ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Vývoj cloud native aplikací v praxi

Robin Vávra
Otevřená Informatika, Software

Květen 2022
Vedoucí práce: Ing. Martin Komárek

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vávra** Jméno: **Robin** Osobní číslo: **492293**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vývoj cloud native aplikací v praxi

Název bakalářské práce anglicky:

Cloud Native Application Development

Pokyny pro vypracování:

1.Nastudujte problematiku vývoje cloud native aplikací. Konkrétně témata: DevOps, mikroservisní architektura, 12 faktor principy, Kubernetes&Docker, OpenTraceID, logování, testovací framework Karate, nástroj na zátěžové testy K6.
2. Na plaformě CodeNOW.com vytvořte iterativním způsobem jednoduchou (třívrstvou) ukázkovou aplikaci implementující výše uvedené principy/přístupy. Aplikaci otestujte jednotkovými testy, automatickými testy UI a zátěžovými testy. Ukázová aplikace bude minimálně umožňovat vytvářet, upravovat a zobrazovat například rezervace lístku na vlak.
3. Nastudujte problematiku asynchroní komunikace přes event streaming platformu Apache Kafka.
4. Iterativním způsobem upravte ukázkovou aplikaci tak, aby využívala principy asynchronní komunikace. Aplikaci opět otestujte a porovnejte výsledky testů obou variant aplikace.

Seznam doporučené literatury:

12-factor app. https://12factor.net/.
Apache Kafka. https://kafka.apache.org/.
The Coders' DevOps Value Stream Delivery Platform. https://www.codenow.com/.
k6 load testing tool. https://k6.io/.
Karate testing framework. https://github.com/karatelabs/karate.
Vendor-neutral APIs and instrumentation for distributed tracing. https://opentracing.io/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Komárek katedra informační bezpečnosti FIT

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

Datum zadání bakalářské práce: **02.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

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

III. PŘEVZETÍ ZADÁNÍ

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

. | Datum převzetí zadání

Podpis studenta

Poděkování / Prohlášení

Chtěl bych poděkovat vedoucímu mé práce panu Ing. Martinovi Komárkovi za vymyšlení zajímavého a přínosného tématu a také za potřebné vedení a průběžné konzultace v průběhu semestru.

Dále bych chtěl poděkovat mému kamarádovi Vítovi Šestákovi za pomoc s nejrůznějšími problémy, které mě při práci potkaly.

V neposlední řadě chci pak také poděkovat Zdeňkovi Kratochvílovi za nastínění problematiky asynchronní komunikace a tracingu. Radimovi Bukovskému pak za pomoc s karate frameworkem.

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.

Abstrakt / Abstract

Cílem této práce je přiblížit problematiku vývoje cloud native aplikací. Porovnat rozdíly synchronní a asynchronní komunikace a dále jednotlivých architektur aplikace. Tyto rozdíly pak demonstrovat na jednoduché demo rezervační aplikaci implementované v programovacím jazyce JAVA používající framework SpringBoot.

Klíčová slova: Cloud-Native, SpringBoot, CodeNOW, APACHE Kafka, Tracing, K6

The main goal of this work is to get to know the new trend of developing cloud-native applications. Compare the synchronous and asynchronous communication and also different architectures of the application. These different approaches then demonstrate on basic demo reservation application implemented in JAVA using SpringBoot framework.

Keywords: Cloud-Native, SpringBoot, CodeNOW, APACHE Kafka, Tracing, K6

/ Obsah

1 Úvod 1
2 Teoretická část 2
2.1 CodeNOW 2
2.2 Architektura aplikace 2
2.2.1 Monolitická architektura 3
2.2.2 N-tier architektura 3
2.2.3 Mikroslužby 3
2.3 Hypertext Transfer Protocol (HTTP) 4
2.4 REST API 5
2.5 12-Factor App 5
2.6 Cloud Native 5
2.7 Moderní techniky vývoje 5
2.7.1 DevOps 5
2.7.2 Agilní vývoj 6
2.7.3 CI/CD 6
2.8 Tracing 6
2.9 Testování 6
2.9.1 Unit Testy 6
2.9.2 Karate Framework 7
2.9.3 K6 7
3 Praktická část 8
3.1 Originální aplikace a její řešení 8
3.2 Rozšíření aplikace 11
3.2.1 Přehled rezervací 11
3.2.2 Zrušení rezervace 12
3.2.3 Implementace posílání emailu 16
3.3 Implementace podle architektur 18
3.3.1 Monolit 18
3.3.2 Mikroslužby se synchronní komunikací 18
3.3.3 Mikroslužby s asynchronní komunikací (kafka) 19
3.4 Testování 24
3.4.1 UI testy (Karate Framework) 25
3.4.2 Zátěžové testy (nástroj K6) 26
4 Závěr 31
4.1 Shrnutí 31
4.2 Budoucí práce 32
A Tracing 33
Literatura 39

/ Obrázky

3.1. Úspěšné vytvoření rezervace 9
3.2. Diagram jednotlivých komponent originální aplikace. 10
3.3. Přehled vytvořených rezervací. 11
3.4. Tabulka rozšířená o zrušení rezervace. 13
3.6. Frontend - tlačítko a tabulka. . 13
3.5. Tabulka se zrušenými rezervacemi. 14
3.7. Frontend - získání dat. 14
3.8. Frontend - vykreslení hlavičky tabulky. 15
3.9. Frontend - vykreslení těla tabulky. 15
3.10. Frontend - cancel reservation. . 15
3.11. Mail - MailRequest POJO. 16
3.12. Mail - dependence. 17
3.13. Mail - properties. 17
3.14. Mail - MailConfig. 17
3.15. Mail - sendMail(). 18
3.16. Diagram komponent - monolit. 19
3.17. Diagram komponent - synchronní komunikace. 20
3.19. Kafka - dependence. 20
3.18. Diagram komponent - asynchronní komunikace. 21
3.20. Kafka - config. 21
3.21. Kafka - producer config. 22
3.22. Kafka - producer. 23
3.23. Kafka - config. 23
3.24. Kafka - consumer config. 24
3.25. Kafka - consumer. 24
3.26. Karate - dependence. 25
3.28. Karate - ReservationsTest. 25
3.27. Karate - config. 26
3.29. Karate - complex test. 26
3.30. Karate - create reservation. 27
3.31. Karate - reservation view. 27
3.32. Karate - cancel reservation. 27
3.33. K6 - code. 28
3.34. K6 - monolith. 29
3.35. K6 - synchronous. 29
3.36. K6 - asynchronous. 30

Kapitola 1
Úvod

Cílem práce bylo nastudovat problematiku vývoje cloud native aplikací a s ní spojenými nástroji a praktikami. Poté pomocí nastudovaného materiálu rozšířit demo rezervační aplikaci na platformě CodeNOW [1] [2] o další funkcionality a vytvořit tři různé verze aplikace podle typu architektury.

Kapitola 2
Teoretická část

V této části se seznámíme s nástroji a praktikami souvisejících s vývojem Cloud Native aplikací, jejichž konkrétní využití a implementaci si dále popíšeme v praktické části.

2.1 CodeNOW

CodeNOW je platforma usnadňující vývoj cloud-native aplikací integrací více než 40 open-source nástrojů do jednoho funkčního celku poskytovaného jako služba.[1]

2.2 Architektura aplikace

Architektura aplikace [3] popisuje různé vzory a techniky, kterých by se měli vývojáři držet při návrhu a vývoji aplikace. Tyto konvence a techniky poskytují určitý návod nebo chcete-li cestu, kterou se vydat, jejíž výsledkem by měla být dobře strukturovaná aplikace.

Vzor popisuje opakovatelné řešení pro určitý problém, konkrétně pak návrhové vzory by měly pomoci ke správnému sestavení aplikace. Návrhové vzory samy o sobě neslouží k vytvoření aplikace, fungují spíše jako popis nebo šablona pro postup při řešení daného problému. Návrhové vzory obsahují nejlepší a osvědčené postupy, kterých by se měl programátor držet při navrhování aplikace, aby dospěl k požadovanému výsledku.

Architektura aplikace je tedy jakýsi počáteční bod při vývoji aplikace, ale k samotnému vývoji aplikace jsou ještě zapotřebí další kroky. Prvním takovým krokem je například výběr programovacího jazyka, ve kterém budeme aplikaci vyvíjet. Programovacích jazyků je spousta a je potřeba si zvolit takový programovací jazyk, který odpovídá požadavkům naší aplikace. Pro webové aplikace je dnes jeden z nejpoužívanějších jazyků JavaScript společně s HTML a CSS. K vývoji aplikací na platformy macOS, watchOS a iOS od společnosti Apple slouží Swift. C Sharp pak převážně pro aplikace na platformě Windows od společnosti Microsoft.

Dříve se aplikace psaly jako celistvý kus kódu, kde všechny komponenty přistupovaly ke stejným zdrojům a paměti. Takováto architektura se nazývá monolitická architektura.

Moderní architektury zastávají spíše volně vázaný (loosely coupled) přístup společně s mikroslužbami a rozhraními pro programování aplikací (API), sloužících k připojení různých služeb. Tento přístup je základem pro vývoj cloude-native aplikací.

Architektur aplikace je celá řada, ale mezi ty dnes nejvýznamnější, rozdělené podle míry jejich vázanosti na společné zdroje a funkcionalitu (coupling), patří:

- monolitická a N-tier architektury, silně vázané (tighlty coupled)
- event-driven a service-oriented architektury, volně vázané (loosely coupled)
- mikroslužby, nevázané (decoupled)

2.2.1 Monolitická architektura

Monolitická architektura je architektura, kde je vše od databáze až po uživatelské rozhraní zahrnuto v jednom jediném programu. Monolitická aplikace by měla obsahovat vše potřebné pro vykonání určité funkcionality bez použití jakékoliv vnější služby.

- Výhody

Jednoduchost v počátečních fázích vývoje je jednou z největších výhod monolitické aplikace. Poskytuje snadné sestavení (build), testovaní a nasazení (deploy). Monolitická aplikace skvěle podporuje takzvané cross-cutting concerns. Cross-cutting concerns jsou části programu, které mohou zasahovat do všech vrstev napříč aplikací. Příkladem cross-cutting concerns je například logování nebo performance monitoring. Výkonnost je další výhodou monolitické aplikace. Protože spolu jednotlivé komponenty většinou sdílí společnou paměť, je komunikace mezi nimi rychlejší, než by tomu bylo u komunikace mezi jednotlivými službami.

- Nevýhody

Špatná škálovatelnost, monolitická aplikace se škáluje vertikálně (přidání větší výpočetní síly na hardwarové úrovni) oproti preferované horizontální škálovatelnosti. Sdílený základní kód s sebou nese spoustu nevýhod jako je například snížená spolehlivost, protože změny ovlivňují celou aplikaci a jakákoliv chyba může způsobit pád celé aplikace. Kvůli silným vazbám mezi komponentami a velkému základnímu kódu je nutné při sebemenší změně nasadit celou aplikaci znovu. V neposlední řadě je to takzvaný technology stack. Technology stack je souhrn všech používaných technologií a jejich verzí. Monolitická aplikace musí používat stejnou verzi pro danou technologii napříč celou aplikací, takže změny v používaných technologiích jsou jak časově, tak finančně náročné.

2.2.2 N-tier architektura

N-tier nebo také vícevrstvá architektura rozdělí funkčnost aplikace do více logických vrstev (většinou tři, ale může být i více), které si samy spravují závislosti (dependencies). Vyšší vrstvy mohou používat služby nižších vrstev, nikoliv však naopak (neplatí pro cross-cutting concerns).

- Výhody

Snazší správa jednotlivých vrstev. Jednodušší opětovné využití jednotlivých služeb.

- Nevýhody

Stejně jako u monolitické aplikace se musí kvůli každé změně znovu nasadit celá aplikace. To stejné platí u chyb (může způsobit nefunkčnost celé aplikace).

2.2.3 Mikroslužby

Oproti monolitické silně vázané aplikaci se přístup pomocí mikroslužeb snaží aplikaci rozdělit na co nejmenší vzájemně nezávislé a volně vázané (loosely coupled) komponenty.

- Výhody

Tento přístup umožňuje snadnou horizontální nebo také dynamickou škálovatelnost. Chyba nějaké mikroslužby neovlivňuje ostatní mikroslužby a tedy nezpůsobí pád celé aplikace. Cílem používání mikroslužeb je rychlé dodání produktu ke koncovému zákazníkovi. Nezávislost mikroslužeb umožňuje snadný paralelní vývoj a při změně nějaké

mikroslužby se nemusí znovu nasazovat celá aplikace, ale stačí pouze přenasadit konkrétní mikroslužbu. To má za důsledek šetření času i peněz, které se mohou investovat do samotného vývoje.

- Nevýhody

  Složitost systému s použitím mikroslužeb značně narůstá, neboť fungují nezávisle a jejich propojení a následná komunikace mezi nimi není jednoduchá. Používání cross-cutting concerns jako je například logování nebo měření různých metric (performance, health check, atd.) je další přítěží u aplikace založené na mikroslužbách, neboť nás zajímá aplikace jako celek a proto potřebujeme nějaký systém, který bude tyto informace sbírat z jednotlivých komponent a zobrazovat na jednom místě. Na platformě CodeNOW 2.1 se o logování stará systém Loki.[4]

2.3 Hypertext Transfer Protocol (HTTP)

HTTP [5] je protokol na aplikační úrovni, který slouží ke komunikaci mezi clientem a serverem (HTTP request/response). Tento protokol ke svému přenosu vyžaduje spolehlivé spojení jako je například TCP/IP [6] a nemůže být tedy přenášeno například pomocí UDP. [7] Mezi hlavní vlastnosti HTTP patří také stateless (jednotlivé požadavky na sobě nejsou nijak závislé) a cacheable (ukládání odpovědí získaných ze strany serveru).

- HTTP Request
  - Message Header

    Header se dále dělí na request line a request headers.
    Request line nese informace o URI, verzi protokolu a HTTP metodě.
  - Message Body

    Obsahuje data uložená pro přenos.
- HTTP Response
  - Message Header

    Header se dále dělí na status line a response headers.
    Status line nese informace o verzi protokolu a response status code.
  - Message Body

    Obsahuje data uložená pro přenos.
- HTTP Metody

  Slouží ke specifikování o jaký typ zprávy se bude jednat a jakou práci by měl server vykonávat.
  Zde si uvedeme několik základních typů.
  - GET
  - POST
  - PUT
  - DELETE

2.4 REST API

API reprezentuje application programming interface, neboli rozhraní pro programování aplikací. REST (Representational State Transfer) nebo také RESTful API [8] je takové API dodržující několik kritérií REST architektury rozhraní navržené pro distribuované prostředí. Komunikace probíhá pomocí HTTP protokolu.

- Kritéria
  - Klient-server architektura
  - Stateless, neukládají se žádné informace klienta na straně serveru
  - Cacheable
  - N-vrstvá architektura

2.5 12-Factor App

V dnešní době je moderním trendem poskytovat software jako službu. 12-factor app [9] je koncept popisující vlastnosti takové aplikace a pravidla při jejím vývoji.

- Vlastnosti definující 12-factor aplikaci
  - Deklarativní přístup pro nastavení automatizace
  - Vhodné pro nasazení na moderní cloud platformy
  - Maximální přenositelnost mezi prostředími
  - Minimální rozdíly mezi vývojovým a produkčním prostředím
  - Snadná škálovatelnost

2.6 Cloud Native

Cloud-native [10] architektura je moderní přístup k vývoji aplikací běžících v cloudu poskytujících maximální využití možností, které cloud platformy nabízí.

- Výhody cloudu
  - Flexibilita a škálovatelnost
  - Bezpečnost, cloud zaručuje bezpečnost uložených dat
  - Snazší spolupráce a komunikace
  - Bezvýpadkovost
  - Připojení odkudkoliv

2.7 Moderní techniky vývoje

Všechny tyto techniky jsou spolu úzce provázané a popisují praktiky usnadňující vývoj a dodání aplikace. [11]

2.7.1 DevOps

Na místo rozdělení vývojářů do specializovaných rolí se DevOps přístup snaží zbourat tyto pomyslné bariéry. DevOps tým pak má na starost celý životní cyklus aplikace.

2.7.2 Agilní vývoj

Agilní vývoj je technika podobná DevOps přístupu, ale na úrovni procesů. Zbourání procesních bariér tak umožňuje úzkou spolupráci mezi vývojáři a zákazníky a tím zrychlený vývoj aplikace společně se zaváděním nových změn.

2.7.3 CI/CD

Tyto techniky se zaměřují na definování životního cyklu aplikace.

- Continuous Integration (CI)

Continuous integration je technika vývoje, která preferuje malé a časté změny, po kterých následuje automatické sestavení a otestování aplikace.

- Continuous Delivery (CD)

Continuous delivery je praktika následující po continuous integration. Zabývá se už samotným zabalením, nasazením a dodáním aplikace zákazníkovi.

2.8 Tracing

Tracing[12] je jednoduchý nástroj, pomocí kterého jde vizualizovat komunikaci mezi jednotlivými komponentami, jak jsou volány při zpracovávání nějakého požadavku. Při obdržení nového požadavku se požadavku automaticky vygeneruje a přiřadí unikátní trace ID. Poté, co komponenta obdrží požadavek, vytvoří se nový span, který se přiřadí dané komponentě a tento span je přidán do výsledného trace. Všechny tyto trace jsou pak odesílány do jaeger collectoru. [13] Jaeger collector pak slouží k zobrazování všech volání pro konkrétní požadavek pomocí již zmiňovaného trace ID. Tracing slouží jak k vizualizaci volání, tak může být použit k debugování aplikace.

2.9 Testování

Vývoj aplikace je složitý a nákladný proces, na kterém se podílí více lidí zároveň. Protože kód píše člověk a lidé nejsou bezchybní, je potřeba správný chod aplikace nějak kontrolovat a předcházet jejím případným pádům. Přesně pro tyto účely slouží testy. Testy nezaručují bezchybovost aplikace, ale výrazně přispívají k odhalovaní chyb. Odhalení chyb již v ranném vývoji aplikace je klíčové pro samotný úspěch celé aplikace, neboť čím dříve se chyby odhalí, tím méně nákladná je pak jejich oprava. Testovacích nástrojů a praktik je celá řada, ale pro účely této bakalářské práce si zmíníme pouze unit testy a testovací nástroje Karate a K6.

2.9.1 Unit Testy

Unit testy nebo také jednotkové testy jsou nástrojem na otestování funkcionality jednotlivých kusů kódu. Unit testy většinou slouží k otestování nějaké složitější funkce. Výsledky testovaných funkcí se pak porovnávají s očekávanými výsledky. Tyto testy se mohou spouštět po každé změně v kódu. Je tedy vhodné tyto unit testy zahrnout jako součást buildu a tím předcházet nejrůznějším chybám. [14]

2.9.2 Karate Framework

Karate framework [15] je nástroj kombinující několik druhů testování dohromady. Karate framework umožňuje testování od API a UI až po mockování nebo zátěžové testy. Testy mohou běžet paralelně a tím zrychlit celkový čas testování. Karate testy se používají například i při testování platformy CodeNOW nebo slouží jako scripty k vytvoření a nasazení nové kopie této demo rezervační aplikace na nová prostředí v rámci CodeNOW.

2.9.3 K6

K6 je nástroj na zátěžové testování. Zátěžové testování je využíváno k otestování spolehlivosti a výkonu aplikace. Tyto zátěžové testy simulují velký nápor uživatelů v jeden čas, díky čemu mohou vývojáři otestovat svoji aplikaci už v raném vývoji a tím předcházet nechtěným a neočekávaným pádům aplikace při větší zátěži. [16]

Kapitola 3
Praktická část

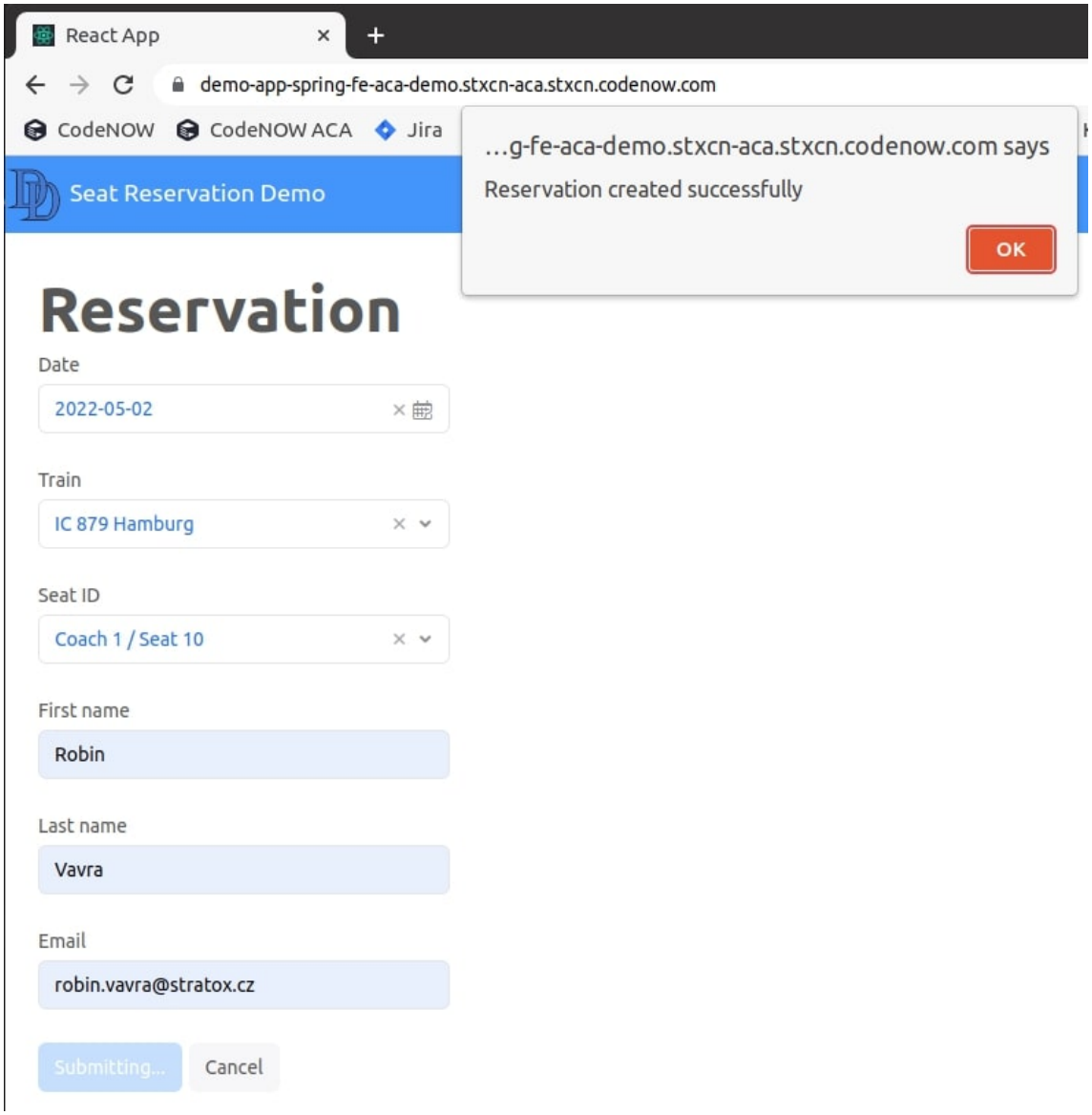
Praktická část se zabývá rozšířením demo rezervační aplikace [17] o funkcionalitu přehled všech vytvořených rezervací a zrušení již vytvořené rezervace s použitím praktik a nástrojů popsaných v teoretické části. Demo rezervační aplikace je rezervační systém jízdenek sloužící k vytváření, ukládání a rušení rezervací pro vlakové spoje. Na této rezervační aplikaci si demonstrujeme rozdíl mezi monolitní, mikroservisní synchronní a mikroservisní asynchronní (cloud native) architekturou. Kromě samotných architektur se hlavně budeme zabývat samotnou implementací, protože rozdílné architektury s sebou přináší i rozdílné přístupy ve vývoji. Cloud native přístup nám sice přináší spoustu výhod, ale implementace cloud native aplikací je výrazně složitější než tomu tak je třeba u monolitní aplikace. Komunikace mezi jednotlivými mikrolužbami je právě jedním z hlavních příkoří při vývoji.

Jak je uvedeno v zadání, práce probíhala iterativním způsobem, proto je i tato kapitola rozdělená na jednotlivé iterativní kroky, jak implementace postupně probíhala. Nejprve si však popíšeme původní aplikaci, ze které jsem vycházel a postupně rozšiřoval.

3.1 Originální aplikace a její řešení

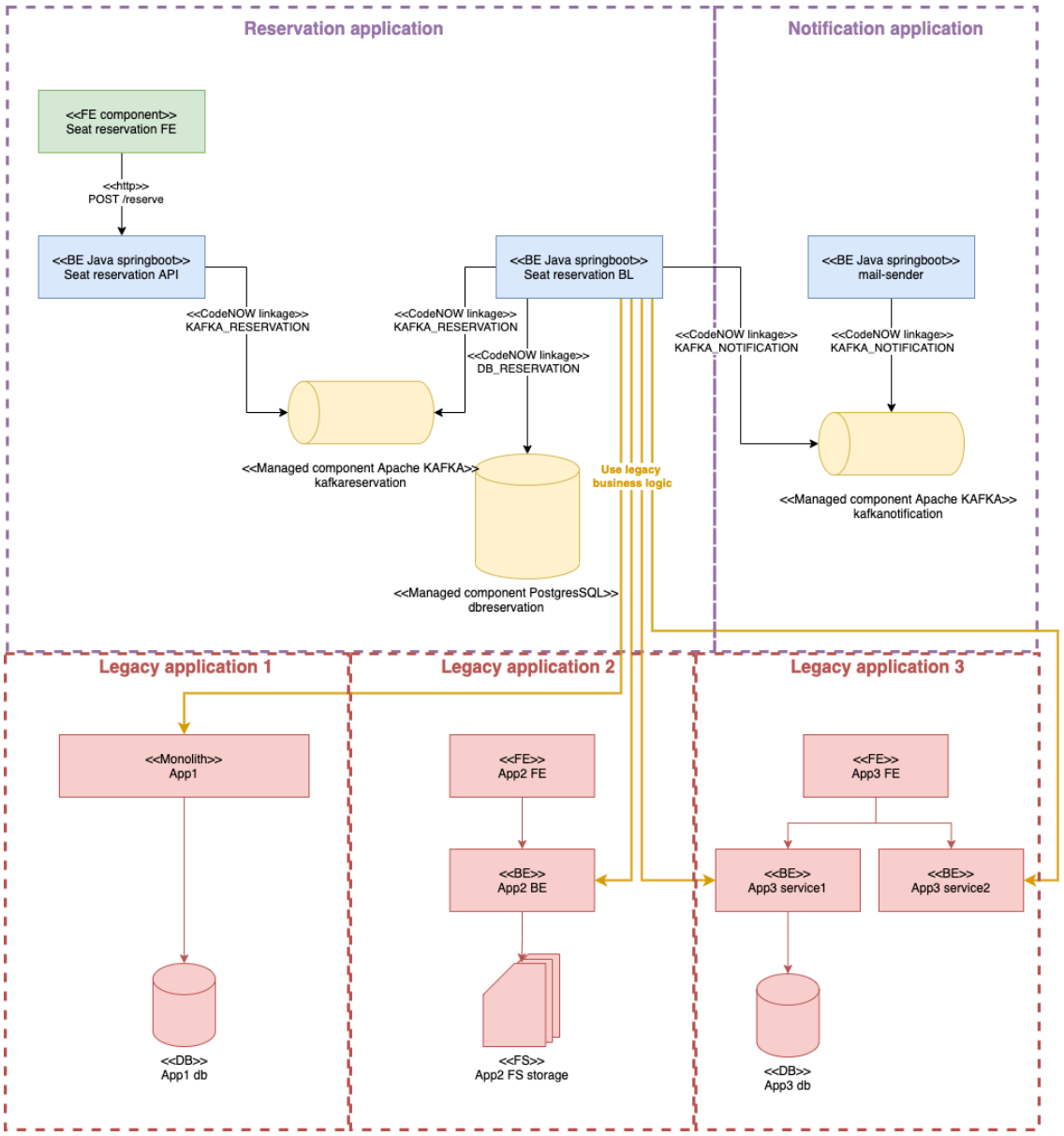
Původní demo rezervační aplikace uměla pouze vytvářet nové rezervace a ukládat je do databáze. Uživatel vyplní formulář obsahující číslo vlaku, číslo sedadla, jméno, příjmení a email. Po vyplnění formuláře klikne na tlačítko `Submit` a tím odešle požadavek s vyplněnými daty pro vytvoření rezervace na API komponentu. API komponenta obdrží požadavek a přepošle ho pomocí kafka producera dál přes kafku na příslušný topic. Na tento topic je připojená backend komponenta jako kafka consumer, který zpracovává zprávy z daného topicu. Backend komponenta je připojená k PostgreSQL databázi, kam ukládá vytvořené rezervace. Backend komponenta tedy obdrží požadavek na vytvoření rezervace, vezme obdržená data a uloží je do databáze. Po uložení zprávy do databáze se zpráva následně přeposílá na MailSender komponentu opět pomocí kafky. MailSender komponenta má za úkol odeslat email s potvrzením o úspěšném vytvoření rezervace. Tato funkcionalita však nebyla implementovaná, neboli MailSender zprávu zpracoval z příslušného topicu, ale dál s ní nic nedělal.

Zpracováním zprávy kafka consumerem dostane kafka producent, který zprávu odesílal, acknowledgment, že zpráva byla zpracována. Jakmile API komponenta tento acknowledgment obdrží, pošle potvrzení zpět na frontend a ten uživateli zobrazí alert se zprávou `Reservation created successfully`, viz Obrázek 3.1.



Obrázek 3.1. Úspěšné vytvoření rezervace

- Architektura
  - Celá aplikace se skládá ze dvou aplikací - Rezervační aplikace a Notifikační aplikace
  - Rezervační aplikace se dělí na tři komponenty - Frontend (React), API (Spring-Boot) a Backend (SpringBoot)
  - Notifikační aplikaci tvoří jedna komponenta - MailSender (SpringBoot)
  - Komunikace mezi jednotlivými komponentami je zprostředkovaná pomocí Kafky[18]
  - K ukládání rezervací se používá PostgreSQL databáze



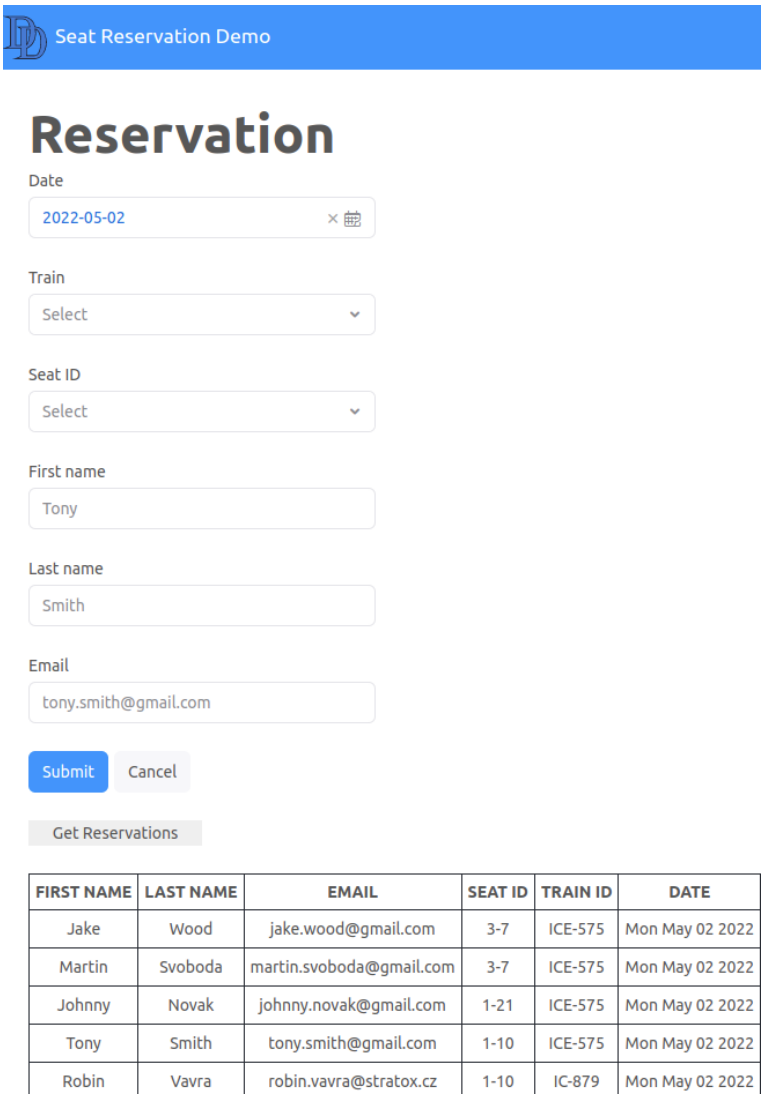
Obrázek 3.2. Diagram jednotlivých komponent originální aplikace.

3.2 Rozšíření aplikace

Protože pro demonstraci rozdílu mezi jednotlivými přístupy bylo pouhé ukládání rezervací málo, rozhodli jsme se rozšířit aplikaci ještě o přehled všech rezervací a dále o jejich zrušení.

3.2.1 Přehled rezervací

Pro přehled rezervací jsme na frontendu museli přidat tlačítko `Get Reservations`, které po kliknutí odešle `GET()` požadavek a čeká na obdržení potřebných dat. Pokud se mu nepodaří získat potřebná data, tedy po cestě nastane nějaká chyba, zobrazí se uživateli alert se zprávou `Failed to get reservation view`. Při obdržení dat se uživateli pod tlačítkem Get Reservations vykreslí tabulka s vytvořenými rezervacemi, viz Obrázek 3.3.



Obrázek 3.3. Přehled vytvořených rezervací.

Implementace získání potřebných dat z databáze se mezi jednotlivými architekturami nijak neliší. Získání všech rezervací z databáze je snadné, neboť k přístupu do databáze používáme repository pattern. Stačí tedy pouze zavolat funkci `findAll()` a ta nám vrátí všechny uložené rezervace v databázi, které pak pošleme zpět na frontend. Implementaci

zobrazování rezervací na frontendu si ukážeme u zrušení rezervace, neboť jsou tyto dvě funkcionality úzce spjaty.

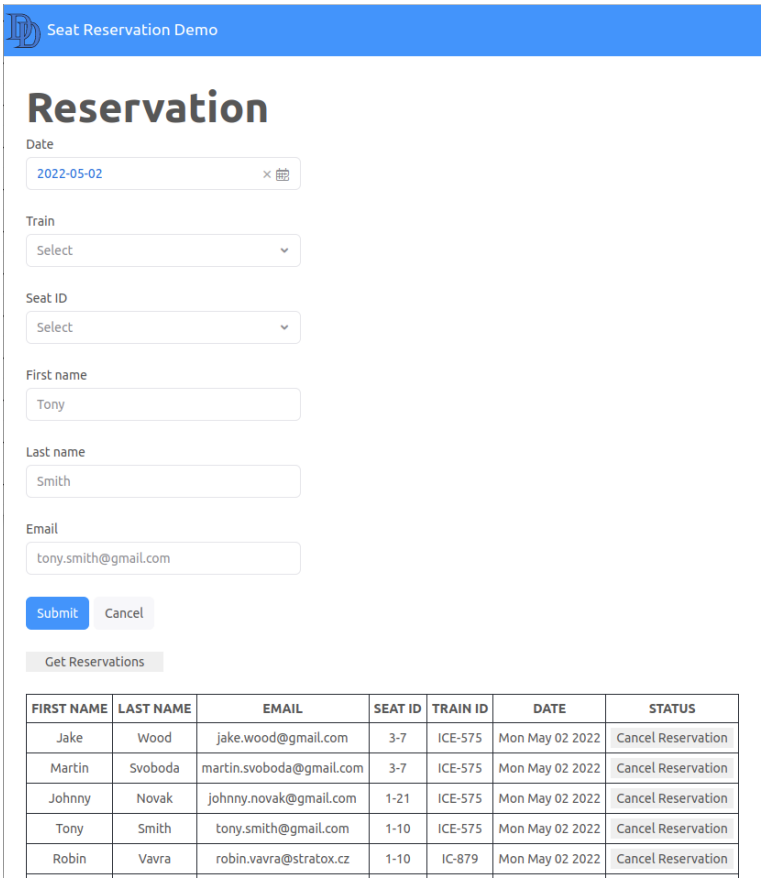
Co však je složité je implementace `GET()` požadavku u asynchronní komunikace v našem případě pomocí kafky. Producer totiž přes kafku pošle příslušný požadavek a dále jen čeká na potvrzení, že požadavek zpracoval nějaký consumer na druhé straně kafky. K poslání potřebných dat zpět by tedy bylo zapotřebí přidat další topic, do kterého by backend komponenta připojená k databázi posílala zprávy s potřebnými daty. Zde ale narážíme na další problém, protože backend komponenta by sice poslala data do příslušného topicu, jenže by nebylo jasné, jaký consumer (API komponenta) na druhé straně topicu má danou zprávu zpracovat a odeslat zpět uživateli, neboť podů API komponenty může naráz běžet několik.

Při řešení tohoto problému jsme se tedy rozhodli nechat implementaci přehledu všech rezervací u každé verze architektury synchronní, protože byl tento problém nad rámec rozsahu této bakalářské práce. Jedním z možných řešení tohoto problému by mohlo být použití například CQRS patternu[19]

3.2.2 Zrušení rezervace

Zrušení již vytvořené rezervace byl po přehledu všech rezervací dalším krokem při rozšiřování aplikace. Abychom mohli nějakou rezervaci zrušit, museli jsme přidat další atribut entity reprezentující rezervaci. Přidaný atribut je třída enum. Tato třída má dvě konstanty `ACTIVE` a `CANCELED`, které reprezentují stav rezervace. Přidání nového atributu k entitě však znamenalo narušení struktury původní databáze. Protože demo rezervační aplikace sloužila pouze jako ukázka a význam již uložených dat byl nulový, byl nejjednodušší způsob smazat všechny záznamy a upravit strukturu položek přímo v databázi.

Přídání nového atributu znamenalo zásah i do vytváření a přehledu všech rezervací. Při vytváření rezervace jsme totiž museli nastavit status rezervace na `ACTIVE`, abychom ho později při případném zrušení mohli nastavit na `CANCELED`. Pro přehled všech rezervací to znamenalo pouze sjednotit reprezentaci entity s reprezentací DTO[20] napříč všemi komponentami, aby se mohly jednotlivé atributy správně namapovat. Na frontend komponentě pak přidat další sloupeček do tabulky, který zobrazuje rezervace. Samotné zrušení konkrétní rezervace uživatelem je řešeno právě pomocí přidaného sloupečku do tabulky. Pokud je rezervace ve stavu `ACTIVE`, zobrazí se ve sloupečku STATUS tlačítko `Cancel Reservation`. Pokud uživatel na toto tlačítko klikne, odešle se `PUT()` požadavek na zrušení rezervace společně s identifikačním číslem rezervace. Protože pro zrušení rezervace si musel uživatel nejprve zobrazit tabulku s uloženými rezervacemi, viz Obrázek 3.4, tak se po úspěšném zrušení rezervace opět ihned volá zobrazení rezervací a tím se zobrazí aktuální tabulka s již zrušenou rezervací, viz Obrázek 3.5.

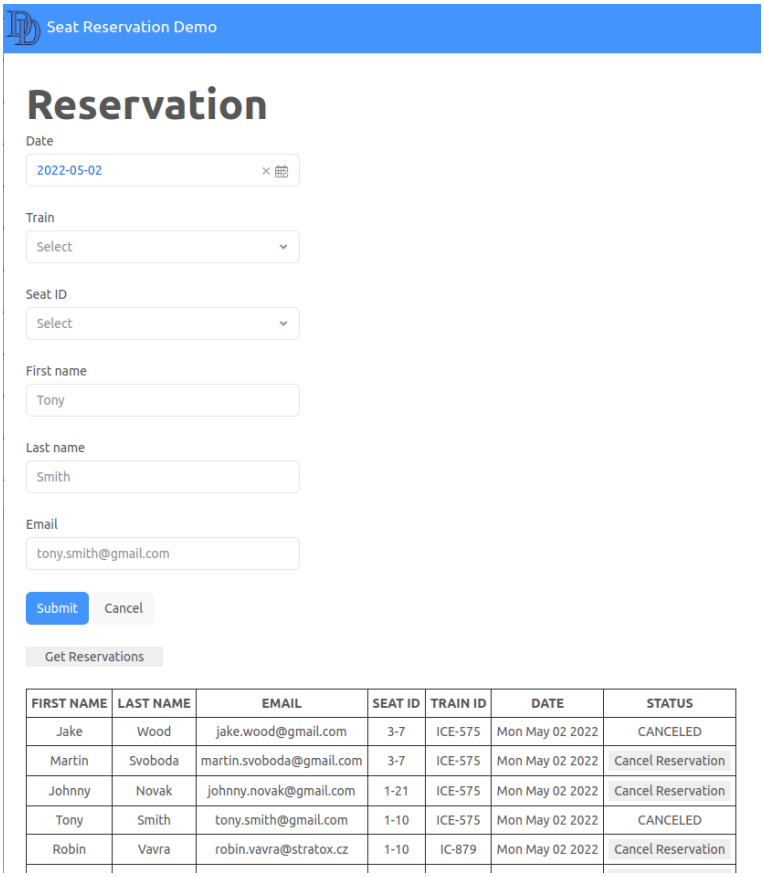


Obrázek 3.4. Tabulka rozšířená o zrušení rezervace.

Jak bylo zmíněno v sekci 3.2.1, zde si ukážeme implementaci přehledu rezervací a zrušení rezervace na frontendu. O vše se stará funkce `ReservationsView`, která zobrazuje tlačítko `Get Reservations` a tabulku s přehledem rezervací, viz Obrázek 3.6.

```
return (
    <div className={"App-reservations-view"}>
        <button style={{margin: "24px 0", width: "150px"}} onClick={() => getList()}>
            Get Reservations
        </button>
        <table style={{textAlign: "center"}} className={"App-reservations-view-table"}>
            <thead className={"App-reservations-view-table"}>
            <tr>
                {renderHeader()}
            </tr>
            </thead>
            <tbody className={"App-reservations-view-table"}>
            {renderBody()}
            </tbody>
        </table>
    </div>
);
```

Obrázek 3.6. Frontend - tlačítko a tabulka.



Obrázek 3.5. Tabulka se zrušenými rezervacemi.

Po kliknutí na tlačítko `Get Reservations` se volá funkce `getList()`, která pošle `GET()` požadavek, viz Obrázek 3.7, čímž získá data a ty následně vykreslí do tabulky pomocí funkcí `renderHeader()`, viz Obrázek 3.8, a `renderBody()`, viz Obrázek 3.9.

```
const getList = () => {
    axios
        .get( url: `${window.env.BACKEND_URL}/reservation`)
        .then((response :AxiosResponse<any> ) => {
            console.log("Response", response);
            setReservations(response.data);
            alert("Get reservations view is successful");
        })
        .catch((error) => {
            console.error("Error", error);
            alert("Failed to get reservation view");
        })
};
```

Obrázek 3.7. Frontend - získání dat.

```
const renderHeader = () => {
    let headerElement = ['first Name', 'last Name', 'email', 'seat Id', 'train Id', 'date', 'status']
    return reservations.length > 0 && headerElement.map((key :string , index :number ) => {
        return <th className={"App-reservations-view-table"} key={index}>{key.toUpperCase()}</th>
    })
}
```

Obrázek 3.8. Frontend - vykreslení hlavičky tabulky.

```
const renderBody = () => {
    console.log(reservations);
    return reservations.length > 0 && reservations.map(({id, firstName, lastName, email, seatId, trainId, date, status}, index :number ) => {
        if(status === "ACTIVE"){
            return (
                <tr key={index}>
                    <td className={"App-reservations-view-table"}>{firstName}</td>
                    <td className={"App-reservations-view-table"}>{lastName}</td>
                    <td className={"App-reservations-view-table"}>{email}</td>
                    <td className={"App-reservations-view-table"}>{seatId}</td>
                    <td className={"App-reservations-view-table"}>{trainId}</td>
                    <td className={"App-reservations-view-table"}>{new Date(date).toDateString()}</td>
                    <td className={"App-reservations-view-table"}><button onClick={() => CancelReservation(id)}> Cancel Reservation</button></td>
                </tr>
            )}
        else {
            return (
                <tr key={index}>
                    <td className={"App-reservations-view-table"}>{firstName}</td>
                    <td className={"App-reservations-view-table"}>{lastName}</td>
                    <td className={"App-reservations-view-table"}>{email}</td>
                    <td className={"App-reservations-view-table"}>{seatId}</td>
                    <td className={"App-reservations-view-table"}>{trainId}</td>
                    <td className={"App-reservations-view-table"}>{new Date(date).toDateString()}</td>
                    <td className={"App-reservations-view-table"}>{status}</td>
                </tr>
            )
        }
    })
}
```

Obrázek 3.9. Frontend - vykreslení těla tabulky.

Pro zrušení rezervace slouží tlačítko `Cancel Reservation` ve sloupečku `status`. Po jeho kliknutí se zavolá funkce `CancelReservation()`, která pošle `PUT()` požadavek pro zrušení rezervace s příslušným `id`. Po obdržení odpovědi zobrazí alert se zprávou `Reservation canceled successfully` a následně volá funkci `getList()` pro obnovení tabulky, viz Obrázek 3.10.

```
const CancelReservation = (id) => {
    console.log(id);
    axios.put( url: `${window.env.BACKEND_URL}/reservation/cancel/${id}`)
        .then(res => {
            console.log(res);
            if(res){
                alert("Reservation canceled successfully.");
            } else {
                alert("Canceling reservation failed.")
            }
            getList();
        });
};
```

Obrázek 3.10. Frontend - cancel reservation.

3.2.3 Implementace posílání emailu

Dalším krokem po přehledu rezervací a zrušení rezervace bylo naimplementovat chybějící posílání emailu při vytvoření a zrušení rezervace. Prvním krokem bylo vytvořit nový emailový účet pro tyto účely demo rezervační aplikace. Když byl účet vytvořený následovala samotná implementace[21].

Nejprve si vytvoříme MailRequest POJO[22] jako jednoduchý objekt, obsahující potřebné atributy k odeslání emailu, viz Obrázek 3.11.

```
/// template of a mail request
/// POJO
9 usages  Vávra, Robin +1
public class MailRequest {

    1 usage
    @Getter
    @Setter
    private String to;
    1 usage
    @Getter
    @Setter
    private String from;
    1 usage
    @Getter
    @Setter
    private String subject;
    1 usage
    @Getter
    @Setter
    private String body;

    2 usages  Robin Vavra
    public MailRequest() {}

    Vávra, Robin
    public MailRequest(String to, String from, String subject, String body) {
        this.to = to;
        this.from = from;
        this.subject = subject;
        this.body = body;
    }
}
```

Obrázek 3.11. Mail - MailRequest POJO.

Dále se musí do projektu přidat potřebná dependence, viz Obrázek 3.12. Po přidání dependence následuje přidání potřebných konfigurací do souboru `application.yaml`, viz Obrázek 3.13. Tyto konfigurace definované v souboru `application.yaml` následně použijeme v třídě `MailConfig` s anotací `@Configuration`, konkrétně při vytváření objektu `JavaMailSender` s anotací `@Bean`, viz Obrázek 3.14.

```xml
<!--Mail-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

Obrázek 3.12. Mail - dependence.

```yaml
mail:
  host: smtp.gmail.com
  port: 587
  username: demoreservationappemail@gmail.com
  password: Monolithapp
  properties:
    mail:
      smtp:
        auth: true
        starttls:
          enable: true
          required: true
      debug: true
```

Obrázek 3.13. Mail - properties.

```java
@Configuration
public class MailConfig {
    Vávra, Robin
    @Bean
    public JavaMailSender getMailSender(@Value("${spring.mail.host}") String host, @Value("${spring.mail.port}") String port,
                                        @Value("${spring.mail.username}") String username, @Value("${spring.mail.password}") String password) {
        JavaMailSenderImpl mailSender = new JavaMailSenderImpl();

        mailSender.setHost(host);
        mailSender.setPort(Integer.parseInt(port));
        mailSender.setUsername(username);
        mailSender.setPassword(password);

        Properties javaMailProperties = new Properties();
        javaMailProperties.put("mail.smtp.starttls.enable", "true");
        javaMailProperties.put("mail.smtp.auth", "true");
        javaMailProperties.put("mail.transport.protocol", "smtp");
        javaMailProperties.put("mail.debug", "true");

        mailSender.setJavaMailProperties(javaMailProperties);
        return mailSender;
    }
}
```

Obrázek 3.14. Mail - MailConfig.

Díky anotaci `@Bean` pak můžeme objekt `JavaMailSender` injectovat při vytváření třídy `MailServiceImpl`. Tento injectovaný objekt použijeme nejprve pro vytvoření `MimeMessage`, do které pomocí `MimeMessageHelper` nastavíme potřebné atributy emailu. Tyto atributy emailu jsou odesílatel, příjemce, předmět a samotný text. Po nastavení těchto atributů zavoláme funkci `send()` starající se o odeslání emailu, viz Obrázek 3.15.

```
1 usage   Vávra, Robin
@Override
public void sendEmail(MailRequest mail) {

        MimeMessage mimeMessage = mailSender.createMimeMessage();
        Span span = tracingHelper.createClientSpan( remoteServiceName: "Mail Sender",  type: "service", this.getClass().getName());
        try {
            log.info("Sending email.");
            MimeMessageHelper mimeMessageHelper = new MimeMessageHelper(mimeMessage,  multipart: true);

            mimeMessageHelper.setSubject(mail.getSubject());
            mimeMessageHelper.setFrom(new InternetAddress(emailFrom,  personal: "Demo Reservation App"));
            mimeMessageHelper.setTo(mail.getTo());
            mimeMessageHelper.setText(mail.getBody());

            mailSender.send(mimeMessageHelper.getMimeMessage());
            log.info("Email was send.");
        } catch (MessagingException | UnsupportedEncodingException e) {
            log.error("Email sending failed!");
            e.printStackTrace();
        }
        span.end();
    }
```

Obrázek 3.15. Mail - sendMail().

3.3 Implementace podle architektur

V kapitole 3.2 jsme se věnovali implementaci jednotlivých funkcionalit. V této kapitole se už nebudeme zabývat implementací, ale popíšeme si rozdíly v architektuře a dále implementaci komunikace mezi jednotlivými komponentami.

3.3.1 Monolit

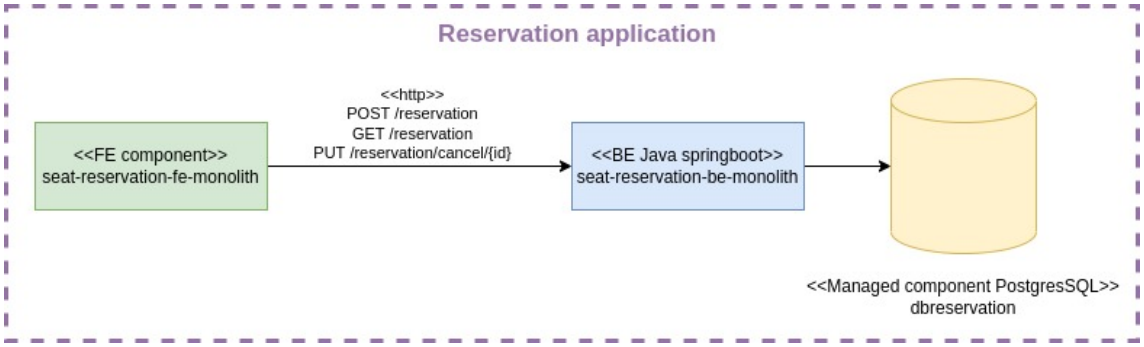
V kapitole 2.2.1 je zmíněno, že monolitická aplikace je jeden jediný program, který se stará o vše. Od tohoto přístupu jsme upustili a rozhodli jsme se aplikaci rozdělit na frontend a backend, protože uplný monolit se dnes již takřka nepoužívá. Jelikož je frontend u všech verzí architektur stejný, přiblížíme si pouze implementaci backend komponenty.

Pro komunikaci s frontendem je vystavený REST controller a komunikace mezi komponentami je synchronní. Backend komponenta je opět připojená k PostgreSQL databázi. Při vytvoření/zrušení rezervace se odešle notifikace o jejím úspěšném vytvoření/zrušení.

- Architektura 3.16
  - Jedna aplikace
  - Dvě komponenty - Frontend (React) a Backend (SpringBoot)

3.3.2 Mikroslužby se synchronní komunikací

Aplikace již není rozdělená pouze na dvě komponenty, ale dostáváme se k mikroslužbám. Komunikace mezi jednotlivými komponentami je stejně jako u monolitu synchronní. Oproti monolitu jsme však pro odesílání emailu vytvořili vlastní aplikaci `MailSender`, která jak jde odvodit ze samotného názvu stará o odesílání emailu. Dále jsme k rezervační aplikaci přidali API komponentu, jak to bylo u originální verze, viz kapitola 3.1.



Obrázek 3.16. Diagram komponent - monolit.

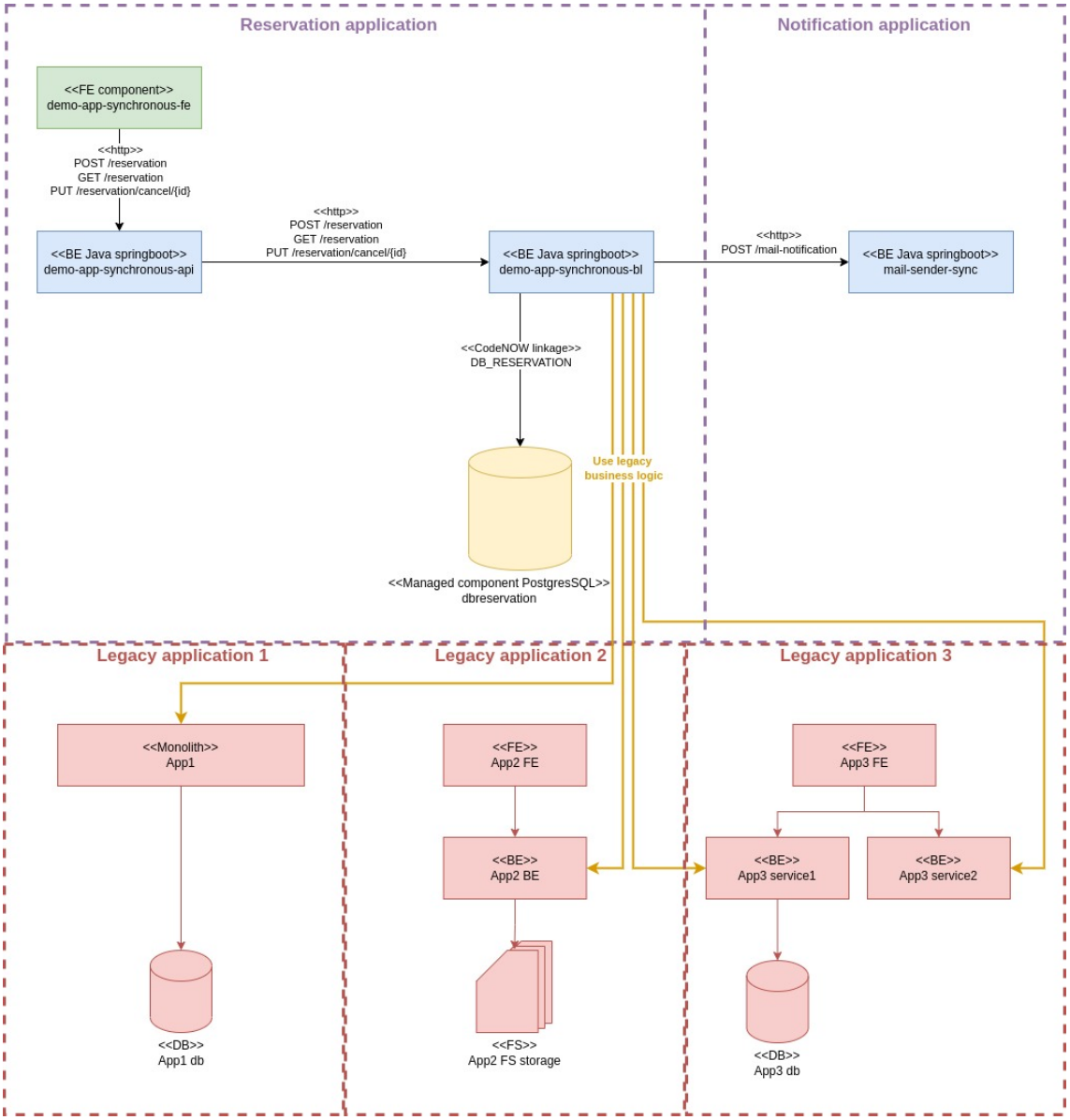
Tato komponenta zatím nemá žádnou funkci, protože jediné o co se stará je přeposílání požadavků obdržených z frontendu na backend a slouží tedy pouze pro ukázkové účely rozdílných architektur.

Při vytvoření/zrušení rezervace se o odesílání emailu již nestará backend, ale vytvoří `EmailRequest` a pošle požadavek na MailSender o jeho odeslání. Po úspěšném odeslání emailu se vrací odpovědi zpět ke klientovi.

- Architektura 3.17
  - Celá aplikace se skládá ze dvou aplikací - Rezervační aplikace a Notifikační aplikace
  - Rezervační aplikace se dělí na tři komponenty - Frontend (React), API (SpringBoot) a Backend (SpringBoot)
  - Notifikační aplikaci tvoří jedna komponenta - MailSender (SpringBoot)
  - Komunikace mezi jednotlivými komponentami je synchronní pomocí REST controllerů[23].
  - K ukládání rezervací se používá PostgreSQL databáze

3.3.3 Mikroslužby s asynchronní komunikací (kafka)

Poslední verzí aplikace z pohledu architektury je demo rezervační aplikace komunikující asynchroně pomocí kafky. Protože architektura asynchronní verze se oproti synchronní verzi, viz kapitola 3.3.2, liší pouze v komunikaci mezi některými komponentami, pro pochopení stačí samotný diagram, viz Obrázek 3.18.



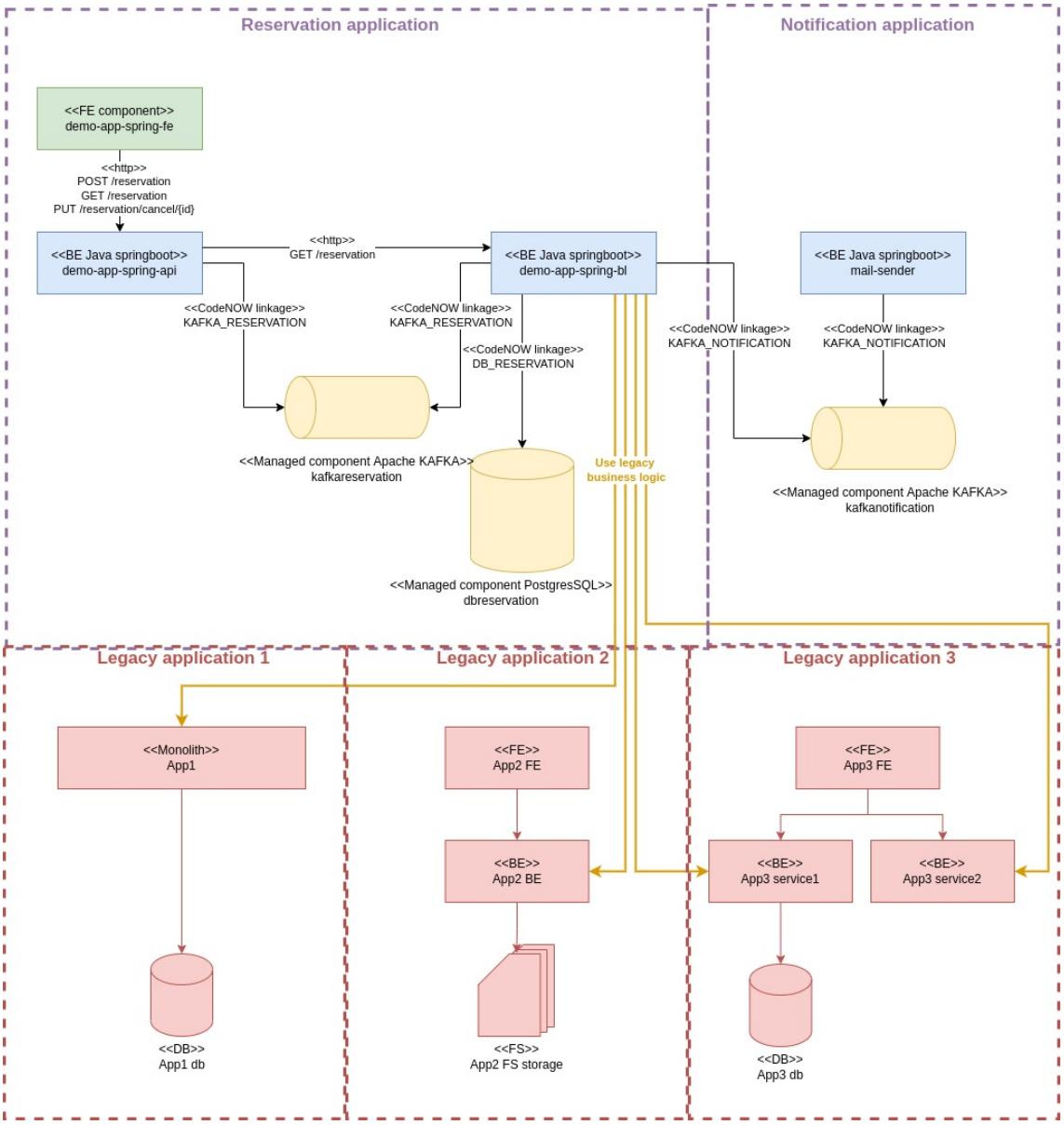
Obrázek 3.17. Diagram komponent - synchronní komunikace.

Hlavním tématem této kapitoly nebude ovšem popis architektury, ale implementace kafka producera a kafka consumera pro asynchronní komunikaci mezi komponentami.

Abychom mohli použít kafku v naší aplikaci, musíme nejprve přidat příslušnou dependenci do souboru `pom.xml`, viz Obrázek 3.19.

```
<!--Kafka-->
    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka</artifactId>
    </dependency>
```

Obrázek 3.19. Kafka - dependence.



Obrázek 3.18. Diagram komponent - asynchronní komunikace.

Dále přidáme potřebné konfigurace do souboru `application.yaml`, viz Obrázek 3.20. `Kafka.bootstrap-servers: ${KAFKA_RESERVATION_BOOTSTRAP_SERVERS}` je environment variable, za kterou se při nasazení vloží URL kafka brokeru. `Kafka.topic` slouží k vytvoření topicu, na který se pak připojí kafka producer a kafka consumer.

```yaml
spring:
  kafka:
    bootstrap-servers: ${KAFKA_RESERVATION_BOOTSTRAP_SERVERS}
    topic:
      new-reservation: new-ticket-reservation
      cancel-reservation: cancel-ticket-reservation
```

Obrázek 3.20. Kafka - config.

- Kafka producer

Pro vytvoření kafka producera musíme nejprve vytvořit třídu `ProducerConfig` s anotací `@Configuration`. Funkce `producerFactory()`[24] vrátí instanci kafka producera, který pomocí `kafkaTemplate`[25] metody pak může vykonávat high-level operace (KafkaOperations[26]), jako je například funkce `send()` pro odeslání zprávy na určitý topic, viz Obrázek 3.21.

```
Vávra, Robin +1 *
@Configuration
public class ProducerConfig {
    1 usage
    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServers;
    1 usage   Robin Vavra +1
    @Bean
    public <K, V> ProducerFactory<K, V> producerFactory(){
        Map<String,Object> config = new HashMap<>();
        config.put(org.apache.kafka.clients.producer.ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        config.put(org.apache.kafka.clients.producer.ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
        config.put(org.apache.kafka.clients.producer.ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
        return new DefaultKafkaProducerFactory(config);
    }
    Robin Vavra +1
    @Bean
    public <K, V> KafkaTemplate<K, V> kafkaTemplate(){
        return new KafkaTemplate<>(producerFactory());
    }
}
```

Obrázek 3.21. Kafka - producer config.

Když máme naimplementovanou třídu `ProducerConfig`, následuje implementace konkrétních producerů. Zde si ukážeme vytváření třídy `CancelReservationProducer`. Vytváření třídy `NewReservationProducer` si ukazovat nemusíme, neboť se liší pouze v posílaném objektu. Při vytváření třídy `CancelReservationProducer` si injectujeme dříve vytvořenou `@Bean KafkaTemplate` ve třídě `ProducerConfig` a hodnotu definovanou v souboru `application.yaml` pro název topicu. Funkce `cancelReservation(Integer id)` jenom zavolá operaci `send()` na injectované `KafkaTemplate`, kde jako argumenty jsou název topicu a id rezervace pro zrušení, čímž odešle zprávu, viz Obrázek 3.22.

- Kafka consumer

Vysvětlilli jsme si, jak se implementuje kafka producer a posílají zprávy na příslušný topic. Zde si vysvětlíme jak vytvořit kafka consumera a zprávy z topicu odebírat.

Dependence zůstává stejná, co však není stejné je konfigurační soubor `application.yaml`. Kromě URL kafka brokeru a hodnot pro kafka topic musíme navíc přidat hodnotu pro `consumer.group-id`, viz Obrázek 3.23, kterou budeme potřebovat později.

```java
@Component
public class CancelReservationProducer {
    2 usages
    private static final Logger log = LoggerFactory.getLogger(NewReservationProducer.class);
    2 usages
    private final KafkaTemplate<String, Integer> cancelReservationSender;
    2 usages
    private final String cancelReservationTopic;
    Robin Vavra *
    public CancelReservationProducer(KafkaTemplate<String, Integer> cancelReservationSender,
                                     @Value("${spring.kafka.topic.cancel-reservation}") String cancelReservationTopic) {
        this.cancelReservationSender = cancelReservationSender;
        this.cancelReservationTopic = cancelReservationTopic;
    }
    // send id of reservation to be canceled via kafka
    1 usage   Robin Vavra
    public boolean cancelReservation(Integer id) throws ExecutionException, InterruptedException {
        SendResult<String, Integer> sendResult = cancelReservationSender.send(cancelReservationTopic, id).get();
        log.info("Cancel reservation request was send via kafka.");
        log.info(sendResult.toString());
        return true;
    }
}
```

Obrázek 3.22. Kafka - producer.

```yaml
spring:
  kafka:
    reservation:
      bootstrap-servers: ${KAFKA_RESERVATION_BOOTSTRAP_SERVERS}
      consumer:
        group-id: reservation-bl-id
      topic:
        cancel-reservation: cancel-ticket-reservation
        new-reservation: new-ticket-reservation
```

Obrázek 3.23. Kafka - config.

Stejně jako při vytváření kafka producera musí i pro kafka consumera nejrpve vytvořit třídu s anotací `@Configuration`. Abychom zůstali konzistentní, ukážeme si implementaci kafka consumera opět pro zrušení rezervace, neboli třídy `CancelReservationConsumerConfig`.Funkce `cancelReservationConsumerFactory()` nám vrátí instanci kafka consumera[27]. Tuto instanci následně použijeme ve funkci `cancelReservationKafkaListenerContainerFactory()` pro vytvoření instance `ConcurrentKafkaListenerContainerFactory`[28], viz Obrázek 3.24, kterou později popužijeme při vytváření `kafka listenera` připojeného na příslušný kafka topic a odebírajícího z něj zprávy, viz Obrázek 3.25.

```java
@EnableKafka
@Configuration
public class CancelReservationConsumerConfig {
    1 usage
    @Value("${spring.kafka.reservation.bootstrap-servers}")
    private String bootstrapServers;
    1 usage
    @Value("${spring.kafka.reservation.consumer.group-id}")
    private String groupId;
    1 usage  Robin Vavra
    @Bean
    public ConsumerFactory<String, Integer> cancelReservationConsumerFactory() {
        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, StringSerializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonSerializer.class);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, v: false);
        return new DefaultKafkaConsumerFactory<>(props,new StringDeserializer(),
                new JsonDeserializer<>(Integer.class));
    }
    1 usage  Robin Vavra
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Integer> cancelReservationKafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Integer> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(cancelReservationConsumerFactory());
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        return factory;
    }
}
```

Obrázek 3.24. Kafka - consumer config.

```java
@Service
public class CancelReservationConsumer {
    1 usage
    private static final Logger log = LoggerFactory.getLogger(NewReservationConsumer.class);
    2 usages
    private final ReservationService reservationService;
    Robin Vavra
    public CancelReservationConsumer(ReservationService reservationService) {
        this.reservationService = reservationService;
    }
    // listen to new messages(reservation id) from kafka
    // cancel reservation
    Robin Vavra +1 *
    @KafkaListener(topics = "${spring.kafka.reservation.topic.cancel-reservation}",
                   containerFactory="cancelReservationKafkaListenerContainerFactory")
    public void cancelReservationListener(@Payload Integer reservationId, Acknowledgment ack) throws ExecutionException, InterruptedException {
        log.info("Cancel reservation was received.");
        reservationService.cancelReservation(reservationId);
        ack.acknowledge();
    }
}
```

Obrázek 3.25. Kafka - consumer.

3.4 Testování

V této sekci si popíšeme testování aplikace a také implementaci jednotlivých testů. Původně jsme plánovali otestovat aplikaci pomocí jednotkových testů, UI testů (karate framework) a zátěžových testů (nástroj K6), ale od jednotkových testů jsme se nakonec rozhodli upustit, neboť aplikace neobsahuje žádnou složitější funkci, pro kterou by testování pomocí jednotkových testů bylo vhodné.

3.4.1 UI testy (Karate Framework)

Pro testování UI neboli frontedové části aplikace jsme použili karate framework[15]. Protože je frontend stejný u všech verzí architektur, budou stejné i UI testy. Jediné, co se bude lišit, je URL adresa, která se přepíná pomocí argumentu v příkazové řádce při spouštění testu.

Nejprve přidáme potřebnou dependenci, viz Obrázek 3.26.

Vytvoříme konfigurační soubour s názvem `karate-config.js` a do něj přidáme funkci `fn()`, která se stará o nastavení `baseUrl, driveru a timeoutů`, viz Obrázek 3.27.

Vytvoříme třídu `ReservationsTest`, která slouží ke spuštění testu z příkazového řádku. Při spuštění testu se automaticky spustí test `testParallel`, který následně spustí všechny karate testy (soubory s příponou .feature), nacházející se ve složce `reservations` s tagem `@ReservationComplex`. Testy budou běžet pouze v jednom vlákně, neboť testujeme UI a paralelizmus zde nedává smysl, viz Obrázek 3.28.

Test s tagem `@ReservationComplex` je pouze jeden komplexní test s názvem `create-view-cancel-view-reservation.feature`, viz Obrázek 3.29. Tento test neprve vygeneruje UUID a převede ho na string. Vygenerováné UUID následně použijeme při vytváření rezervace, kde ho nahradíme na místo jména, viz Obrázek 3.30. Po vytvoření rezervace zobrazíme přehled rezervací, viz Obrázek 3.31. Po zobrazení tabulky s vytvořenými rezervacemi najdeme dříve vytvořenou rezervaci s vygenerovaným UUID místo jména a zrušíme ji, viz Obrázek 3.32. Na závěr opět zobrazíme tabulku s vytvořenými rezervacemi, kde naše rezervace je již zrušená. Tento jednoduchý test tedy zaráz otestoval vytvoření a uložení rezervace do databáze, zobrazení přehledu rezervací a zrušení dříve vytvořené rezervace. Spuštění testu například pro monolitní verzi provedeme pomocí příkazu `mvn clean test -Dtest=ReservationsTest -Dkarate.env=monolith`.

```
<!--Karate-->
    <dependency>
      <groupId>com.intuit.karate</groupId>
      <artifactId>karate-junit5</artifactId>
      <version>1.1.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Obrázek 3.26. Karate - dependence.

```
Vávra, Robin *
public class ReservationsTest {
    Vávra, Robin
    @Test
    void testParallel() {
        Results results = Runner.path( ...paths: "classpath:reservations").tags("@ReservationComplex").parallel( threadCount: 1);
        assertEquals( expected: 0, results.getFailCount(), results.getErrorMessages());
    }
}
```

Obrázek 3.28. Karate - ReservationsTest.

```
function fn() {
    var env = karate.env; // get java system property 'karate.env'
    karate.log('karate.env system property was:', env);
    var config = { // base config JSON
        env: env
    };
    switch (env) {
        case 'monolith':
            config.baseUrl = "https://seat-reservation-fe-monolith-aca-robin-vavra.stxcn-aca.stxcn.codenow.com/";
            break;
        case 'synchrounous':
            config.baseUrl = "https://demo-app-synchronous-fe-aca-demo.stxcn-aca.stxcn.codenow.com/";
            break;
        case 'asynchrounous':
            config.baseUrl = "https://demo-app-spring-fe-aca-demo.stxcn-aca.stxcn.codenow.com/";
            break;
        default:
            config.baseUrl = env;
            break;
    }
    // karate.configure('driver', {type: 'chrome'});
    karate.configure('driver', { type: 'geckodriver', executable: '/home/vavrarob/Downloads/geckodriver' });
    //timeouts settings
    karate.configure('retry',{count:5, interval:3000});
    karate.configure('connectTimeout', 25000);
    karate.configure('readTimeout', 25000);
    return config;
}
```

Obrázek 3.27. Karate - config.

```
@ReservationComplex
Feature:
  Scenario:
    * driver baseUrl

    * def uuid = function(){ return java.util.UUID.randomUUID() + '' }
    * def myId = uuid().toString()
    * print(myId)
    * call read('../common/create-reservation.feature') {firstName : '#(myId)'}
    * delay(1000)
    * call read('../common/get-view-of-reservations.feature')
    * delay(1000)
    * call read('../common/cancel-reservation.feature') {firstName : '#(myId)'}
    * delay(1000)
    * call read('../common/get-view-of-reservations.feature')
    * delay(1000)
```

Obrázek 3.29. Karate - complex test.

3.4.2 Zátěžové testy (nástroj K6)

V této části si ukážeme jednoduchý script, který posílá požadavky pro vytváření rezervací pomocí nástroje K6[16], viz Obrázek 3.33. Při spouštění scriptu nadefinujeme název prostředí a clusteru, na kterých je aplikace nasazená a dále název komponenty, která zpracovává požadavky z frontendu. U monolitní verze aplikace to je backend komponenta, u ostatních verzí API komponenta. Z těchto názvů se skládá výsledná URL adresa, kam se budou posílat `POST` požadavky společně s připravenými daty. Tyto data

```
@Reservation
Feature: Create reservation
  Scenario:

    Then waitForText('h1', 'Reservation')
    * retry().click('a[name=trainId]')
    * retry().click('{div/div/a}ICE 575 Munich')
    * retry().click('a[name=seatId]')
    * retry().click('{div/div/a}Coach 1 / Seat 10')
    * retry().input('input[name=firstName]', firstName)
    * retry().input('input[name=lastName]', 'lastName')
    * retry().input('input[name=email]', 'karate@test.com')
    * delay(1000)
    * retry().click('button[type=submit]')
    * delay(4000)
    * match driver.dialog == 'Reservation created successfully'
    * retry().dialog(true)
```

Obrázek 3.30. Karate - create reservation.

```
@ReservationView
Feature: Get view of reservations
  Scenario:

    Then waitForText('h1', 'Reservation')
    * retry().click('{}Get Reservations')
    * waitFor('{}STATUS')
```

Obrázek 3.31. Karate - reservation view.

```
@ReservationCancel
Feature: Cancel reservation
  Scenario:

    Then waitForText('h1', 'Reservation')
    * retry().click('//tbody/tr/td[normalize-space(text()) = \'' + firstName + '\']/../td[7]/button[normalize-space(text()) = \'Cancel Reservation\']')
    * delay(2000)
    * retry().dialog(true)
    * delay(1000)
```

Obrázek 3.32. Karate - cancel reservation.

však nejsou úplná, záměrně jsme nechali `email` prázdný, neboť nechceme, aby emailová adresa sloužící pro rozesílání emailů nebyla zablokovaná z důvodu spamu.

Zátěžový test se spouští pomocí příkazu `k6 run -d 30s -u 20 -e COMPONENT=<your_component> -e ENVIRONMENT=<your_environment> -e CLUSTER=<your_cluster> ./script.js`.
Parametr -d určuje délku běhu testu, parametr -u určuje počet fiktivních uživatelů (počet požadavků za jednu sekundu).

Všechny tři verze aplikace jsme otestovali zátěžovým testem 3.33 s parametry `-d 30s -u 20`. Z výsledků můžeme vidět výhodu asynchronní verze, oproti ostatním verzím. Asynchronní verze dokázala zpracovat 549 požadavků, viz Obrázek 3.36, zatímco synchronní verze 70, viz Obrázek 3.35, a monolitní verze pouze 69, viz Obrázek 3.34.

```
import http from 'k6/http';
import { check, sleep,execution } from 'k6';
import { Rate } from 'k6/metrics';
export let errorRate = new Rate('errors');


const environment = __ENV.ENVIRONMENT;
const component = __ENV.COMPONENT;
const cluster = __ENV.CLUSTER;
const url = `https://${component}-${environment}.${cluster}.codenow.com/reservation`;

export default function () {
    var data = JSON.stringify( value: {
        date: new Date(),
        email: '',
        firstName: 'On',
        lastName: 'Test',
        seatId: '1',
        trainId: '1',
    });
    var params = {
        headers: {
            'Content-Type': 'application/json',
        },
    };
    check(http.post(url, data, params), {
        'status is 201': (r) => r.status == 201,
    }) || errorRate.add(1);
    sleep(1);
}
```

Obrázek 3.33. K6 - code.

```
  scenarios: (100.00%) 1 scenario, 20 max VUs, 1m0s max duration (incl. graceful stop):
           * default: 20 looping VUs for 30s (gracefulStop: 30s)


running (0m40.4s), 00/20 VUs, 69 complete and 0 interrupted iterations
default ✓ [======================================] 20 VUs  30s

     ✓ status is 201

     checks.........................: 100.00% ✓ 69       ✗ 0
     data_received..................: 90 kB   2.2 kB/s
     data_sent......................: 25 kB   626 B/s
     http_req_blocked...............: avg=24.25ms  min=693ns   med=1.14µs   max=84.7ms   p(90)=84.1ms   p(95)=84.43ms
     http_req_connecting............: avg=5.69ms   min=0s      med=0s       max=26.6ms   p(90)=20.49ms  p(95)=23.91ms
     http_req_duration..............: avg=9.18s    min=2.55s   med=9.51s    max=15.61s   p(90)=11.46s   p(95)=13.12s
       { expected_response:true }...: avg=9.18s    min=2.55s   med=9.51s    max=15.61s   p(90)=11.46s   p(95)=13.12s
     http_req_failed................: 0.00%   ✓ 0        ✗ 69
     http_req_receiving.............: avg=128.77µs min=59.72µs med=126.32µs max=189.28µs p(90)=161.91µs p(95)=172.77µs
     http_req_sending...............: avg=183.28µs min=98.1µs  med=181.53µs max=303.63µs p(90)=223.71µs p(95)=227.57µs
     http_req_tls_handshaking.......: avg=13.19ms  min=0s      med=0s       max=52.03ms  p(90)=47.3ms   p(95)=49.51ms
     http_req_waiting...............: avg=9.18s    min=2.55s   med=9.51s    max=15.61s   p(90)=11.46s   p(95)=13.12s
     http_reqs......................: 69      1.708221/s
     iteration_duration.............: avg=10.2s    min=3.55s   med=10.51s   max=16.61s   p(90)=12.55s   p(95)=14.2s
     iterations.....................: 69      1.708221/s
     vus............................: 1       min=1      max=20
     vus_max........................: 20      min=20     max=20
```

Obrázek 3.34. K6 - monolith.

```
running (0m47.2s), 00/20 VUs, 70 complete and 0 interrupted iterations
default ✓ [======================================] 20 VUs  30s

     ✓ status is 201

     checks.........................: 100.00% ✓ 70       ✗ 0
     data_received..................: 81 kB   1.7 kB/s
     data_sent......................: 24 kB   518 B/s
     http_req_blocked...............: avg=17.87ms  min=302ns   med=1.14µs   max=63.34ms  p(90)=62.82ms  p(95)=63.02ms
     http_req_connecting............: avg=4.86ms   min=0s      med=0s       max=20.32ms  p(90)=17.42ms  p(95)=17.46ms
     http_req_duration..............: avg=9.96s    min=1.71s   med=10.18s   max=29.06s   p(90)=14.87s   p(95)=17.19s
       { expected_response:true }...: avg=9.96s    min=1.71s   med=10.18s   max=29.06s   p(90)=14.87s   p(95)=17.19s
     http_req_failed................: 0.00%   ✓ 0        ✗ 70
     http_req_receiving.............: avg=112.83µs min=22.61µs med=117.03µs max=183.88µs p(90)=138.89µs p(95)=145.1µs
     http_req_sending...............: avg=139.18µs min=36.19µs med=172.11µs max=215.03µs p(90)=194.91µs p(95)=207.76µs
     http_req_tls_handshaking.......: avg=12.15ms  min=0s      med=0s       max=45.81ms  p(90)=42.79ms  p(95)=43.21ms
     http_req_waiting...............: avg=9.96s    min=1.71s   med=10.18s   max=29.06s   p(90)=14.87s   p(95)=17.19s
     http_reqs......................: 70      1.484323/s
     iteration_duration.............: avg=10.98s   min=2.77s   med=11.18s   max=30.06s   p(90)=15.88s   p(95)=18.19s
     iterations.....................: 70      1.484323/s
     vus............................: 1       min=1      max=20
     vus_max........................: 20      min=20     max=20
```

Obrázek 3.35. K6 - synchronous.

```
running (0m30.8s), 00/20 VUs, 549 complete and 0 interrupted iterations
default ✓ [======================================] 20 VUs  30s

     ✓ status is 201

     checks.........................: 100.00% ✓ 549       ✗ 0
     data_received..................: 173 kB  5.6 kB/s
     data_sent......................: 109 kB  3.5 kB/s
     http_req_blocked...............: avg=2.59ms   min=218ns   med=1.08µs   max=72.64ms  p(90)=1.53µs   p(95)=1.76µs
     http_req_connecting............: avg=669.34µs min=0s      med=0s       max=20.28ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=101.93ms min=21.67ms med=43.39ms  max=522.36ms p(90)=293.3ms  p(95)=328.99ms
       { expected_response:true }...: avg=101.93ms min=21.67ms med=43.39ms  max=522.36ms p(90)=293.3ms  p(95)=328.99ms
     http_req_failed................: 0.00%   ✓ 0         ✗ 549
     http_req_receiving.............: avg=110.41µs min=25.46µs med=113.17µs max=291.86µs p(90)=138.17µs p(95)=147.88µs
     http_req_sending...............: avg=167.24µs min=35.18µs med=172.28µs max=548.36µs p(90)=211.93µs p(95)=225.57µs
     http_req_tls_handshaking.......: avg=1.83ms   min=0s      med=0s       max=52ms     p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=101.65ms min=21.4ms  med=43.15ms  max=522.05ms p(90)=293ms    p(95)=328.67ms
     http_reqs......................: 549     17.823533/s
     iteration_duration.............: avg=1.1s     min=1.02s   med=1.04s    max=1.59s    p(90)=1.29s    p(95)=1.38s
     iterations.....................: 549     17.823533/s
     vus............................: 20      min=20      max=20
     vus_max........................: 20      min=20      max=20
```

Obrázek 3.36. K6 - asynchronous.

Kapitola 4
Závěr

4.1 Shrnutí

Cílem práce bylo nastudovat moderní cloud-native technologie a praktiky a iterativním způsobem tyto získané znalosti využít při rozšiřování demo rezervační aplikace na platformě CodeNOW.

Studium zmiňovaných technologií a praktik bylo nezbytnou součástí práce. Nabité vědomosti přinesly potřebný nadhled na námi řešený problém, bez kterého by vyl vývoj aplikace velmi obtížný.

Výsledkem práce je demo rezervační aplikace implementovaná podle tří architektur. Částečný monolit rozdělený na frontend a backend komponenty. Synchronní mikroslužby rozšířené oproti částečnému monolitu o API komponentu a MailSender. A poslední verzí jsou asynchronní mikroslužby, kde komponenty jsou stejné jako u synchronních mikroslužeb, ale ke komunikace se používá převážně platforma APACHE Kafka.

Funkcionalita aplikace byla kromě ukládání rezervací navíc rozšířena o přehled uložených rezervací a zrušení rezervace. Dále bylo implementováno odesílání emailu při úspěšném vytvoření nebo zrušení rezervace.

Pro sledování komunikace mezi komponentami při zpracovávání nějakého požadavku byl navíc přidaný tracing, který umožní vizualizaci toku určitého požadavku napříč komponentami. Kromě samotné vizualizace komunikace mezi komponentami se navíc přidala možnost vytvářet spany ručně a tím zobrazit libovolnou část kódu. Poslední částí týkající se tracingu bylo přidání aspectu umožňujícího zobrazovat dotazy do databáze.

Při testování aplikace jsme dospěli k názoru, že aplikace pro jednotkové testy nevhodná, neboť jsou jednotkové testy vhodné pro testování složitějších funkcí, jako je například výpočet nějaké hodnoty, a demo rezervační aplikace žádné takovéto funkce neobsahuje. Aplikace však obsahuje komplexní Karate test, který vytvoří rezervaci, kde na místo jména je vygenerováno UUID, zobrazí uložené rezervace, zruší vytvořenou rezervaci s dříve vygenerovaným UUID místo jména a opět zobrazí rezervace. Zde je nutno zmínit, že psaní těchto Karate testů bylo poměrně náročné, protože jsem se setkal s chybou, jejíž řešení se mi nikde nepovedlo najít. Po dlouhém hledání a zkoušení jsme dospěli k názoru, že je problém v operačním systému (Ubuntu 20.04.4 LTS) a ne v kódu, neboť stejný test na Mac OS fungoval. Další problém byl websocket exception, kdy v průběhu testu bylo přerušeno spojení s webovým prohlížečem. Tento problém však vyřešil přechod z Chrome driveru na Geckodriver. Pro zátěžové testy jsme se rozhodli implementovat zátěžové testy pro vytváření rezervací pomocí nástroje K6. Zde bylo možné vidět hlavní rozdíl mezi asynchronní verzí a ostatními verzemi. Díky asynchronní komunikaci se totiž nečeká na odpověď a může se tak naráz zpracovat větší množství požadavků, protože se požadavky posílají přes kafku a backend komponenta si je pak postupně sama odebírá.

4.2 Budoucí práce

Dalším krokem je vymyslet způsob zobrazování rezervací, aby asynchronní verze aplikace byla opravdu celá asynchronní. Způsobů jak tento problém vyřešit je celá řada a proto je nezbytná konzultace se SW architektem.

Dále by bylo vhodné rozšířit aplikaci o přihlašování a rozdělit role uživatelů alespoň na admin a user, aby si user mohl zobrazovat a rušit pouze ním vytvořené rezervace. Společně s přihlašováním se také musí vyřešit bezpečnost, aby nedošlo k úniku tajných informací, jako je například heslo.

Možným rozšířením je také nahradit MailSender transakční mailingovou službou nebo integrovat platební systém pro platbu jízdenek.

Příloha A
Tracing

Tracing in CodeNOW - Java Spring Boot

Created a month ago, last updated 14 days ago

Tracing is a simple way how to visualize the flow of a request between components. Unique trace ID is generated for every new request. As some component receives the request a new span is assigned for that component and the span is added to the trace. The traces are then send to jaeger collector. You can find more information about the tracing here.

How to implement tracing into your app

Dependencies

First we need to add these dependencies into our pom.xml file.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
    <version>3.0.2</version>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
    <version>3.0.2</version>
</dependency>
```

Configuration

We also need to change the configuration file (application.yaml) for our application.

Set name of the application, for example **demo-app-spring-bl.**

```
spring:
    application:
        name: demo-app-spring-bl
```

Add zipkin and sleuth properties.

CodeNOW jaeger collector (deploy).

```
spring:
    zipkin:
        enabled: true
```

```
        baseUrl:
http://tracing-jaeger-collector.tracing-system:9411
    sleuth:
        propagation:
            type: B3
            tag:
                enabled: true
```

Localhost jaeger collector (running locally).
To run jaeger in your local environment check this.

```
spring:
    zipkin:
        enabled: true
        baseUrl: http://localhost:9411
    sleuth:
        propagation:
            type: B3
            tag:
                enabled: true
```

Tracing with APACHE Kafka.

```
spring:
    sleuth:
        propagation:
            type: B3
            tag:
                enabled: true
        messaging:
            kafka:
                enabled: true
```

Tracing and database queries

You can visualize any call to the DB by adding aspect.
This aspect automatically creates a new span for every query to the DB.

First we need to add a TracingHelper class. This class can create new spans.

```
package org.example.service.tracing;

import org.springframework.cloud.sleuth.Span;
```

```
import org.springframework.cloud.sleuth.Span.Kind;
import org.springframework.cloud.sleuth.Tracer;
import org.springframework.stereotype.Component;

@Component
public class TracingHelper {

    private static final String CLASS = "class";

    private static final String TYPE = "type";

    private final Tracer tracer;

    /**
     * Constructor.
     *
     * @param tracer sleuth tracer
     */
    public TracingHelper(Tracer tracer) {
        this.tracer = tracer;

    }

    /**
     * Create sleuth client span.
     *
     * @param remoteServiceName executed service name
     * @param type tag 'type'
     * @param className tag 'class'
     * @return new tracing span
     */
    public Span createClientSpan(String remoteServiceName,
String type, String className) {
        return tracer.nextSpan().tag(TYPE, type)
                                .tag(CLASS, className)
                                .name(remoteServiceName);
    }


    /**
     * Start sleuth client span.
     *
     * @param span span
     * @return new Tracer.spanInScope
```

```
     */
    public Tracer.SpanInScope spanInScope(Span span) {
        return tracer.withSpan(span.start());
    }
}
```

Now we need to add an aspect for our repository. Adding this aspect will cause a new span to be automatically created with every db query and it will take care of the whole lifecycle of the created span.

```
package org.example.service.tracing;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.sleuth.Span;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class RepositoryTracingAspect {
    private static final String TRACING_TYPE = "repository";
    private TracingHelper tracingHelper;

    @Autowired
    public RepositoryTracingAspect(final TracingHelper
tracingHelper) {
        this.tracingHelper = tracingHelper;
    }


@Around("within(org.springframework.data.repository.CrudReposi
tory+)")
    public Object traceRepositoryCalls(ProceedingJoinPoint
joinPoint) throws Throwable {
        String className =
joinPoint.getSignature().getDeclaringTypeName();
        String targetMethod =
joinPoint.getSignature().getName();

        Object proceed;
```

```
        Span span =
tracingHelper.createClientSpan(targetMethod, TRACING_TYPE,
className);
        try (Tracer.SpanInScope ws =
tracingHelper.spanInScope(span)) {
           proceed = joinPoint.proceed();
        } finally {
            span.end();
        }

        return proceed;
    }
}
```

Manually created spans

We can even create new spans manually by using the createClientSpan function from TracingHelper class. If we want to create spans manually, we have to take care of the lifecycle of the created span on our own. Once we have the span created, we need to start it with the spanInScope function from TracingHelper class. Every span needs to be ended and because our code could produce some exceptions we need to warp it into a try(){} finally{} block as is shown below.

Code can look like this:

```
Span span = tracingHelper.createClientSpan("Legacy App 1",
                                           "service",

this.getClass().getName())
try(Tracer.SpanInScope ws = tracingHelper.spanInScope(span)){
    // Enter your code here
} finally {
    span.end();
}
```

Literatura

[1] *The Coders' DevOps Value Stream Delivery Platform.*
https://www.codenow.com/.

[2] *CodeNOW documentation.*
https://docs.codenow.com/docs/.

[3] *What is an application architecture?*
https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture.

[4] *Grafana Loki.*
https://grafana.com/oss/loki/.

[5] *Hypertext Transfer Protocol.*
https://cw.fel.cvut.cz/b211/_media/courses/b6b36ear/lectures/lecture-06-rest-s.pdf.

[6] *TCP/IP.*
https://www.ibm.com/docs/en/aix/7.2?topic=protocol-tcpip-protocols.

[7] *UDP.*
https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/.

[8] *What is a REST API?*
https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[9] *12-factor app.*
https://12factor.net/.

[10] *The power of cloud technology.*
https://www.cedar-bay.com/the-power-of-cloud-technology-and-how-it-helps-us-navigate-the-new-normal/.

[11] *What's the difference between agile, CI/CD, and DevOps?*
https://www.synopsys.com/blogs/software-security/agile-cicd-devops-difference.

[12] *Distributed Tracing with Spring Cloud Sleuth and Spring Cloud Zipkin.*
https://spring.io/blog/2016/02/15/distributed-tracing-with-spring-cloud-sleuth-and-spring-cloud-zipkin.

[13] *Jaeger: open source, end-to-end distributed tracing.*
https://www.jaegertracing.io/.

[14] *Jendotkové testování: JUnit, TestNG a základy efektivního návrhu.*
https://moodle.fel.cvut.cz/pluginfile.php/289703/mod_resource/content/1/TS1_prednaska_6_7_8.pdf.

[15] *Test Automation Made Simple.*
https://github.com/karatelabs/karate.

[16] *The best developer experience for load testing.*
https://k6.io/.

[17] *Demo application reservation.*
https://app.tettra.co/teams/STXCN/pages/ticket-reservation-demo-app-main-page.

[18] *Apache Kafka.*
https://kafka.apache.org/.

[19] *CQRS.*
https://martinfowler.com/bliki/CQRS.html.

[20] *Data Transfer Object.*
https://www.baeldung.com/java-dto-pattern.

[21] *How to Send Email Using Spring Boot.*
https://www.technicalkeeda.com/spring-boot-tutorials/how-to-send-email-using-spring-boot.

[22] *Plain old Java object.*
https://en.wikipedia.org/wiki/Plain_old_Java_object.

[23]

[24] *Interface ProducerFactory¡K,■V¿.*
https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/ProducerFactory.html .

[25] *Class KafkaTemplate¡K,■V¿.*
https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/KafkaTemplate.html .

[26] *Interface KafkaOperations¡K,■V¿.*
https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/KafkaOperations.html .

[27] *Interface ConsumerFactory¡K,■V¿.*
https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/ConsumerFactory.html .

[28] *Class ConcurrentKafkaListenerContainerFactory¡K,■V¿.*
https://docs.spring.io/spring-kafka/api/org/springframework/kafka/config/ConcurrentKafkaListenerContainerFactory.html .