + received);
    Assert.assertEquals(MESSAGE_CONTENT, received.value());
}
```

**Obrázek 4.11.** Ukázka jak vypadá testování Apache Kafka. Část 2.

```
 (Re-)joining group
2023-05-12 12:57:55.713  INFO 11460 --- [           main] r.b.work.service.ApplicationTests        : Started ApplicationTests in
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 18.694 s - in remel.bachelor.work.service.ApplicationTest
2023-05-12 12:57:56.786  INFO 11460 --- [ntainer#0-0-C-1] o.a.k.c.c.internals.AbstractCoordinator  : [Consumer clientId=consumer
 consumer-reveiver-1-becf6e6d-6b57-49e2-9063-db91d45f15a9 sending LeaveGroup request to coordinator localhost:9092 (id: 2147482
 unsubscribed from all topics
2023-05-12 12:57:56.788  INFO 11460 --- [ntainer#0-0-C-1] o.a.k.clients.consumer.KafkaConsumer     : [Consumer clientId=consumer
 Unsubscribed all topics or patterns and assigned partitions
2023-05-12 12:57:56.788  INFO 11460 --- [ntainer#0-0-C-1] o.s.s.c.ThreadPoolTaskScheduler          : Shutting down ExecutorServi
2023-05-12 12:57:58.769  INFO 11460 --- [ntainer#0-0-C-1] essageListenerContainer$ListenerConsumer : reveiver: Consumer stopped
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  33.856 s
```

**Obrázek 4.12.** Ukázka výpisu úspěšného testu

## 4.4 RabbitMQ

RabbitMQ je message broker založený na protokolu AMQP (Advanced Message Queuing Protocol [38]).RabbitMQ přesně ví, kdy se data nedostanou ke konzumentovi.

RabbitMQ se skládá z několika komponent [39]:

- **AMQP broker** předává zprávy od spotřebitele příjemci.
- **Producent** odesílá zprávy brokeru a **konzument** zprávy přijímá.
- **Exchange** přijímá zprávy a distribuuje je do fronty
- **Fronta** (queue) ukládá zprávy a předává je přihlášeným konzumentům.
- **Vazba na frontu** (Queue Binding) je spojovacím článkem mezi frontou a výměníkem. Obsahuje pravidla směrování, podle kterých budou zprávy předávány z výměníku do konkrétní fronty.

### 4.4.1 Jak funguje RabbitMQ

Zpráva musí být odeslána ve formátu JSON. Zprávy mají tělo a hlavičku. Hlavička obsahuje údaje o směrování, tělo obsahuje data, která chce producent odeslat. Zprávy jsou odeslány do exchangeru, exchanger zkontroluje vazební pravidla (binding rules) a z exchangeru jsou předány do fronty.

Podívejme se na několik typů exchangu:

- **Fanout:** zprávy jsou odesílány do všech front
- **Direct:** zprávy jsou odesílány do všech front, které mají v záhlaví stejný klíč jako zpráva.
- **Topic:** zprávy jsou odesílány do front, kde existuje shoda mezi směrovacím klíčem a směrovacím vzorem uvedeným ve vazbě.
- **Headers:** zpráva je odeslána do front, kde je shoda se záhlavím.

Zprávy jsou z fronty vymazány buď automaticky po přečtení zprávy spotřebitelem, nebo když spotřebitel odešle potvrzení, že zprávu přečetl.

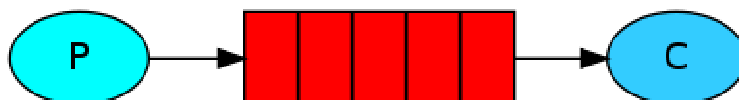
## 4.5 Ukazka RabbitMQ code

Ukážu několik příkladů použití RabbitMQ, přesněji řečeno

- Jak RabbitMQ funguje, když producent přidává do fronty zprávy a konzument z této fronty čte.
- Jak funguje spolehlivé publikování s potvrzením od konzumenta.

### 4.5.1 Jednoduchá komunikace

Tento příklad je jednoduchý, protože máme jednoho producenta, který zapisuje zprávu do jedné konkrétní fronty, a konzument z této fronty ji čte (viz. Obrázek 4.13).




**Obrázek 4.13.** Jak funguje jednoduchá komunikace v RabbitMQ [40].

Nejprve musíme vytvořit konfigurační soubor (viz. Obrázek 4.14).

```java
@Configuration
public class MessagingConfig {
    public static final String QUEUE = "messaging_queue";
    public static final String TOPIC_EXCHANGE = "messaging_exchange";
    public static final String ROUTING_KEY = "messaging_routingKey";
    @Bean
    public Queue queue() { return new Queue(QUEUE); }
    @Bean
    public TopicExchange exchange() { return new TopicExchange(TOPIC_EXCHANGE); }
    @Bean
    public Binding binding(Queue queue,TopicExchange exchange){
        return BindingBuilder.bind(queue).to(exchange).with(ROUTING_KEY);
    }
    @Bean
    public MessageConverter converter() { return new Jackson2JsonMessageConverter(); }
    @Bean
    public AmqpTemplate template(ConnectionFactory connectionFactory){
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(converter());
        return rabbitTemplate;
    }
}
```

**Obrázek 4.14.** Příklad konfigurace RabbitMQ v jazyce Java.

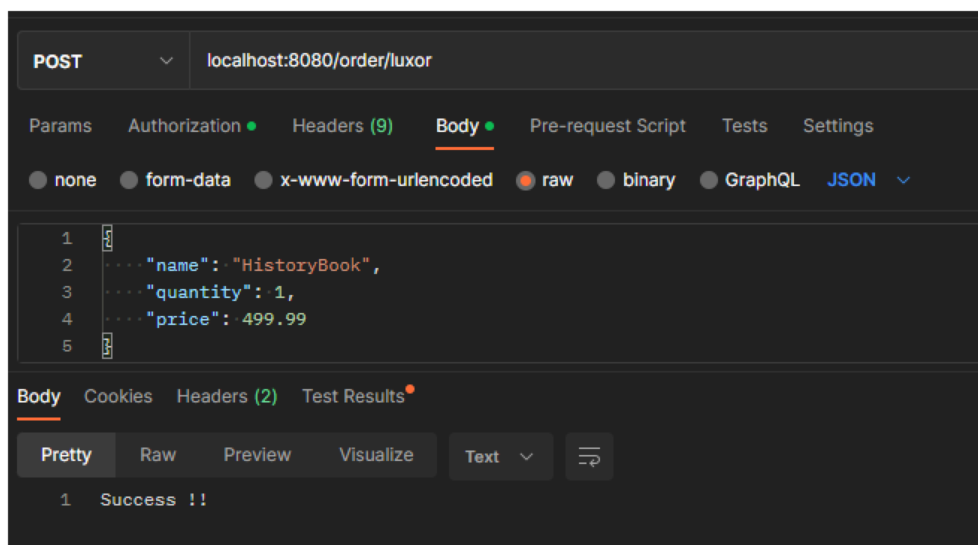
Konfigurační soubor obsahuje 5 metod, které vracejí objekty pro implementaci závislostí. Jsou to:

- *Metoda queue()* vytvoří frontu s názvem `messaging_queue`. Metoda vrátí frontu z importované knihovny org.springframework.amqp.core.
- *Metoda exchange()* vytvoří novou exchange s názvem `messaging_exchange`. TopicExchange je exchange, která řadí zprávy do fronty pomocí směrovacích klíčů.
- *Metoda binding()* spojuje Frontu a TopicExchange pomocí směrovacího klíče `messaging_routingKey`.
- *Metoda converter()* pomáhá převádět objekty na JSON a zpět.
- *Metoda template()* vytvoří šablonu pro odesílání a přijímání zpráv z fronty.

V tomto příkladu je Controller producentem, kterému posíláme určité údaje o objednávce. K objednávce přidáme stav a zprávu o objednávce. Objednávku odešleme do fronty a vypíšeme zprávu, že objednávka byla odeslána.

```java
@RestController
@Slf4j
@RequestMapping("/order")
public class Publisher {
    1 usage
    @Autowired
    private RabbitTemplate template;
    no usages  new *
    @PostMapping("/{shopName}")
    public String bookOrder(@RequestBody Order order, @PathVariable String shopName) {
        order.setOrderId(UUID.randomUUID().toString());
        OrderStatus orderStatus = new OrderStatus(order, status: "PROCESS", message: "order placed succesfully in " + shopName);
        template.convertAndSend(MessagingConfig.TOPIC_EXCHANGE, MessagingConfig.ROUTING_KEY, orderStatus);
        log.info("Order {} send success", order);
        return "Success !!";
    }
}
```

**Obrázek 4.15.** Příklad producenta v RabbitMQ v jazyce Java.




**Obrázek 4.16.** Příklad toho, jak vypadají data odeslaná přes Postman.

Na rozdíl od Apache Kafka poskytuje RabbitMQ snadno použitelné uživatelské rozhraní bez připojení knihoven třetích stran. Po spuštění aplikace a konfiguračního souboru můžeme přejít na stránku *http://localhost:15672/*. Objeví se před námi přihlašovací okno s přihlašovacím jménem a heslem. Přihlašovací jméno a heslo **je** slovo: **guest**.

Zde nalezneme seznam front, které máme k dispozici. Každá fronta má informace o zprávách, které jsou v ní uloženy, dokud je konzument nepřečte.

Protože jsme konzumenta ještě nespustili, můžeme se podívat, jaká zpráva je ve frontě uložena (viz. Obrázek 4.17).

The server reported 0 messages remaining.

| | |
|---|---|
| Exchange | messaging_exchange |
| Routing Key | messaging_routingKey |
| Redelivered | • |
| Properties | priority: 0<br>delivery_mode: 2<br>headers: __TypeId__: remel.bachelor.work.service.dto.OrderStatus<br>content_encoding: UTF-8<br>content_type: application/json |
| Payload 174 bytes Encoding: string | {"order":{"orderId":"ee4b9268-7306-4dda-9d47-153560e61a64","name":"HistoryBook","quantity":1,"price":499.99},"status":"PROCESS" |

**Obrázek 4.17.** Příklad uložení zpráv ve frontě v RabbitMQ.

Konzument obsahuje úplně stejný konfigurační soubor jako producent, ale neobsahuje žádné informace o frontě, o TopicExchange a o vazbě. Příklad konzumenta (viz. Obrázek 4.18).

```
no usages   new *
@Component
public class Consumer {

    no usages   new *
    @RabbitListener(queues = "messaging_queue")
    public void consumeMessageFromQueue(OrderStatus orderStatus){
        System.out.println("Message from queue "+ orderStatus);
    }

}
```

**Obrázek 4.18.** Příklad konzumenta v RabbitMQ v jazyce Java.

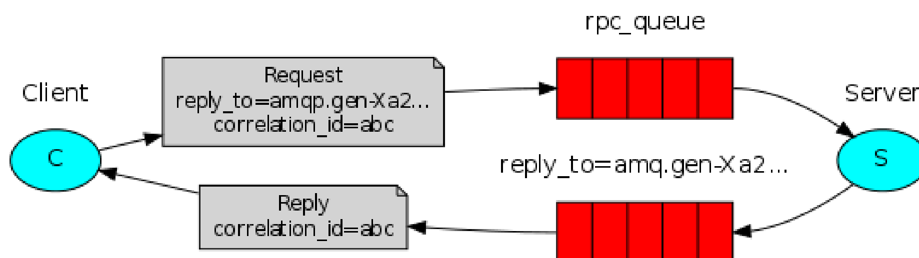
Anotace @RabbitListener definuje, že metoda je konzumentem zpráv z fronty_zpráv. Když se ve frontě objeví zpráva, metoda se automaticky zavolá a vypíše zprávu, kterou z fronty přečetla. Když jsem spustil konzumenta, ve frontě již byla zpráva, kterou jsem předal dříve, a konzument mi ji vypsal.

```
2023-04-25 23:22:44.816  INFO 12140 --- [           main] remel.bachelor.work.service.Application  : Started Application in 3.078 seconds (JVM running for 3.922)
Message from queue OrderStatus(order=Order(orderId=ee4b9268-7306-4dda-9d47-153560e61a64, name=HistoryBook, quantity=1, price=499.99), status=PROCESS, message=order placed succesfully in luxor)
```

**Obrázek 4.19.** Příklad výpisu konzumenta při čtení fronty.

## 4.5.2 Spolehlivé publikování

V tomto příkladu ukážu, jak funguje vzor RPC (Remote Procedure Call). Jedná se o vzor interakce mezi klientem a serverem, kde klient odesílá požadavek na server, server ho zpracovává a posílá zpět klientovi (viz. Obrázek 4.20).




**Obrázek 4.20.** Jak funguje RPC v RabbitMQ [40].

Stejně jako v předchozím příkladu je třeba vytvořit frontu, ale v tomto případě budou fronty dvě, jedna pro požadavky `requestQueue` a druhá pro odpovědi `replyQueue`.

```java
@Configuration
@EnableScheduling
public class RabbitMQConfiguration {
    1 usage
    @Value("${routingKey.request}")
    private String requestRoutingKey;
    1 usage
    @Value("${exchange.direct}")
    private String directExchange;
    1 usage
    @Value("${queue.request}")
    private String requestQueue;
    2 usages
    @Value("${queue.reply}")
    private String replyQueue;
    1 usage
    @Autowired
    private RabbitTemplate rabbitTemplate;
    1 usage
    @Autowired
    private ConnectionFactory connectionFactory;
    no usages   new *
    @Bean
    DirectExchange exchange() { return new DirectExchange(directExchange); }
    no usages   new *
    @Bean
    Queue requestQueue() { return QueueBuilder.durable(requestQueue).build(); }
```

**Obrázek 4.21.** Příklad konfigurace RabbitMQ v jazyce Java pro RPC. Část 1.

```java
@Bean
Binding binding(Queue requestQueue, DirectExchange exchange) {
    return BindingBuilder.bind(requestQueue).to(exchange).with(requestRoutingKey);
}

no usages   new *
@Bean
AsyncRabbitTemplate template() {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);
    container.setQueueNames(replyQueue);
    return new AsyncRabbitTemplate(rabbitTemplate, container);
}
```

**Obrázek 4.22.** Příklad konfigurace RabbitMQ v jazyce Java pro RPC. Část 2.

Tedy také vytvoří `exchange()`, která odešle zprávu pouze těm frontám, které mají stejný směrovací klíč. `Binding()` váže frontu požadavků na výměnu. `Template()` vytvoří šablonu pro odesílání a přijímání zpráv z fronty.

Producent obsahuje metodu `publishToDirectExchangeRPCStyle()`, která každých 100 milisekund vytvoří novou zprávu (díky anotaci `@Scheduled`). Tato metoda vygeneruje celé číslo a vytvoří zprávu jako řetězec, který toto číslo obsahuje. K odeslání zprávy do fronty používá šablonu asyncRabbitTemplate. Používá také callback pro zpracování odpovědi na požadavek (viz. Obrázek 4.23).

```
@Configuration
@Slf4j
public class Publisher {
    1 usage
    @Value("${routingKey.request}")
    private String requestRoutingKey;
    1 usage
    @Value("${exchange.direct}")
    private String directExchange;
    1 usage
    @Autowired
    private AsyncRabbitTemplate asyncRabbitTemplate;
    1 usage
    private static AtomicInteger atomicInteger = new AtomicInteger( initialValue: 1);
    no usages   new *
    @Scheduled(fixedDelay = 100 * 10)
    public void publishToDirectExchangeRPCStyle() {
        Integer integer = atomicInteger.getAndIncrement();
        String sampleRequestMessage = String.valueOf(integer);
        log.info("Sending out message on direct directExchange:" + integer);
        AsyncRabbitTemplate.RabbitConverterFuture<String> sampleResponseMessageRabbitConverterFuture = asyncRabbitTemplate
                .convertSendAndReceive(directExchange, requestRoutingKey, String.valueOf(integer));
        sampleResponseMessageRabbitConverterFuture.addCallback(
                sampleResponseMessage ->
                        System.out.println("Response for request message:" + sampleRequestMessage + " is:" + sampleResponseMessage)
                , failure ->
                        System.out.println(failure.getMessage())
        );
```

**Obrázek 4.23.** Příklad producenta v RabbitMQ s použitím RPC.

Konzument obsahuje ve svém konfiguračním souboru pouze informace o frontě požadavků `requestQueue` jako producent.

Metoda `subscribeToRequestQueue()` je zavolána, když konzument obdrží zprávu z fronty pomocí anotace `@RabbitListener`. `SampleRequestMessage` je standardní zpráva a `message` obsahuje zprávu a informace o ní. V rámci metody dojde k prodlevě 0 až 5000 milisekund a poté je zpráva vrácena jako odpověď (viz. Obrázek 4.23).

Na dvou obrázcích (viz. Obrázek 4.24) a (viz. Obrázek 4.25) je vidět výsledek práce producenta a konzumenta.

```
2023-05-03 23:17:20.929  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:1
2023-05-03 23:17:20.933  INFO 9896 --- [           main] remel.bachelor.work.service.Application  : Started Application in 4.298 seconds (JVM runni
2023-05-03 23:17:21.940  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:2
2023-05-03 23:17:22.947  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:3
Response for request message:1 is:1
2023-05-03 23:17:23.954  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:4
2023-05-03 23:17:24.963  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:5
Response for request message:2 is:2
2023-05-03 23:17:25.978  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:6
2023-05-03 23:17:26.987  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:7
2023-05-03 23:17:27.998  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:8
2023-05-03 23:17:29.012  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:9
2023-05-03 23:17:30.027  INFO 9896 --- [   scheduling-1] remel.bachelor.work.service.Publisher    : Sending out message on direct directExchange:10
Response for request message:3 is:3
```

**Obrázek 4.24.** Příklad výpisu producenta s použitím RPC.

```
2023-05-03 23:17:20.947  INFO 14104 --- [ntContainer#0-1] remel.bachelor.work.service.Subscriber   : Sample request 1
2023-05-03 23:17:20.947  INFO 14104 --- [ntContainer#0-1] remel.bachelor.work.service.Subscriber   : Received message :(Body:'1' MessageProperties [headers={},
 correlationId=9f551b50-2640-4b27-a55b-694e54b37893, replyTo=RPCReply, contentType=text/plain, contentEncoding=UTF-8, contentLength=0, receivedDeliveryMode=PERSISTEN
 priority=0, redelivered=false, receivedExchange=rpc-direct, receivedRoutingKey=RPCRequest, deliveryTag=1, consumerTag=amq.ctag-OdI3Jx7Z8u5vDJheNM-53g,
 consumerQueue=RPCRequest])
2023-05-03 23:17:23.710  INFO 14104 --- [ntContainer#0-1] remel.bachelor.work.service.Subscriber   : Completed processing and sending the message :(Body:'1'
 MessageProperties [headers={}, correlationId=9f551b50-2640-4b27-a55b-694e54b37893, replyTo=RPCReply, contentType=text/plain, contentEncoding=UTF-8, contentLength=0,
 receivedDeliveryMode=PERSISTENT, priority=0, redelivered=false, receivedExchange=rpc-direct, receivedRoutingKey=RPCRequest, deliveryTag=1, consumerTag=amq
 .ctag-OdI3Jx7Z8u5vDJheNM-53g, consumerQueue=RPCRequest])
2023-05-03 23:17:23.728  INFO 14104 --- [ntContainer#0-1] remel.bachelor.work.service.Subscriber   : Sample request 2
2023-05-03 23:17:23.729  INFO 14104 --- [ntContainer#0-1] remel.bachelor.work.service.Subscriber   : Received message :(Body:'2' MessageProperties [headers={},
 correlationId=2eb63a5b-7fe6-45e2-a5e7-97f8798f930d, replyTo=RPCReply, contentType=text/plain, contentEncoding=UTF-8, contentLength=0, receivedDeliveryMode=PERSISTEN
 priority=0, redelivered=false, receivedExchange=rpc-direct, receivedRoutingKey=RPCRequest, deliveryTag=2, consumerTag=amq.ctag-OdI3Jx7Z8u5vDJheNM-53g,
 consumerQueue=RPCRequest])
2023-05-03 23:17:25.853  INFO 14104 --- [ntContainer#0-1] remel.bachelor.work.service.Subscriber   : Completed processing and sending the message :(Body:'2'
 MessageProperties [headers={}, correlationId=2eb63a5b-7fe6-45e2-a5e7-97f8798f930d, replyTo=RPCReply, contentType=text/plain, contentEncoding=UTF-8, contentLength=0,
 receivedDeliveryMode=PERSISTENT, priority=0, redelivered=false, receivedExchange=rpc-direct, receivedRoutingKey=RPCRequest, deliveryTag=2, consumerTag=amq
```

**Obrázek 4.25.** Příklad výpisu konzumenta s použitím RPC.

```java
@Configuration
@Slf4j
public class Subscriber {
    @RabbitListener(queues = "${queue.request}")
    public String subscribeToRequestQueue(@Payload String sampleRequestMessage, Message message) {
        log.info("Received message :" + message);
        try {
            Thread.sleep(generateRandomInt(5000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        log.info("Completed processing and sending the message :" + message);
        return sampleRequestMessage;
    }
    private static int generateRandomInt(int upperRange) {
        Random random = new Random();
        return random.nextInt(upperRange);
    }
}
```

**Obrázek 4.26.** Příklad konzumenta v RabbitMQ s použitím RPC.

### 4.5.3 Testování RabbitMQ

Jak jsem psal dříve 4.3.1, testy pro RabbitMQ budou také realizovány na jiném příkladu.

Ke kontrole odeslání a přijetí zprávy používám jednoduchý test JUnit. Aby tento test fungoval správně, je nutné spustit RabbitMQ, aby spolu mohli producent a konzument komunikovat.

Metoda `testSendAndReceiveMessage` vytvoří připojení k lokálnímu serveru RabbitMQ. Poté se vytvoří kanál, který obsahuje frontu nazvanou `test-queue`. Vytvoří se také blokující fronta `queue`, která slouží k čekání na přijetí zprávy. DefaultConsumer poslouchá frontu `test-queue` a přidává přijaté zprávy do blokující fronty. Poté během 5 sekund obdržíme zprávu z blokující fronty a porovnáme ji s odeslanou zprávou. Pokud se přijatá zpráva neshoduje s odeslanou zprávou, test selže.

```java
private final static String QUEUE_NAME = "test_queue";
private final static String MESSAGE = "Hello, RabbitMQ!";
@Test
public void testSendAndReceiveMessage() throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    BlockingQueue<String> queue = new LinkedBlockingQueue<>();
    DefaultConsumer consumer = new DefaultConsumer(channel) { @Override
        public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) {
            String message = new String(body, StandardCharsets.UTF_8);
            queue.offer(message);
        }
    };
    channel.basicConsume(QUEUE_NAME, true, consumer);
    channel.basicPublish("", QUEUE_NAME, null, MESSAGE.getBytes(StandardCharsets.UTF_8));
    String receivedMessage = queue.poll(5, TimeUnit.SECONDS);
    log.info("received message: " + receivedMessage);
    assertEquals(MESSAGE, receivedMessage);
    channel.close();
    connection.close();
}
```

**Obrázek 4.27.** Ukázka jak vypadá testování RabbitMQ

```
2023-05-12 17:24:32.715  INFO 14412 --- [           main] r.b.work.service.ApplicationTests        : Starting Application
 19.0.2 on nvremel with PID 14412 (started by vremel in C:\Users\vremel\studium\bachelor_work\TESTOVANI\TestovaniRabbitMQ\
2023-05-12 17:24:32.717  INFO 14412 --- [           main] r.b.work.service.ApplicationTests        : The following profile
2023-05-12 17:24:35.538  INFO 14412 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2 endpoint(s
  '/actuator'
2023-05-12 17:24:37.719  INFO 14412 --- [           main] r.b.work.service.ApplicationTests        : Started ApplicationTe
 seconds (JVM running for 6.285)
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.308 s - in remel.bachelor.work.service.Applicatio
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  9.039 s
```

**Obrázek 4.28.** Ukázka výpisu úspěšného testu

## 4.6 Porovnání Apache Kafka a RabbitMQ

Jedním z hlavních rozdílů mezi službami Apache Kafka a RabbitMQ je jejich architektura. Apache Kafka je představitelem architektury Publisher/Subscriber, zatímco RabbitMQ má směřovací konstrukci (routing design).

Fronty RabbitMQ pomáhají uspořádat zprávy v rámci brokeru, což Kafka nedělá. Ale fronty nedokážou zpracovávat velké objemy dat, jako to umí Kafka. RabbitMQ to zvládne, ale bude vyžadovat více zdrojů [41].

Kafka používá *Pull* mechanismus, kdy si konzumenti sami vyžádají zprávy. RabbitMQ používá *Push* mechanismus, kdy producenti předávají zprávu do fronty a z fronty je automaticky předána konzumentům. RabbitMQ však nezaručuje pořadí, v jakém jsou zprávy doručeny [41].

Jak jsem se už zmiňoval, oddíly v systému Kafka lze rozdělit mezi brokery. Tím zvýšíme *propustnost systému* [5]. V RabbitMQ můžeme vytvořit více front, čímž se zvýší propustnost systému. Pouze Kafka se škáluje horizontálně přidáním nových strojů, zatímco RabbitMQ se škáluje vertikálně přidáním větší výkonnosti [41].

V RabbitMQ je možné některé zprávy prioritizovat. V takových případech se používá prioritní fronta a prioritní zprávy se do ní zapisují. V Kafka není možné zprávy prioritizovat [41].

RabbitMQ podporuje jazyky Java, Spring Framework, .NET, Ruby, Python, PHP, JavaScript a Node, Go, Swift, Objective-C, Rust, C, C++, Haskell, Erlang, Cobol [42]. Kafka podporuje Python, Javu, Spring Framework, Go, PHP, Node .NET, Ruby, Swift, Rust, C, C++, Haskell, Erlang, Cloujre [43].

RabbitMQ podporuje protokoly AMQP, STOMP,MQTT,HTTP/HTTPS [6]. Apache Kafka podporuje binární protokol přes TCP [41].

### 4.6.1 V jakých případech je lepší použít Apache Kafka a v jakých případech RabbitMQ

Jak jsme viděli, Apache Kafka a RabbitMQ poskytují různé systémy pro zasílání zpráv a samozřejmě mají různé scénáře použití.

Apache Kafka je vhodný pro:

[5] tj. můžeme přenášet nebo zpracovávat více dat.
[6] Více informací o těchto protokolech najdete v článku o protokolech pro zasílání zpráv [44].

- Práci s velkým množstvím dat. Kafka dokáže zpracovat miliony zpráv za sekundu, a to i s omezenými zdroji. Tohoto výkonu je možné dosáhnout díky horizontální škálovatelnosti. Kafka je také vhodná pro datové proudy
- Práci s Event Sourcing. Například k využití historie transakcí pro přepočet zůstatku na účtu uživatele.

RabbitMQ je vhodný pro:

- Provoz systému vyžadujícího garantované doručování zpráv. To je případ, kdy není důležité pořadí doručení zprávy, ale fakt doručení zprávy.
- Práci se systémem, který vyžaduje flexibilitu a správu zpráv. RabbitMQ umožňuje směrovat, filtrovat a transformovat zprávy. RabbitMQ je ideální pro systémy, jako je správa objednávek a oznámení.

### 4.6.2 Výhody a nevýhody Apache Kafka

Následující výhody a nevýhody jsou převzaty z článku [45].

Výhody Apache Kafka:

- **Vysoká propustnost.** Kafka je schopna zpracovávat velké objemy dat. Díky tomu je Kafka ideální variantou pro systém zaměřený na datový proud v reálném čase.
- **Odolnost vůči poruchám.** Kafka poskytuje vysokou odolnost proti chybám a zaručuje integritu zpráv v případě selhání uzlu.
- **Ukládání dat/zpráv.** Uložená data lze použít k replikaci chování systému až k aktuálnímu času.
- **Distribuovaná architektura.** Díky tomu je Kafka snadno škálovatelná.

Nevýhody Apache Kafka:

- **Žádná kompletní sada monitorovacích nástrojů.** Chybí úplná sada nástrojů pro správu a monitorování. Proto pracovníci podnikové podpory pociťovali obavy nebo strach z volby Kafky a její dlouhodobé podpory.
- **Snížení výkonu.** Obecně neexistují žádné problémy s velikostí jednotlivých zpráv. Brokeři a konzumenti však s rostoucí velikostí začínají tyto zprávy stlačovat. Kvůli tomu se při rozbalování pomalu využívá paměť uzlu. Ke stlačení dochází také při datovém proudu v pipeline. To ovlivňuje propustnost a také výkon.
- **Chybí některá paradigmata pro zasílání zpráv.** V systému Kafka chybí některá paradigmata pro zasílání zpráv, jako je požadavek/odpověď, fronty point-to-point atd. Ne vždy, ale pro určité případy použití to zní problematicky.

### 4.6.3 Výhody a nevýhody RabbitMQ

Následující výhody a nevýhody jsou převzaty z článku [46].
Výhody RabbitMQ:

- **Zaručené doručení zprávy.** RabbitMQ zaručuje doručení zpráv a poskytuje vysokou spolehlivost.
- **Jasné zasílání zpráv.** Odesílá zprávy vhodným konzumentům a frontám. Pomocí svého uživatelského rozhraní lze zjistit, kde a jaká zpráva je uložena.

Nevýhody RabbitMQ:

- Omezený rozsah brokerů, protože je založen na jednom brokerovi.
- Špatná dokumentace.

## 4.7 Závěry

Nelze říci, že Apache Kafka je lepší než RabbitMQ nebo naopak. Každý message broker má své výhody a nevýhody, takže je možné si vybrat podle daného úkolu.

Apache Kafka poskytuje vysokou propustnost a škálovatelnost pro zpracování velkého množství dat v reálném čase. Apache Kafka také udržuje historii zpráv, takže můžeme ověřit, co se stalo v minulosti.

RabbitMQ poskytuje širší sadu funkcí, jako jsou fronty zpráv, výměny a routování zpráv. To je velmi výhodné v aplikacích, kde je třeba zpracovávat mnoho různých typů zpráv a kde se jedná o složitější směrování.

# Kapitola 5
# Závěr

Cílem této práce bylo seznámit se s architekturou řízenou událostmi. Seznámit se s návrhovými vzory a implementovat vzor Transaction Log Tailing. Dalším dílčím cílem práce bylo seznámit se s message brokery a jejich využitím.

Během práce bylo prostudováno několik nových technologií a postupů, jako jsou Docker, Debezium, RabbitMQ, Event Storming, Event Sourcing, Transactional Outbox, Transaction Log Tailing, Change Data Capture, CodeNow.

Jednou z hlavních částí práce bylo studium a porovnání architektury řízené událostmi s jinými známými architekturami, jako jsou monolitická architektura a architektura mikroslužeb. Při studiu architektury řízené událostmi byly rozebrány výhody této architektury, jako jsou nezávislost, agilita a výkonná odezva v reálném čase. Také byly prozkoumány nevýhody této architektury, jako jsou složitost, testování, zkreslení dat a řešení chyb. Poté byly rozebrány výhody a nevýhody architektury řízené událostmi ve srovnání s monolitickou architekturou a architekturou mikroslužeb.

Byly uvedeny důvody, proč je vhodné používat vzory při návrhu aplikací s architekturou řízenou událostmi.Také byly zmíněny nejčastěji používané vzory při návrhu aplikací pomocí architektury řízené událostmi. Podrobně byly probrány vzory Event Storming, Event Sourcing a Transaction Log Tailing.

Další významnou částí práce byla implementace Transaction Log Tailing. Pro Transaction Log Tailing jsem použil Docker s Apache Kafka, Zookeeper, Debezium a databázi PostgreSQL. Moje implementace Transaction Log Tailing zahrnuje jednu službu pro rezervaci jízdenek. Při rezervaci jízdenky vznikne událost NewReservation, která je uložena do tabulky message_outbox v databázi Postgres a ihned z této tabulky je odstraněna. Tabulka message_outbox je sledována Debeziumem, který používá Change Data Capture k sledování změn v databázi. Ačkoli událost NewReservation je v tabulce message_outbox uložena méně než sekundu, Debezium má dostatek času na sledování změny v databázi a odeslání zprávy o nové události do Apache Kafka. Druhá služba přijímá zprávy z Apache Kafka. Tato služba získá z přijaté zprávy informace o události a na základě této události provede určité akce. Vzhledem k tomu, že v mé implementaci existuje pouze jedna událost, moje služba automaticky získává data z přijaté zprávy a na základě těchto dat odesílá zprávu o nové rezervaci na e-mail získaný z přijaté zprávy z Apache Kafka.

Byla zmíněna koncepce message brokerů a princip jejich fungování. Byly podrobně rozebrány Apache Kafka a RabbitMQ. Popisoval jsem princip fungování Apache Kafka a princip fungování RabbitMQ. Pro každého message brokera byly napsány dva příklady. V případě Apache Kafka byl jeden příklad napsán během implementace Transaction Log Tailing, kdy consumer čte zprávy z Apache Kafka, a druhý příklad byl napsán během implementace vlastní mikroslužby pro notifikaci uživatelů o nové rezervaci.

Moje mikroslužba pro notifikace uživatelů byla připojena k Apache Kafka běžící na platformě CodeNow. Služba přijímala zprávy o rezervaci z Apache Kafka prostřednictvím tématu `reservation-mail-notification`. Zpráva obsahovala informace o tom,

komu poslat e-mail, předmět e-mailu a text e-mailu. Pro odesílání e-mailů jsem použil službu Mailgun.

V případě RabbitMQ byly také napsány dva příklady. Jeden příklad ukazoval, jak funguje běžné odesílání a přijímání zpráv v RabbitMQ. Druhý příklad ukazoval, jak funguje odesílání zprávy od klienta k serveru, server zpracovával tuto zprávu a posílal ji zpět. Považoval jsem to za užitečný příklad, protože to může být použito v situaci, kdy je zapotřebí bezpečný a spolehlivý přenos dat mezi klientem a serverem.

Také bylo nutné nahrát veškerý napsaný kód na platformu CodeNow, ale po dohodě s vedoucím práce bylo rozhodnuto, že kód Transaction Log Tailing nebude nahrán na platformu CodeNow, protože vznikl problém s konfigurací Debezium na této platformě. Také po dohodě s vedoucím práce bylo rozhodnuto, že kód s RabbitMQ nebude nahrán, protože na platformě CodeNow nejsou k dispozici zdroje, které by umožnily spuštění brokeru RabbitMQ, přes který by komunikovaly mé mikroslužby.

# Literatura


[1] IBM. *Advantages of the event-driven architecture pattern.* https://developer.ibm.com/articles/advantages-of-an-event-driven-architecture/.

[2] RedHat. *What is event-driven architecture?* https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture.

[3] IBM. *Event-driven solutions for cloud-native architectures.* https://www.ibm.com/cloud/architecture/architectures/eventDrivenArchitecture.

[4] IBM. *What is event-driven architecture?* https://www.ibm.com/topics/event-driven-architecture.

[5] Amazon Web Services. *The benefits of event-driven architectures.* https://docs.aws.amazon.com/lambda/latest/operatorguide/event-driven-benefits.html.

[6] CyberlinkASP. *What is Software Scalability and Why is it Important?* https://www.cyberlinkasp.com/insights/what-is-software-scalability-and-why-is-it-important/.

[7] Twain Taylor. *Event-driven architecture pros and cons: Is EDA worth it?* https://www.techtarget.com/searchapparchitecture/tip/Event-driven-architecture-pros-and-cons-Is-EDA-worth-it.

[8] Chandler Harris. *Microservices vs. monolithic architecture.* https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[9] Google. *What is Microservices Architecture?* https://cloud.google.com/learn/what-is-microservices-architecture.

[10] Amazon. *What is event-driven architecture (EDA)?* https://aws.amazon.com/what-is/eda/.

[11] StackChief LLC. *Event Driven vs REST in Microservice Architecture.* https://www.stackchief.com/blog/Event%20Driven%20vs%20REST%20in%20Microservice%20Architecture.

[12] IBM. *Event-driven patterns?* https://www.ibm.com/cloud/architecture/architectures/eventDrivenArchitecture/patterns/.

[13] Oracle. *What Is a Database?* https://www.oracle.com/database/what-is-database/.

[14] Fauna. *What is a database transaction?* 2021. https://fauna.com/blog/database-transaction.

[15] Chris Richardson. *Microservices Patterns.* Edition 1 vyd.. Manning, 2019. ISBN 9781617294549.

[16] Chris Richardson. *Pattern: Saga.*
https://microservices.io/patterns/data/saga.html.

[17] IBM. *Command Query Responsibility Segregation (CQRS) pattern.*
https://www.ibm.com/cloud/architecture/architectures/event-driven-cqrs-pattern/.

[18] Context Mapper. *Model Event Storming Results in Context Mapper.*
https://contextmapper.org/docs/event-storming/.

[19] Nfroza. *The advantages of event storming .*
https://www.nforza.nl/blog/the-advantages-of-event-storming%20%20%C2%A0.

[20] Miko Lehman. *Event Storming: A New Method of Idea Generation.*
https://www.gmihub.com/blog/event-storming-a-new-method-of-idea-generation/.

[21] Chris Richardson. *Pattern: Transaction log tailing.*
https://microservices.io/patterns/data/transaction-log-tailing.html.

[22] Red Hat. *Getting Started with Debezium.*
https://access.redhat.com/documentation/en-us/red_hat_integration/2021.q1/html-single/getting_started_with_debezium/index.

[23] Microsoft. *What is change data capture (CDC)?*
https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/about-change-data-capture-sql-server?view=sql-server-ver16.

[24] PostgreSQL. *What is PostgreSQL?*
https://www.postgresql.org/about/.

[25] Mailgun. *About Mailgun.*
https://www.mailgun.com/about/company/.

[26] Docker Inc. *Docker overview.*
https://docs.docker.com/get-started/overview/.

[27] The Apache Software Foundation. *Welcome to Apache ZooKeeper.*
https://zookeeper.apache.org/.

[28] MDN contributors. *Working with JSON.*
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON.

[29] RingCentral. *Synchronous Communication.*
https://www.ringcentral.com/gb/en/blog/definitions/synchronous-communication/.

[30] Amir Salihefendic. *What the heck is asynchronous communication anyway?*
https://async.twist.com/asynchronous-communication/.

[31] Amazon Web Services. *What is XML?*
https://aws.amazon.com/what-is/xml/.

[32] TechTarget Contributor. *Apache Kafka Definition.* 2019.
https://www.techtarget.com/whatis/definition/Apache-Kafka.

[33] Gwen Shapira. *Kafka The Definitive Guide Real-Time Data and Stream Processing at Scale.* Edition 2 vyd.. O'Reilly Media, 2022. ISBN 978-1-492-04308-9.

[34] Data Flair. *Kafka Architecture and Its Fundamental Concepts.*
https://data-flair.training/blogs/kafka-architecture/.

[35] Lawrence Williams. *Round Robin Scheduling Algorithm with Example*. 2023. https://www.guru99.com/round-robin-scheduling-example.html.

[36] CodeNow. *What is CodeNOW*. https://docs.codenow.com/what-is-codenow.

[37] Mailgun. *Account Sending Limitation*. https://help.mailgun.com/hc/en-us/articles/115001124753-Account-Sending-Limitation.

[38] Oasis. *AMQP is the Internet Protocol for Business Messaging*. https://www.amqp.org/about/what.

[39] CloudAMQP. *Part 1: RabbitMQ for beginners - What is RabbitMQ?* https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html.

[40] RabbitMQ. https://www.rabbitmq.com/img/tutorials/python-one.png.

[41] ProjectPro. *Kafka vs RabbitMQ - A Head-to-Head Comparison for 2023*. https://www.projectpro.io/article/kafka-vs-rabbitmq/451##mcetoc_1fb64j0f74.

[42] RabbitMQ. *Clients Libraries and Developer Tools*. https://www.rabbitmq.com/devtools.html.

[43] Confluent. *Programming Languages and Tools for Apache Kafka*. https://developer.confluent.io/kafka-languages-and-tools/.

[44] Callum Jackson. *Understanding enterprise messaging APIs and protocols*. 2021. https://developer.ibm.com/articles/messaging-protocols/.

[45] Data Flair. *Advantages and Disadvantages of Kafka*. https://data-flair.training/blogs/advantages-and-disadvantages-of-kafka/.

[46] Emad Bin Abid. *RabbitMQ vs Kafka – Message Brokers (Pros and Cons)*. https://cloudinfrastructureservices.co.uk/rabbitmq-vs-kafka-message-brokers/.

# Příloha A
## Přiložené soubory

Odevzdávané soubory:

| | |
|---|---|
| `TLTReservation.zip` | zdrojové kódy Transaction Log Tailing pro vytvoření rezervace (kap. 3.4) |
| `TLTNotifcation.zip` | zdrojové kódy Transaction Log Tailing pro odesílání notifikací (kap. 3.4) |
| `NotificationForCodeNow.zip` | zdrojové kódy mikroslužby pro zasílání notifikací na platformě CodeNow (kap. 4.3) |
| `KafkaTesting.zip` | zdrojové kódy pro testování Apache Kafka (kap. 4.3.1) |
| `RabbitMQProducer.zip` | zdrojové kódy implementace producenta RabbitMQ (kap. 4.5.1) |
| `RabbitMQConsumer.zip` | zdrojové kódy implementace konzumenta RabbitMQ (kap. 4.5.1) |
| `RabbitMQRPCProducer.zip` | zdrojové kódy implementace producenta RabbitMQ s využitím vzoru RPC (kap. 4.5.2) |
| `RabbitMQRPCConsumer.zip` | zdrojové kódy implementace konzumenta RabbitMQ s využitím vzoru RPC (kap. 4.5.2) |
| `RabbitMQTesting.zip` | zdrojové kódy pro testování RabbitMQ (kap. 4.5.3) |