VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
EKONOMICKÁ FAKULTA

KATEDRA APLIKOVANÉ INFORMATIKY

Optimalizace e-mailového messagingu cloudové služby

Email Messaging Optimization of a Cloud Service

Student: Patrik Polaček
Vedoucí bakalářské práce: Ing. Vítězslav Novák, Ph.D.

Ostrava 2019

VŠB - Technická univerzita Ostrava
Ekonomická fakulta
Katedra aplikované informatiky

Zadání bakalářské práce

Student: **Patrik Polaček**

Studijní program: B6209 Systémové inženýrství a informatika

Studijní obor: 6209R017 Informatika v ekonomice

Téma: Optimalizace e-mailového messagingu cloudové služby
Email Messaging Optimization of a Cloud Service

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Úvod
2. Teoretický popis technologií použitých při optimalizaci
3. Analýza požadavků a současného stavu e-mailového messagingu cloudové služby
4. Implementace řešení optimalizace
5. Závěr

Seznam použité literatury
Seznam zkratek
Prohlášení o využití výsledků bakalářské práce
Seznam příloh
Přílohy

Seznam doporučené odborné literatury:

WALLS, Craig. *Spring in Action, Fifth Edition*. Shelter, Island, NY: Manning Publications Co., 2018. ISBN 9781617294945.
CARNELL, John. *Spring Microservices in Action*. Shelter, Island, NY: Manning Publications Co., 2017. ISBN 9781617293986.
BLOCH, Joshua. *Effective Java*. Third edition. Boston: Addison-Wesley, 2018. ISBN 0134685997.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Vítězslav Novák, Ph.D.**

Datum zadání: 23.11.2018
Datum odevzdání: 10.05.2019

Ing. Petr Rozehnal, Ph.D.
vedoucí katedry

prof. Dr. Ing. Zdeněk Zmeškal
děkan fakulty

Prehlasujem, že som celú prácu, vrátane všetkých príloh, vypracoval samostatne.

V Ostrave dňa 6.mája 2019

....

Patrik Polaček

Rád by som sa touto cestou poďakoval pánovi Ing.Vítězslavu Novákovi Ph.D za pomoc, odborné vedenie a cenné rady pri vypracovaní bakalárskej práce

Obsah

1 Úvod **5**

2 Teoretický popis technológií použitých pri optimalizácii **7**

2.1 Technologická architektúra projektu Tieto DevOps Space 7

2.1.1 Mikroslužba 7

2.1.2 Softvérová architektúra orientovaná na mikroslužby 8

2.2 Programovacie jazyky, nástroje a frameworky použité pri optimalizácii 10

2.2.1 Spring Framework 10

2.2.2 Spring Boot 13

2.2.3 Java 14

2.2.4 Maven 16

2.2.4.1 Štruktúra pom.xml 19

2.2.5 Lombok 20

2.3 Technológie určené pre prácu s databázou 20

2.3.1 PostgreSQL 21

2.3.2 Objektovo-relačné mapovanie 22

2.3.2.1 Java Persistence API 22

2.3.2.2 Hibernate 23

2.3.3 Tvorba prístupového API k záznamom databáze 24

2.3.3.1 JPA Repository 24

2.4 RabbitMQ 25

3 Analýza požiadavkou a súčasného stavu e-mailového messagingu cloudovej služby 28

3.1 Zázemie poskytovanej služby 28

3.2 Súčasný stav e-mailového messagingu 28

3.3 Požiadavky optimalizácie 29

4 Implementácia riešení optimalizácie **31**

4.1.1 Entity Relationship Diagram databáze mikroslužby 31

4.2 Triedy a rozhrania mikroslužby 32

4.2.1 Controller 32

4.2.2 Objekty pre prenos dát 34

4.2.2.1 MailMessageDto 34

4.2.2.2 RabbitMailMessage 36

4.2.2.3 RabbitJsonFile 37

4.2.3 Objekty entít pre ukládanie do relačnej databáze 38
4.2.3.1 EntityMailMessage 38
4.2.3.2 EntityFile 41
4.2.4 JPA repozitár 42
4.2.5 Implementácia RabbitMQ 44
4.2.5.1 MailMessageRabbitListener 45
4.2.6 MainService 46
4.2.6.1 Spracovanie správy prijatej pomocou Rest Controlleru 48
4.2.6.2 Konfigurácia RabbitMQ 49
4.2.6.3 Spracovanie správy prijatej pomocou nástroja RabbitMQ 50
5 Záver 52

1 Úvod

Hlavným cieľom bakalárskej práce je zabezpečiť optimalizáciu e-mailového messagingu cloudovej služby.

Základným prvkom optimalizácie bude mikroslužba, ktorá zabezpečí požadovanú funkcionalitu vyplývajúcu z požiadaviek na optimalizáciu. Východzím stanoviskom pre tvorbu bakalárskej práce bola požiadavka na všeobecnú optimalizáciu procesu rozosielania e-mailov. Jedná sa o automatizovaný proces inicializovaný zo strany ostatných mikroslužieb v rámci celkového projektu aplikácie.

Jadrom projektu, na ktorom bola bakalárska práca realizovaná je cloudová služba, poskytovaná spoločnosťou Tieto. Základnou hodnotou tejto služby je poskytovanie cloudovej platformy ako služby (PAAS – Platform as a Service). Presnejšie je v nej obsiahnutá možnosť zákazníkov využiť výpočtové prostredie a aplikácie potrebné na tvorbu aplikačného softvéru metodikou vývoja DevOps. Názov poskytovanej služby je Tieto DevOps Space.

Aplikačná architektúra celého projektu je postavená na štruktúre mikroslužieb. Jednotlivé mikroslužby zabezpečujú rôznorodú funkcionalitu v rámci celého projektu. Niektoré mikroslužby umožňujú zákazníkom prideľovanie aplikácií a výpočtových prostriedkov pre tvorbu vlastných aplikácií vo webovom grafickom rozhraní platformy ako služby, iné napríklad zabezpečujú monitorovanie využitia daných prostriedkov. Po pridelení prostriedkov je zákazníkom odoslaný e-mail, ktorý ich informuje o využití jednotlivých služieb. Súčasná podoba daného messagingu je riešená na úrovni jednotlivých mikroslužieb, u ktorých je táto funkcionalita vyžadovaná. Tento proces zahŕňa vytvorenie e-mailu s požadovaným obsahom a s prípadnými prílohami. Nasleduje odoslanie e-mailovej správy v kompatibilnej forme na SMTP server.

Optimalizáciu by sme mohli rozdeliť do dvoch základných častí. Prvou časťou optimalizácie bude vytvorenie samostatnej novej mikroslužby, ktorá bude zabezpečovať poskytovanie požadovaných funkcionalít potrebných pre optimalizáciu e-mailového messagingu. Medzi tieto funkcionality bude patriť schopnosť vystavovať rozhranie pre

programovanie aplikácie (API – Application Programming Interface) identifikované jednotlivými URI (Unified Resource Identifier), ktoré budú schopné ostatné mikroslužby použiť v momente záujmu o vytvorenie e-mailu pri novom pridelených výpočtových prostriedkoch alebo aplikácií zo strany zákazníka, prípadne iných prípadov použitia ktorých záujmom je odoslanie e-mailovej správy. Táto forma komunikácie je založená na protokole http. Výhodou bude centralizovaná správa tejto funkcionality, odstránenie zastaralého a redundantného kódu z ostatných mikroslužieb, lepšia možnosť kontroly vyplývajúca z použitia relačnej databázy, ktorá bude ukladať záznamy o odoslaných e-mailoch.

Ďalšou z funkcionalít poskytovanou mikroslužbou je schopnosť preposielania e-mailových správ zamestnancom pracujúcim na tvorbe Tieto DevOps Space. Podpora tejto funkcionality vyplýva zo záujmu o príjem e-mailu zamestnancami alebo ich skupinou, v prípade, že je napríklad nasadený určitý špecifický server potrebný na chod Tieto DevOps Space alebo zamestnanci chcú získať určitú informatívnu správu zo strany ostatných mikroslužieb. Obsah e-mailov je v tomto prípade informačný a môže obsahovať prílohy.

Vyššie uvedené požiadavky môžu byť alternatívne realizované prostredníctvom asynchrónneho message brokeru RabbitMQ. Vo všeobecnosti môžeme túto funkcionalitu definovať ako alternatívu k prvému menovanému spôsobu posielania správ pomocou protokolu HTTP. RabbitMQ umožňuje príjem správ zo strany viacerých príjemcov jednoduchým a škálovateľným spôsobom. Uvedený softvér je už v rámci projektu využívaný, a preto je jeho použitie benefitujúce z ekonomického a funkčného hľadiska, ktoré nám umožní naplniť požiadavky pracovníkov bez nutnosti kúpy dodatočných softvérových licencií.

2 Teoretický popis technológií použitých pri optimalizácii

2.1 Technologická architektúra projektu Tieto DevOps Space

2.1.1 Mikroslužba

Ako definuje Carnell (2017, s. 2) „*Mikroslužba je malá, voľne viazaná a distribuovaná služba. Umožňuje dekomponovať rozsiahlu aplikačnú štruktúru na ľahko riaditeľné komponenty s úzko definovaným rozsahom zodpovedností.*“

Za základné charakteristiky mikroslužby sa považuje:

- definovaná funkcionalita, jej rozsah je vždy presne vymedzení, aby naplňovala iba svoj účel a nezasahovala explicitným spôsobom do iných častí aplikácie. Táto vlastnosť zároveň zabezpečuje jednoduchší spôsob navigácie v celom aplikačnom projekte vďaka čomu sa znižuje jeho celková komplexnosť,

- voľná viazanosť, ktorá zabezpečuje komunikáciu medzi jednotlivými mikroslužbami v rámci celkovej architektúry napríklad pomocou metód tvorby programových rozhraní, medzi ktoré patrí metóda REST(Representational state transfer) a sieťového protokolu HTTP (Hypertext Transfer Protocol) s častím používaním dátového formátu JSON (JavaScript Object Notation) pre prenos dát,

- individuálna dátová architektúra, ktorá zabezpečuje špecifický dátový model pre každú z mikroslužieb, ktorá vlastní dáta a samostatne pristupuje k databáze v rámci svojej vymedzenej funkcionality,

- samostatnosť, pomocou ktorej môže byť každá z mikroslužieb samostatne kompilovaná a nasadená nezávisle na ostatných službách aplikačnej architektúry. Táto vlastnosť zjednodušuje schopnosť testovania, bezpečnosti a stability celkovej aplikácie a má rozsiahle možnosti využitia pri nasadený v cloudovom prostredí.

Pri tvorbe mikroslužby je nutné zvoliť správny rozsah funkcionalít, ktoré bude daná mikroslužba poskytovať v rámci celkovej architektúry systému. V prípade extrémnej špecifikácie môže dochádzať k tvorbe štruktúry určenej iba na zápis dát do databáze, vzniká teda úroveň abstraktného rozhrania reprezentujúceho jedine zápis entít do databáze. Takáto mikroslužba nemá žiaden zmysluplný cieľ v rámci celkovej architektúry orientovanej na mikroslužby.

Prípad príliš rozsiahleho záberu funkcionalít poskytovaných mikroslužieb rovnako nie je vhodný z dôvodov zvyšujúcej sa komplexnosti systému, porušovania zásady voľnej viazanosti, nedodržania konzistencie dát spravovaných danou mikroslužbou a ich logickej nesúvislosti, prípadne sťažením testovaním mikroslužby.

2.1.2 **Softvérová architektúra orientovaná na mikroslužby**

Tento architektonický prístup využíva súbor mikroslužieb na komplexnú tvorbu aplikácie. Mikroslužba vystupuje ako základný prvok vývoja týmto prístupom, pre popis základnej charakteristiky mikroslužby viz kapitola 2.1.1.

Medzi základné výhody prístupu k tvorbe aplikácií pomocou tejto architektúry ako uvádza *Walls(2018) patrí:*

- *technologická nezávislosť mikroslužieb od použitia špecifického programovacieho jazyka, knižníc, frameworku alebo platformy pre tvorbu celej aplikácie. Tento benefit vyplýva zo spôsobu voľnej komunikácie medzi mikroslužbami pomocou protokolu HTTP a metódy tvorby programových rozhraní,*

- *zvýšená stabilita aplikácie, ktorej fungovanie môže byť v obmedzenej podobe zachované pri zlyhaní jednej z mikroslužieb, čím môžeme znížiť celkové riziko výpadku aplikácie,*

- *flexibilná škálovateľnosť výpočtových prostriedkov, ktorá umožňuje priraďovať špecifické výpočtové prostriedky pri použití cloudových technológií každej z mikroslužieb tvoriacej celkovú architektúru samostatne podľa individuálnej potreby, vďaka čomu je možné výrazne znížiť ekonomické náklady na fungovanie aplikácie pri využití cloudových služieb,*

- *časová náročnosť implementácie mikroslužby do produkčného fungovania je značne znížená, pretože sa jedná o malú časť celkovej aplikácie vďaka čomu nemusíme nasadzovať celú aplikáciu, ale iba jej malú časť,*

- *zjednodušené testovanie vďaka značne zníženému rozsahu mikroslužby v porovnaní s testovaním jednotnej monolitickej aplikácie a času potrebného pre jej nasadenie,*

- *zvýšená celková štruktúrovanosť a riadenie vývoja systému vďaka pevne definovanému zámeru jednotlivých mikroslužieb.*

Táto architektúra vývoja so sebou prináša aj riziká, hlavne v podobe zabezpečenia sieťovej stability pre zachovanie efektívnej komunikácie jednotlivých mikroslužieb. Ďalším faktorom je potreba implementácie nových techník umožňujúcich implementáciu súboru mikroslužieb v cloudovom prostredí pri zachovaní distribuovanosti celkovej aplikácie.

Cieľom bakalárskej práce je vytvoriť samostatnú mikroslužbu s definovaným účelom, ktorá bude zapadať do celkovej architektúry vývoja Tieto DevOps Space orientovanej na mikroslužby. Špecifický technologický postup nasadzovania a fungovania architektúry mikroslužieb v cloudovom prostredí nebude diskutovaný z dôvodu rozsahu a zamerania problematiky bakalárskej práce na optimalizáciu e-mailového messagingu, ktorá je spracovaná v rámci jednej samostatnej mikroslužby a prípadných ďalších doplnkových činnostiach zabezpečujúcich dosiahnutie optimalizovaného stavu.

2.2 Programovacie jazyky, nástroje a frameworky použité pri optimalizácii

2.2.1 Spring Framework

Jedná sa o framework založený na programovacom jazyku Java, ktorý slúži na tvorbu aplikácií. Prvá verzia Spring Frameworku bola vytvorená v roku 2003. V súčasnosti vývoj spadá pod spoločnosť Pivotal Software, ktorej majoritným vlastníkom je spoločnosť Dell Technologies. Prvotným autorom Spring Frameworku bol Rod Johnson.

Ako uvádza Walls(2018, s. 4) „*Vo svojom jadre, Spring ponúka kontajner, nazývaný Spring Application Context, ktorý vytvára a spravuje životný cyklus aplikačných komponentov. Tieto komponenty, nazývané aj beans, sú vzájomne prepojené práve cez Spring Application Context, aby spoločne tvorili kompletnú aplikáciu".*

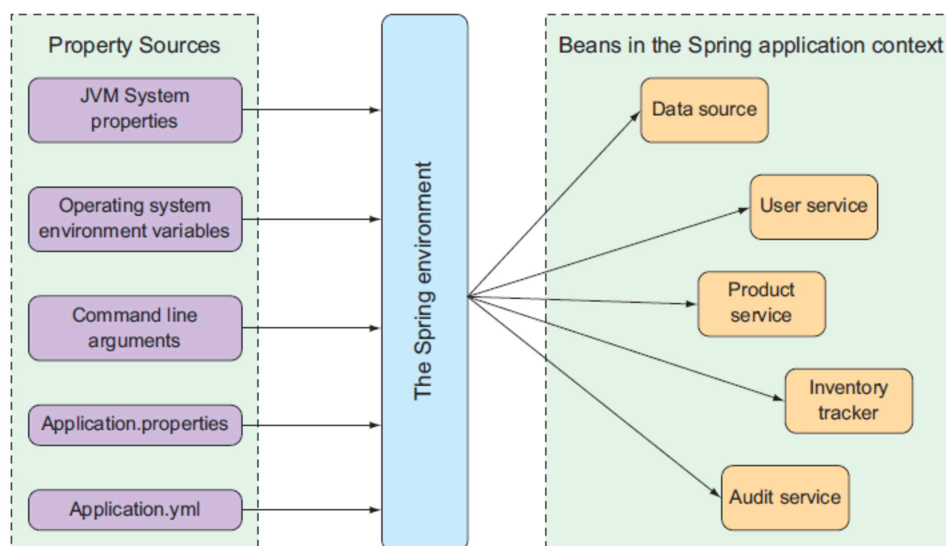
Proces vzájomného prepájania jednotlivých aplikačných komponentov sa nazýva aj vkladanie závislostí (DI – Dependency Injection). Táto technika spolieha na samostatnú entitu kontajnera, aby vytváral a spravoval životný cyklus všetkých aplikačných komponentov a vzájomne im vkladal závislosti podľa potreby. Tento proces zvyčajne prebieha pomocou vkladania závislostí pomocou konštruktoru triedy, setteru objektu alebo implementáciou špecifického rozhrania.

Konfigurácia jednotlivých komponentov prebieha pomocou anotácií. Tieto anotácie umožnujú kontajneru identifikovať jednotlivé komponenty v rámci mikroslužby. V súčasnosti je používaná nadstavba Spring Frameworku, Spring Boot pre automatizáciu konfigurácie základných komponentov. Spring Framework a Spring Boot spoločne tvoria základný súbor knižníc pre tvorbu aplikácií touto technológiou.

The Spring Environment, je abstrakciou ktorá reprezentuje miesto pre dodatočnú konfiguráciu všetkých komponentov Spring aplikácie. Pre celkovú architektúru prostredia Spring Enviroment viz. Obr. 2.1.

Toto prostredie je primárne reprezentované v rámci aplikačného adresáru položkou application.properties v zložke resources. Slúži, ale aj na centralizáciu konfiguračných nastavení Java Virtual Machine systémových premenných, environmentálnych premenných operačného systému a argumentov príkazového riadku.

The Spring Environment zabezpečuje dostupnosť jednotlivých konfiguračných nastavení pre všetky komponenty nachádzajúce sa v Spring application context. Najpoužívanejšou lokáciou konfigurácie komponentov je vyššie uvedená položka application.properties, v ktorej je možné konfigurovať napríklad nastavenia spojené s pripojením k databázovému systému, nastavovaním aplikačného URL, prípadne URL iných integrovaných nástrojov s danou mikroslužbou a mnohé ďalšie oblasti konfigurácie potrebné pre zabezpečenie funkcionality jednotlivých komponentov a celej aplikácie.



Obrázok 2.1 Reprezentácia Spring Environment
Zdroj: [Craig Walls, s. 116]

Obr 2.1 zobrazuje základnú architektúru a vzťah medzi Spring Environment a kontajnerom Spring application context. Pod názvom „Property Sources“ sú zobrazené všetky dostupné konfiguračné vstupy.

Časť zobrazenia „Beans in the Spring application context“ zobrazuje ilustračné komponenty, ktoré využívajú konfiguračných premenných charakterizovaných v The Spring Environment. Patria sem všetky typy komponentov, či už services, repositories alebo controllers. Pre bližší popis komponentov viz. kap. 2.2.2

Spring Framework má rozsiahlu funkcionalitu a skladá sa z mnohých knižníc, ktoré podporujú a zabezpečujú implementáciu rôznorodej funkcionality v rámci celkového procesu tvorby aplikácie.

Medzi základné skupiny knižníc patrí:

- Spring Framework, čo je skupina základných knižníc určená na prácu so Spring Frameworkom. Pokrýva technologické elementy definované na začiatku tejto kapitoly,

- Spring Boot, v súčasnosti považovaný za štandard pri tvorbe aplikácií založených na tomto frameworku. Je mu venovaná samostatná podkapitola č. 2.2.2,

- Spring Data, zastrešujúce súbor knižníc zabezpečujúcich prístup k dátovej základni a jej správu, či sa už jedná o relačné alebo iné systémy riadenia báze dát, ako napríklad NoSQL. Táto nadstavba Spring Frameworku využíva aj Hibernate, čo je softvérový framework umožňujúci objektovo – relačné mapovanie,

- Spring Security, rozsiahly súbor knižnícslúžiacich na zabezpečenie aplikácie, poskytujúci možnosť nastavenia autentifikačných a autorizačných pravidiel.

- Spring AMQP, určený na integráciu Spring Frameworku s externými aplikáciami určenými na rôzne formy messagingu. Na túto integráciu využíva AMQP (Advanced Messaging Queuing Protocol) protokol. Patrí sem aj RabbitMQ, asynchrónny message broker, ktorému ju venovaná samostatná kapitola č.2.4.

Spring Cloud a Spring Cloud Foundry, skupiny knižníc určené na správu distribuovaných aplikácií v cloudovom prostredí. Tieto knižnice podporujú jeden zo základných cieľov použitia architektúry mikroslužieb – vývoj aplikácií schopných a kompatibilných s prácou v cloudovom prostredí.

Spring Framework ponúka aj ďalšie súbory knižníc, medzi ktoré patria Spring Integration, Spring Batch, Spring Mobile, Spring Web Services, Spring Reactor a ďalšie.

2.2.2 **Spring Boot**

Spring Boot poskytuje Spring Frameworku schopnosť automatickej detekcie komponentov v mikroslužbe. Táto možnosť funguje na základe analýzy štruktúry kódu triedy reprezentujúcej daný komponent. Spring Boot automaticky analyzuje triedy podľa použitých knižníc, deklarovaných premenných, umiestnenia v adresátovej štruktúre a ich názvu.

Odporúčaním je však stále anotovať triedy reprezentujúce jednotlivé komponenty odpovedajúcou anotáciou, ktorá explicitne informuje Spring application context o ich existencií.

Medzi základné anotácie reprezentujúce jednotlivé komponenty patrí:

- @Component – základná anotácia informujúca kontajner o existencií aplikačného komponentu. Používajú sa však zvyčajne nižšie uvedené špecifické anotácie,

- @Service – anotácia informujúca kontajner o druhu komponentu zvaného služba. Tento komponent v sebe implementuje business logiku mikroslužby,

- @Repository – anotácia informujúca kontajner o druhu komponentu vyjadrujúceho repozitár. Tento komponent slúži napríklad na vytvorenie prístupovej vrstvy pre databázu mikroslužby,

- @Controller – anotácia informujúca kontajner o komponente nazvanom kontrolér. Tento komponent má za účel vystavovanie a prácu s API danej mikroslužby.

2.2.3 **Java**

Jeden z najvýznamnejších a najpoužívanejších programovacích jazykov súčasnosti. Za hlavného architekta programovacieho jazyka Java sa považuje James Gosling. Práva tohto programovacieho jazyka sú v súčasnosti vlastnené spoločnosťou Oracle.

Pôvodný fundamentálny zámer pri tvorbe daného jazyka bola schopnosť prenositeľnosti medzi rôznymi technologickými platformami a podpora programovania distribuovaných aplikácii. Rozvoj tohto programovacieho jazyka nastal pri rozširovaný technologickej funkcionality internetových technológií, ktoré výrazne zvýšili požiadavky na prenositeľnosť programov.

Základ technologického fungovania programovacieho jazyka Java, ako ho definuje Schildt(2016, s. 27) *„Kompilátor Java na výstupe neprodukuje spustiteľný kód, ale bajtový kód. Bajtový kód sa skladá z vysoko optimalizovanej sady inštrukcii, ktoré sú určené k spúšťaniu v systéme runtime jazyka Java označovanom ako modul JVM (Java Virtual Machine).“*

Java Virtual Machine následne interpretuje tento bajtový kód na strojový kód určený špecificky pre platformu na ktorej sa JVM nachádza.

Tento technologický koncept, na ktorom je programovací jazyk Java založený, zabezpečuje spustiteľnosť Java kódu na každej platforme implementujúcej modul JVM.

Medzi základne vlastnosti jazyka Java patrí:

- Objektovo orientovaný prístup, programová paradigma, ktorá pracuje s triedami a objektmi ako so základnými prvkami programu. Objekty môžu obsahovať rôzne druhy dát a programového kódu a vzájomne medzi sebou interagovať. V programovacom jazyku Javu sú objekty reprezentované inštanciou triedy reprezentujúcou daný objekt.

Medzi významné koncepty objektovo orientovaného prístupu patrí dedičnosť, enkapsulácia a polymorfyzmus,

- Prenositeľnosť,
- Schopnosť pracovať s viacerými programovými vláknami,
- Zabezpečenie distribuovanosti aplikácií,
- Podpora viacerých programových prístupov – štruktúrovaný, imperatívny prístup.

Programovací jazyk Java existuje vo forme niekoľkých základných verzií určených pre použitie na rôznych platformách, zaraďujeme sem:

- **Java Platform, Micro Edition (Java ME),** verzia určená primárne pre mobilné zariadenia a embedované systémy, patria sem napríklad senzory, tlačiarne, mobilné telefóny, digitálni asistenti...,
- **Java Platform, Standart Edition (Java SE),** štandardná verzia obsahujúca základné knižnice programovacieho jazyka Java. Jedná sa o verziu určenú pre použitie na tvorbu aplikácií na strane desktopu aj serveru. Najznámejšou implementáciou Java Standart Edition je Java Development Kit (JDK), od spoločnosti Oracle. Najnovšia verzia programovacieho jazyka Java vo verzií Java SE je 12,
- **Java Platform, Enterprise Edition (Java EE),** verzia slúžiaca primárne na tvorbu distribuovaných serverových aplikácií, pracujúcich v paralelnom výpočtovom prostredí. Jedná sa o rozšírenie Java SE o technológie určené na tvorbu webových aplikácií.

2.2.4 **Maven**

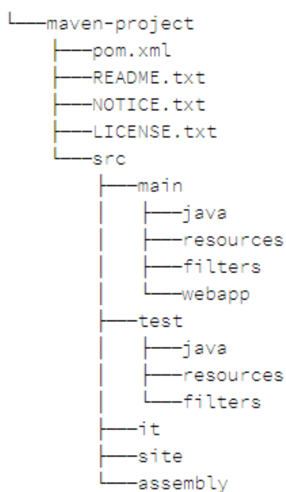
Maven môžeme charakterizovať ako nástroj pre podporu softvérového projektového manažmentu. Vývoj nástroja Maven prebieha formou open-source nadáciou The Apache Software Foundation. Podobný nástroj využívaný hlavne z účelu budovania aplikácií pre mobilný operačný systém Android je Gradle.

Základnou úlohou nástroja Maven je automatizovaná tvorba a štandardizácia zostavenia (build) aplikácie. Tento proces zahŕňa prevádzanie samotných programov
a ich dokumentácie zo zdrojového stavu do cieľového skompilovaného stavu, teda na formu samostatne spustiteľných softvérových artefaktov, či už vo formáte JAR(Java Archive) alebo WAR(Web Application Archive).

Maven poskytuje schopnosť riadiť tento proces zostavenia aplikácie z centralizovaného miesta. Štruktúra potrebná pre centralizáciu riadenia zostavenia
sa nazýva objektový model projektu (POM – project object model). Táto centralizovaná oblasť je v rámci adresárovej štruktúry aplikácie vyjadrená názvom POM a nachádza sa vo formáte jazyka XML(Extensible Markup Language).

Ďalšou významnou funkcionalitou poskytovanou nástrojom Maven je riadenie a správa knižníc. Táto funkcionalita je spravovaná konfiguráciou súboru pom.xml v adresári aplikácie.

Projekty využívajúce funkcionalitu nástroja Maven majú špecifickú štruktúru programového adresára, viz Obr. 2.2



Obrázok 2.2 Adresárová štruktúra pri použití nástroja Maven
Zdroj[5]

Jednotlivé adresárové zložky obsahujú špecifické súbory zabezpečujúce funkcionalitu celého projektu, patrí sem:

- pom.xml – súbor vo formáte xml obsahujúci objektový model projektu.,
- README.txt – voliteľný textový súbor obsahujúci sumarizáciu projektu,
- NOTICE.txt – voliteľný textový súbor obsahujúci dodatočné informácie o knižniciach tretích strán,
- LICENSE.txt – voliteľný textový súbor obsahujúci informácie o licencií projektu,
- it, site, asembly – voliteľné súbory pre dodatočnú konfiguráciu,
- src – zdrojová zložka projektu,

- **src/main** – zložka obsahujúca zdrojový kód a všetky položky, ktoré budú po kompilácií súčasťou vytvoreného artefaktu vo formáte JAR alebo WAR,

- src/main/java – zložka obsahujúca triedy v programovacom jazyku Java,

- src/main/resources – zložka obsahujúca konfiguračné súbory, patrí sem aj súbor application.properties slúžiaci na konfiguráciu The Spring Environment, konceptu bližšie popísaného v kap. 2.2.1

- src/main/filters – voliteľna zložka obsahujúca konfiguráciu filtrov,

- src/main/webapp - voliteľná zložka obsahujúca súbory jazyka HTML, CSS, Javascript pre prípadnú tvorbu GUI(Graphical User Interface),

- **src/test –** zložka obsahujúca všetok kód potrebný pre testovanie danej mikroslužby a prípadne ďalšie konfigurovateľné zdroje potrebné k testovaniu,

- src/test/java – zložka obsahujúca triedy programovacieho jazyka Java určených na testovanie mikroslužby,

- src/test/resources **–** zložka obsahujúca konfiguračné súbory potrebné pre testovanie,

- srct/test/filters – voliteľná zložka obsahujúca konfiguráciu filtrov použitých pri testovaní mikroslužby,

2.2.4.1 **Štruktúra pom.xml**

Implementácia a konfigurácia funkcionalít objektového modelu projektu pomocou nástroja Maven je prevádzaná primárne v súbore pom.xml, tento súbor obsahuje,

- základnú softvérovú projektovú konfiguráciu,
- zoznam implementovaných závislostí (dependency),
- zoznam implementovaných pluginov.

Do základnej projektovej konfigurácie patrí nastavenie,

- name, vyjadrujúce názov nami vyvíjanej mikroslužby,
- description, vyjadruje popis vyvíjanej mikroslužby,
- groupId, určuje jedinečný identifikátor mikroslužby, napríklad plne kvalifikovaný názov programového balíčka - com.tieto.tds,
- artifactId, určuje špecifický artefakt v rámci groupId. Artefakt vyjadruje čokoľvek produkované daným Maven Projektom.
- packaging, označujúcí výsledný formát artefaktu produkovaného projektom. Môže sem patriť JAR alebo WAR.

Základná konfigurácia poskytuje možnosť konfigurovať aj **rodičovský (parent) projekt** súčasného projektu. Súčasný projekt následne automaticky preberá všetky závislosti deklarované v rodičovskom projekte. Rodičovský projekt musí byť označený
minimálne vlastnosťami groupID a artifactId. Tento prístup sa používa pri vývoji viacerých mikroslužieb alebo iných komponentov, ktoré vychádzajú z určitého vopred zadefinovaného základu. Hlavnou výhodou tohto prístupu je lepšia správa vzájomných závislosti jednotlivých častí celkovej aplikácie a z toho plynúca časová efektivita tohto prístupu.

Štruktúra deklarovania závislostí nástroja Maven je vyjadrená jazykom XML. Závislosť vyjadruje externú knižnicu implementovanú mikroslužbou. Na rozdiel od bežného importovania knižníc, avšak Maven automaticky importuje knižnicu zo svojho externého repozitára knižníc.

Každá závislosť musí mať definované minimálne parametre groupId a artifactId, aby bola jednoznačne určiteľná. Takto deklarovaná závislosť je následne automaticky spravovaná nástrojom Maven.

2.2.5 Lombok

Lombok je knižnica umožňujúca deklarovať určité veľmi často sa vyskytujúce metódy triedy, objektu a rôzne druhy konštruktorov pomocou anotácii.

Anotácia **@Data** zabezpečuje deklaráciu základných metód triedy – getter, setter, toString, prepis hash a equals. Pri použití tejto anotácie sú vyššie menované metódy automaticky generované bez priameho viditeľného zápisu daného kódu, pracujú avšak rovnako ako pri normálnom zápise.

Anotácia **@RequiredArgsConstructor** zabezpečuje generovanie bežného parametrizovaného konštruktoru všetkých atribútov s modifikátorom final, alebo anotáciou **@NonNull**, ktorá je v tomto vyjadrení používaná v rámci celej mikroslužby. Atribúty s modifikátorom transient a tie, ktoré sú priamo inicializované v rámci deklarácie nebudú zahrnuté v tomto konštruktore.

Anotácia **@NoArgsConstructor** je určená na generáciu bezparametrového konštruktoru objektu.

2.3 Technológie určené pre prácu s databázou

Mikroslužba bude udržiavať spojenie s databázou, do ktorej budú zapisované informácie o jednotlivých e-mailoch a ich prílohách. Ako systém riadenia báze dát bude použitý objektovo – relačný databázový systém PostgreSQL. Spojenie bude naviazané pomocou The Spring Environment. Samotná práca s databázou bude zabezpečená pomocou komponentu repository. Pre reprezentáciu entít bude použitá technika objektovo-relačného mapovania.

2.3.1 **PostgreSQL**

Objektovo-relačný databázový systém, dostupný na platformách operačných systémov Windows, Linux a MacOS. Tento databázový systém je vyvíjaný skupinou PostgreSQL Global Development Group, tvorenou viacerými spoločnosťami a jednotlivcami. Jedná sa o open-source softvér s existujúcou skupinou spoločností poskytujúcich spoplatnenú aplikačnú podporu.

Základom tohto databázového systému je štandardizovaný štruktúrovaný dopytovací jazyk (SQL – Structured Query Language), slúžiaci na základnú manipuláciu a správu štruktúrovaných dát v databázovom systéme.

PostgreSQL plnohodnotne podporuje databázové transakcie s vlastnosťami ACID. Transakcia reprezentuje konečnú postupnosť operácií prevedených v rámci databázového systému.

Tieto vlastnosti reprezentuje

- atomocita, zabezpečujúca vyjadrenie transakcie ako jedinej samostatnej atomickej operácie, ktorá je buď úspešne vykonaná ako celok alebo nie je vykonaná vôbec

- konzistencia, vďaka ktorej transakcia vždy prevádza databázový systém z jedného konzistentného stavu do druhého povolenými spôsobmi. V prípade neúspešného prevedenia transakcie je databáza vrátená do konzistentného stavu v ktorom sa nachádzala pred zahájením danej transakcie,

- izolovanosť, zaisťujúca súbežný priebeh viacerých transakcií v rovnakom čase, zanechá databázový systém v rovnakom stave, v akom by bol v prípade, že by všetky transakcie boli vykonávané samostatne,

- trvalosť, vďaka ktorej ukončené transakcie zaznamenajú svoje výsledky na energeticky nezávislé pamäťové médium, čím chráni uchovanie výsledkov transakcie v prípade systémovej poruchy, napríklad výpadku prúdu.

2.3.2 **Objektovo-relačné mapovanie**

Objektovo relačné mapovanie (ORM), je programovacia technika používajúca objektovo orientovaný programovací prístup na konverziu záznamov relačnej databáze na objekty objektovo orientovaného programovacieho jazyka.

Hlavnou úlohou tohto prístupu je zabezpečiť schopnosť konverzie objektov na požadovanú formu, ktorú je možné uložiť v relačnom databázovom systéme. Zároveň musia byť zachované všetky vlastnosti objektov a vzájomné vzťahy medzi objektmi tak, aby dokázali reprezentovať premenné a definované vzťahy relačnej databáze a boli vzájomne konvertovateľné.

Uvedené objekty tým pádom existujú v špecifickom stave – relačnej databáze, aj po ukončení aplikačného procesu, v rámci ktorého boli vytvorené. Objekty spĺňajúce tieto vlastnosti sa nazývajú **perzistentné**.

Technologická implementácia objektovo relačného mapovania existuje vo viacerých úrovniach. Medzi základné technológie patrí Java Persistence API (JPA) a objektovo relačný framework Hibernate.

2.3.2.1 **Java Persistence API**

JPA je rozhranie programovacieho jazyka Java určené na správu relačného databázového systému v rámci aplikácie. Toto rozhranie je dostupné ako súčasť platformy Java SE a Java EE.

Reprezentácia v programovacom jazyku Java je zaistená knižnicou *javax.persistence.*
Toto aplikačné rozhranie obsahuje základnú definíciu anotácií potrebných na objektovo relačné mapovanie aplikácie.

Distribúcia rozhrania je riešená pomocou externých služieb, medzi najznámejšie patrí napríklad EclipseLink, OpenJPA a framework pre objektovo relačné mapovanie Hibernate.

2.3.2.2 **Hibernate**

Hibernate patrí medzi objektovo relačné frameworky. Vývoj tohto frameworku je zastrešovaný spoločnosťou Red Hat.

Framework poskytuje možnosť tvorby perzistentných dátových objektov technikou objektovo relačného mapovania. Hibernate implementuje Java Persistence API, vďaka čomu je používanie tejto technológie možné na akejkoľvek platforme podporujúcej Java SE a aplikačnom serveru Java EE.

Hibernate obsahuje aj radu vlastných funkcionalít a vysokú úroveň flexibility a konfigurovateľnosti. Na prístup k relačnému databázovému systému používa techniku JDBC (Java Database Connectivity).

Základným prvkom tvorby objektovo relačného mapovania je tvorba triedy entity. Táto trieda reprezentuje perzistentný objekt spojený s tabuľkou relačného databázového systému. Mapovanie je implementované pomocou anotácií. Tieto anotácie sú deklarované v rámci rozhrania JPA.

Ako uvádza *Walls(2018), trieda entít musí spĺňať niekoľko základných požiadaviek, medzi ktoré patrí:*

- *Trieda reprezentujúca perzistovaný objekt musí byť anotovaná @Entity,*
- *Musí obsahovať bezparametrový konštruktor,*
- *Atributy triedy musia mať prístupový modifikátor private a na prístup k nim musia byť použité get a set metódy – enkapsulácia,*
- *Trieda reprezentujúca perzistovaný objekt musí obsahovať atribút reprezentujúci primárny kľúč asociovanej tabuľky relačného databázového systému, tento atribút musí byť anotovaný @Id.*

Spring Framework využíva Hibernate v súbore knižníc Spring Data, ktoré sú určené na správu databázovej funkcionality aplikácií.

2.3.3 Tvorba prístupového API k záznamom databáze

Prístup k záznamom z databázy s e-mailovými správami a prílohami, ktoré boli vytvorené touto mikroslužbou musí byť na základe požiadaviek možný pomocou protokolu HTTP.

Vždy je možný prístup k záznamom v danom databázovom systéme priamo pomocou databázového rozhrania, v tomto prípade má PostgreSQL svoj vlastný GUI nástroj, s ktorým môžeme pristupovať priamo k zdroju databáze.

Spring Framework obsahuje v rámci súboru knižníc Spring Data rozhranie Spring Data JPA, ktoré zabezpečuje vytvorenie a konfiguráciu API , pomocou ktorého bude môcť strana klienta získavať záznamy z databázového systému mikroslužby vo formáte JSON. Formát JSON sa používa ako štandardný prenosový dátový formát v architektúre orientovanej na mikroslužby.

Automatizované no zároveň konfigurovateľné vystavovanie API bude zabezpečovať aplikačný komponent *Repository,* respektíve jeho špecifická implementácia JPA Repository.

2.3.3.1 JPA Repository

Aplikačný komponent JPA repository slúži na vytvorenie programového mechanizmu implementácie objektovo - relačného mapovania. JPA repository je súčasťou knižnice Spring Data JPA.

Potom, čo vytvoríme triedy entít reprezentujúce perzistované objekty relačného databázového systému, môžeme použiť JPA repository, ktoré automaticky zabezpečí :

- vytvorenie metódy umožňujúcej ukladanie perzistovaných objektov do relačného databázového systému,

- automatické vytvorenie koncových bodov API, ktoré umožňujú svojim použitím prevádzať základné databázové operácie. Tieto operácie špecificky reprezentujú na strane databáze príkazy INSERT, SELECT, UPDATE, DELETE,

- Prenos všetkých objektov dátovým formátom JSON protokolom HTTP,

- Schopnosť číslovania a zoraďovania výsledkov operácie SELECT na strane klienta.

Štruktúra JPA repository musí spĺňať nasledovné charakteristiky:

- programová implementácia komponentu repository je riešená pomocou rozhrania,

- rozhranie je anotované anotáciou @Repository,

- toto rozhranie využíva vzťahu dedičnosti s rozhraním CrudRepository,

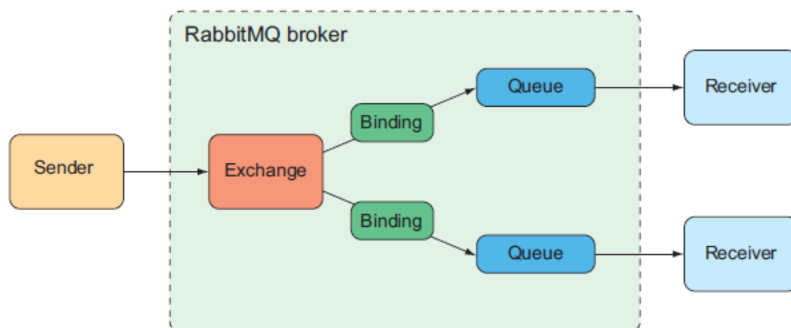
- CrudRepository<T, ID> má dva základne parametre a to parameter T, reprezentujúci objekt entity, ktorá bude perzistovaná do databázy a parameter ID, vyjadrujúci dátový typ primárneho kľúča danej entity deklarovaného v triede reprezentujúcej entitu. Názov CRUD vyjadruje skratku pre slová Create, Read, Update a Delete. Jedná sa o pomenovanie databázových príkazov INSERT, SELECT, UPDATE a DELETE.

- Mikroslužba musí mať prístup ku knižnici Spring Data JPA. Implementáciu tejto knižnice prevádza softvérový nástroj Maven po importovaní závislosti *spring-boot-starter-data-jpa* pomocou súboru pom.xml.

2.4 **RabbitMQ**

RabbitMQ môžeme charakterizovať ako softvérový nástroj určený na implementáciu procesu asynchrónnej komunikácie, poskytuje teda možnosť prenosu správ asynchrónnym spôsobom medzi jednotlivými aplikáciami a programovými komponentmi, v našom prípade mikroslužbami.

Asynchrónne preposielanie správ má význam v rámci tvorby aplikácie architektúrou orientovanou na mikroslužby, ako definuje *Walls(2018, s. 179) „Asynchrónne preposielanie správ zabezpečuje možnosť nepriameho preposielania správ medzi aplikáciami, prípadne ich komponentmi bez nutnosti čakania na priamu odpoveď. Tento nepriamy prístup podporuje voľnú viazanosť a lepšiu škálovateľnosť komunikácie medzi aplikáciami alebo ich komponentmi.“*



Obrázok 2.3 Základná štruktúra preposielania správ pomocou RabbitMQ
Zdroj: [Craig Walls, s. 192]

Práca s nástrojom RabbitMQ je zabezpečená pomocou súboru knižníc Spring AMQP. Pre základnú štruktúru procesu asynchrónneho rozosielania správ viz. Obr. 2.3.

„*Sender*“(odosielatel), reprezentuje klientsku aplikačnú časť komunikujúcu so softvérovým nástrojom RabbitMQ. Má schopnosť odoslať správu v špecificky definovanom formáte do „*Exchange*“. Hlavnou úlohou *Exchange* je smerovať túto správu do „*Queue*“, teda do *fronty správ*. Smerovanie nemusí prebiehať iba do jednej *fronty,* ale do všeobecného konečného počtu *front*. *Exchange* musí byť presné schopné identifikovať, do ktorých špecifických front musí danú správu zo strany *odosielateľa* smerovať.

Identifikácia front zo strany *Exchange* prebieha pomocou, tzv. „*Binding*“(vzájomného viazania). Proces vzájomného viazania medzi *Exchange* a *frontou* môže prebiehať v rôznych variantoch. Dôležitým prvkom *vzájomného viazania* je **smerovací kľuč**. Tento kľuč slúži pre informovanie *Exchange,* do ktorej špecifickej fronty má smerovať správu. Špecifická identifikácia smerovacieho kľúča prebieha na základe konfigurácie typu Exchange.

Každá fronta má aj svoj vlastný **viazací kľuč**. Tento kľuč je v základnej konfigurácií zhodný s názvom danej fronty, je ho možné dodatočne zmeniť.

Existuje rada konfigurácii Exchange, medzi základné patria ako uvádza *Walls(2018):*

- *Default – základne použitá konfigurácia automatický vytvorená nástrojom RabbitMQ. Všetky správy sú smerované do front, ktorých názov je zhodný s hodnotou smerovacieho kľúča. Všetky fronty sú automaticky previazané s počiatočnou exchange.*

- *Direct – priama konfigurácia smerujúca správy do front, ktorých viazací kľuč je rovnaký ako smerovací kľuč správy.*

- *Topic – podobná direct, viazací kľuč môže obsahovať zástupné znaky(wildcards),*

- *Fanout – exchange smeruje správy do všetkých zviazaných front bez ohľadu na použitie kľúčov,*

Potom, čo *Exchange* správne identifikuje a vykoná smerovanie prijatej správy od *odosielateľa* do *fronty*, môže byť správa prijatá na strane „*Receiver*“(prijímateľ).
Prijímanie správ z fronty bude vykonávať aj naša mikroslužba. Prijímanie správ z fronty slúži na základnom princípe fungovania dátovej prístupu fronty FIFO (First In, First Out).

Súbor knižníc Spring AMQP ponúka skupinu špecifických tried a anotácií určených na prijímanie správ z fronty.

3 Analýza požiadavkou a súčasného stavu e-mailového messagingu cloudovej služby

Požiadavka na optimalizáciu e-mailového messagingu vznikla na základe záujmu vedúcich pracovníkov projektu zodpovedajúcich za príslušnú časť softvérového vývoja o optimalizáciu súčasného softvérového procesu rozosielania e-mailov.

3.1 Zázemie poskytovanej služby

Jedná sa o IT službu Tieto DevOps Space vlastnenú a poskytovanú spoločnosťou Tieto. Základným účelom tejto služby je ponuka aplikačných a výpočtových prostriedkov pre zákazníkov, určených na tvorbu softvérových aplikácií metodikou vývoja DevOps. Tieto DevOps Space podporuje celý proces softvérového vývoja, testovania, jeho projektového riadenia a poskytuje zákaznícku podporu pre danú službu.

3.2 Súčasný stav e-mailového messagingu

Súčasná podoba e-mailového messagingu je riešená na úrovni jednotlivých mikroslužieb. Každá mikroslužba vyžadujúca odoslanie e-mailovej správy musí implementovať funkcionalitu zabezpečujúcu odoslanie potrebnej požiadavky v kompatibilnom formáte na SMTP server.

Tento spôsob priebehu daného technologického procesu vytvára potrebu tvorby duplicitného kódu v jednotlivých mikroslužbách v súčasnom a potencionálne aj budúcom stave, pri prípadnom rozšírení celkovej architektúry cloudovej služby o ďalšie mikroslužby, ktoré budú vyžadovať možnosť odosielať informačné e-maily vonkajším používateľom alebo interným pracovníkom projektu. Nie je teda v prípade projektu použitý princíp znovupoužiteľnosti ako jeden z charakteristických prístupov v rámci objektovo orientovaného paradigma.

Celkový kód zabezpečujúci e-mailový messaging je v súčasnosti zastaralým, menej efektívnym riešením, ktoré nevyužíva aktuálne programové knižnice a riešenia.

Ďalšou u nevýhod je absencia akýchkoľvek záznamov o vytvorených a odoslaných e-mailoch.

Schopnosť asynchrónneho rozosielania e-mailov pomocou RabbitMQ brokeru v nami požadovanej forme neexistuje v súčasnej verzií cloudovej služby.

3.3 Požiadavky optimalizácie

Medzi základné požiadavky patrí odstránenie nedostatkov definovaných v rozbore súčasného stavu procesu e-mailového messagingu. Po analýze jednotlivých nedostatkov boli definované postupy pre naplnenie požiadavky optimalizácie.

Funkcionalita asynchrónneho rozosielania e-mailových správ pomocou RabbitMQ brokeru bude vytvorená v rámci našej mikroslužby.

Výhodou celkového riešenia bude podobný zámer oboch funkcionalít a schopnosť využitia jednotného databázového modelu, do ktorého môžeme ukladať vytvorené a odoslané e-mailové správy.

Rozdielom je forma komunikácie pri príjme entít, z ktorých bude mikroslužba vytvárať a odosielať e-maily v kompatibilnej forme na SMTP server. Preto je veľmi dôležité zabezpečiť schopnosť komunikácie medzi jednotlivými mikroslužbami v rámci celkového projektu pomocou technológie REST a protokolu HTTP a zároveň integrovať mikroslužbu s externou aplikáciou RabbitMQ brokeru.

Samotný zámer vytvorenia novej mikroslužby so sebou prináša nové požiadavky vyplývajúce zo základnej koncepcie a fungovania softvérovej architektúry mikroslužieb. Jedná sa o architektonický zámer, ktorému vyhovuje spoločný dátový model oboch funkcionalít poskytovaných danou mikroslužbou . Všeobecná požiadavka na otestovanie komponentu pomocou unit testov je súčasťou tvorby každej novej mikroslužby prítomnej v projekte.

Medzi jednotlivé časti celkového postupu pri optimalizácií patria:

1. Vytvorenie samostatnej mikroslužby, ktorá bude poskytovať funkcionality potrebné pre optimalizáciu procesu e-mailového messagingu.

2. Zabezpečenie správnej komunikácie jednotlivých mikroslužieb v prípade záujmu o odoslanie e-mailu, pomocou architektúry REST.

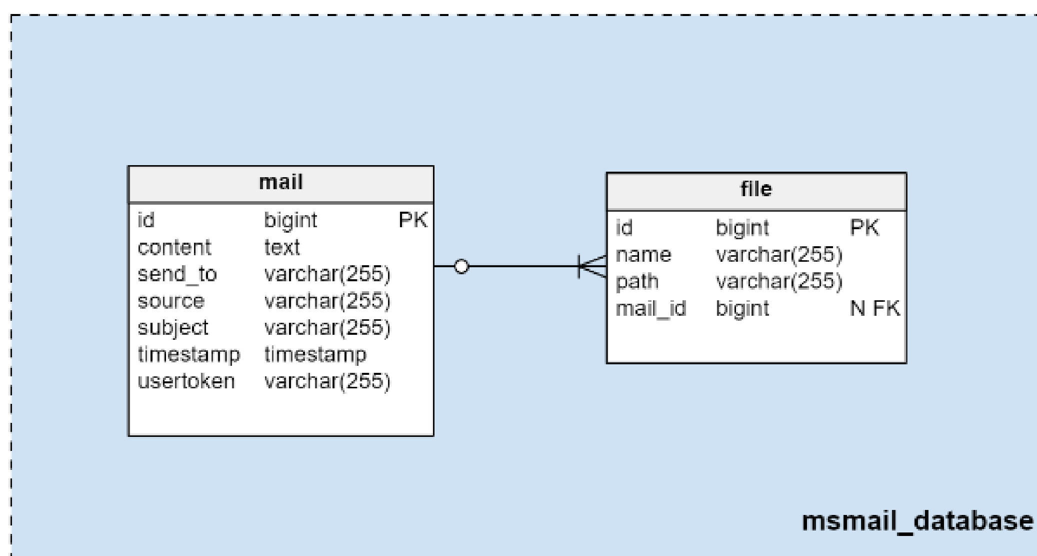
3. Integrácia mikroslužby so softvérovým nástrojom RabbitMQ broker.

4. Návrh a implementácia relačného databázového modelu fungujúceho na abstraktnej úrovni pod danou mikroslužbou, v rámci záujmu naplnenia požiadaviek o uchovávaní záznamov odoslaných e-mailov a ich prípadných príloh.

5. Testovanie danej mikroslužby pomocou unit testov na úroveň požadovanú zadávateľom.

4 Implementácia riešení optimalizácie

4.1.1 Entity Relationship Diagram databáze mikroslužby



Obrázok 4-1 Entity Relationship Diagram databáze mikroslužby
Zdroj: vlastné vypracovanie

Relačná databáza je tvorená dvoma základnými tabuľkami. Tabuľka mail reprezentuje prijatú e-mailovú správu danou mikroslužbou. Daná tabuľka má automaticky generovaný primárny kľúč **id**, rovnako tak tabuľka file obsahuje svoj vlastný automaticky generovaný primárny kľúč **id**.

Jednotlivé atribúty reprezentujú základné vlastnosti či už prijatej e-mailovej správy alebo jej príloh ktoré je nutné ukladať do relačnej databáze.

Vzťah medzi tabuľkami môžeme charakterizovať:

- Kardinalitou one-to-many, kde cudzí kľúč tabuľky file s názvom **mail_id** je tvorený primárnym kľúčom tabuľky mail **id**,
- Voliteľným vzťahom zo strany tabuľky mail, keďže daná e-mailová správa môže a nemusí mať žiadne prílohy.

4.2 Triedy a rozhrania mikroslužby

4.2.1 Controller

Kontrolér zabezpečuje príjem http správy s potrebnými parametrami. Určuje presnú URL adresu http požiadavku, na ktorej bude požiadavka spracovaná a metódu jeho spracovania.

Trieda **MainController** reprezentuje daný kontrolér mikroslužby.

```
@RestController
@RequestMapping("/api")
@Slf4j
public class MainController {
    private MainService mainService;
  @Autowired
  public MainController(MainService mainService) {
     this.mainService = mainService;
  }
  @PostMapping("/mails/send")
  public ResponseEntity send(
        @RequestParam("usertoken") String usertoken,
        @RequestParam("sendTo") String sendTo,
        @RequestParam(required = false, name= "subject") String subject,
        @RequestParam("content") String content,
        @RequestParam(required = false, name = "files") List<MultipartFile> multipartFiles)
{

     log.debug("{}, {}, {}, {}", usertoken, sendTo, subject, content);
    return mainService.processMessage(usertoken, sendTo, subject, content, multipartFiles);
  }
}
```

Anotácia **@RestController** označuje triedu ako kontrolér pre jej detekciu Spring Application Contextom. Rest reprezentuje špecifický typ kontrolóra použitý na čistú komunikáciu bez navrátenia odkazu na webovú stránku.

@RequestMapping("/api") definuje základnú formu URL, na ktorej budú prijímané jednotlivé http požiadavky v rámci mikroslužby.

@Slf4j Je anotácia ktorá automaticky vytvára bez explicitnej deklarácie atribút triedy
private static final Logger log = LoggerFactory.getLogger(MainController.class);
Jedná sa o logovací nástroj používaný naprieč celou mikroslužbou slúžiaci na logovanie, primárne výnimiek, v prípade triedy **MainController** zaznamenáva základné informácie o jednotlivých parametroch prijatých v rámci http požiadavky, čo je vyjadrené volaním metódy **log.debug("{}, {}, {}, {}", usertoken, sendTo, subject, content)**

Privátny atribút triedy **private MainService mainService** odkazuje na objekt triedy MainService, v ktorej je implementovaná základná logika mikroslužby a je popísaná v kapitole č.4.2.6.

Metóda **mainService.processMessage(usertoken, sendTo, subject, content, multipartFiles)** je určená na spracovanie atribútov http požiadavku prijatého kontrolérom.

Konštruktor kontrolóra má za parameter priamo objekt MainService a je anotovaný anotáciou **@Autowired.** Jedná sa o techniku vkladania závislostí, v tomto prípade pomocou konštruktoru triedy vytvoríme závislosť s triedou MainService, pre jej použitie popísané v predchádzajúcom odstavci.

Metóda **send** je anotovaná **@PostMapping("/mails/send"),** čo zabezpečí, že na url adrese api/mails/send budú prijaté požiadavky iba metódou post. Jednotlivé parametre metódy send anotované **@RequestParam("názovAtributu"),** reprezentujú atribúty http požiadavku a mapujú ich do jednotlivých parametrov metódy send.

Voliteľná hodnota required v parametroch anotácie **RequestParam** slúži na označenie daného atribútu ako voliteľného. Východzia hodnota je nastavená na true, čo znamená, že v prípade kedy http požiadavka neobsahuje atribút s daným názvom alebo je prázdny, je automaticky vyhodená výnimka, čo nie je požadovaná vlastnosť pri tvorbe e-mailu, v prípade atribútov subject, ktorý reprezentuje predmet danej správy a atribútu files, ktorý reprezentuje prílohu danej správy, čo sú parametre ktoré môžu alebo nemusia byť prítomné v prijatom požiadavku a koncovej e-mailovej správe.

Návratový typ ResponseEntity zabezpečuje vrátenie špecifikovanej HTTP odpovede o úspešnom poslaní daného e-mailu na stranu klienta.

4.2.2 **Objekty pre prenos dát**

Mikroslužba príjme správu pomocou http požiadavky definovanej triedou MainController, popísanou v kap.4.2.2 alebo pomocou RabbitMQ brokeru a AMQP protokolu v rámci triedy MailMessageListener popísanej v kap.4.2.5.1

Po prijatí správy musí mikroslužba zo správy vytvoriť objekt, obsahujúci požadované atribúty a následne objekt odoslať na SMTP server.

Požiadavky definujú zároveň aj nutnosť uloženia záznamu o správe v relačnej databáze, pričom použijeme techniku objektovo relačného mapovania, ktorá vyžaduje na svoju funkcionalitu taktiež objekt. Formát objektov potrebných pre tieto dve úlohy je rozdielny. Balíček *dto* obsahuje triedy reprezentujúce práve objekty určené na transport medzi prijatými správami a jej odoslaním na SMTP server, zatiaľ čo balíček entity popísaný v kap.4.2.3, obsahuje triedy reprezentujúce objekty určené na uloženie do relačnej databáze s účelom vytvorenia záznamu o prijatých správach.

4.2.2.1 **MailMessageDto**

Trieda MailMessageDto reprezentuje správu, ktorá bude odoslaná na stranu SMTP serveru.

```
@Data
@RequiredArgsConstructor
@NoArgsConstructor
public class MailMessageDto {
@NonNull
private String usertoken;
@NonNull
private String sendTo;
@NonNull
private String subject;
@NonNull
private String content;
@NonNull
private List<File> files;}
```

Obsahuje atribúty typu String reprezentujúce jednotlivé parametre danej správy. Posledný parameter **files**, je kolekcia List, v ktorej sa budú ukladať súbory typu **File**, reprezentujúce prípadné prílohy e-mailovej správy, ak budú existovať. Trieda **File** reprezentujúca dané súbory sa nachádza v balíčku entity.

Anotácie **@Data** a **@RequiredArgsConstructor** pochádzajú z externej knižnice Lombok. Pre popis ich funkcionalít viz kap.2.2.5

Posledná anotácia triedy **@NoArgsConstructor** je určená na generáciu bezparametrového konštruktoru, ktorý sa po generácií parametrizovaného konštruktoru v rámci triedy automaticky nenachádza a je neskôr využitý v triede MainService.

4.2.2.2 **RabbitMailMessage**

RabbitMailMessage je trieda reprezentujúca prijatú správu z brokera RabbitMQ. Správa z nástroja RabbitMQ je vo formáte JSON. Táto trieda zabezpečuje konverziu takto prijatej správy do objektu danej triedy. S týmto objektom môžeme následne pracovať a použiť ho na uloženie danej správy do databázy a jej odoslanie na SMTP server. Odosielanie a ukladanie tejto správy prebieha pomocou objektu triedy RabbitMailMesage popísaného v predchádzajúcej kapitole a pomocou tried entity MailMessage a File. Celkový postup tohto procesu je implementovaný v triede MainService.

```
@Data
@NoArgsConstructor
@RequiredArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
public class RabbitMailMessage {
    @NonNull
    private String userToken;
    @NonNull
    private String sendTo;
    @NonNull
    private String subject;
    @NonNull
    private String content;
    private List<RabbitJsonFile> rabbitJsonFiles;
}
```

Anotácie @Data, @NoArgsConstructor a @RequiredArgsConstructor pochádzajú z nástroja Lombok.

Anotácia **@JsonIgnoreProperties(ignoreUnknown = true)** zabezpečuje, že v prípade príchodu správy s neznámymi JSON kľúčmi alebo hodnotami nebudú tieto kľúče a hodnoty konvertované do objektu RabbitMailMessage.

Kolekcia List typu RabbitJsonFile je určená na ukladanie objektov typu RabbitJsonFile, ktoré sa môžu nachádzať v správe prijatej z fronty RabbitMQ. Jedná sa o bity súborov prekonvertované na String pomocou špecifického formátu base64 na strane klienta. Po prijatí danej správy je nutné prekonvertovať tieto súbory z formátu String na formát file, čo je možné v rámci použitia formátu base64.

4.2.2.3 RabbitJsonFile

Trieda reprezentujúca objekty súborov prijaté zo správ fronty RabbitMQ. Parameter **name** vyjadruje názov daného súboru, parameter **path** vyjadruje absolútnu cestu k fyzickej lokácií súboru prijatej našou mikroslužbou. Parameter **content** reprezentuje base64 bitový formát vyjadrený vo forme Stringu, jedná sa teda o prekonvertované súbory reprezentujúce prílohy e-mailov na strane klientskych mikroslužieb. Na strane našej mikroslužby sú tieto súbory vyjadrené vo forme Stringu spätne prekonvertované na súbory typu java.io.File.

```
@Data
@NoArgsConstructor
@RequiredArgsConstructor
public class RabbitJsonFile {
   @NonNull
   private String name;
   @NonNull
   private String path;

   @NonNull
   private String content;
}
```

4.2.3 **Objekty entít pre ukládanie do relačnej databáze**

Objekty entít sú použité na ukladanie záznamov e-mailov spracovaných mikroslužbou do relačnej databáze pomocou techniky objektovo – relačného mapovania.

4.2.3.1 **EntityMailMessage**

```
@Data
@Entity
@Table(name = "mail")
@Relation(value = "EntityMailMessage", collectionRelation = "EntityMailMessages")
@RestResource(rel = "mails", path = "mails")
@NoArgsConstructor
@RequiredArgsConstructor
public class EntityMailMessage {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;
    @NonNull
    @Column(name = "usertoken")
    private String userToken;
    @NonNull
    @Column(name = "send_to")
    private String sendTo;
    @NonNull
    @Column(name = "subject")
    private String subject;
    @NonNull
    @Column(name = "content", columnDefinition = "Clob")
    private String content;
```

```
    @NonNull
    @Column(name = "timestamp")
    private Timestamp timestamp;
    @NonNull
    @Column(name="source")
    private String source;
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "mail_id")
    private List<EntityFile> entityFiles;

    public void addFile(EntityFile entityFile) {
        if (entityFiles == null) {
            entityFiles = new ArrayList<>();
        }
        entityFiles.add(file);
    }
}
```

Trieda obsahuje anotácie:

- **@Entity,** označujúcu danú triedu ako triedu entity, určenú na perzistovanie objektov do relačnej databázy.
- **@Table(name="mail")** označuje názov príslušnej tabuľky relačnej databázy, v rámci ktorej bude prebiehať perzistovanie objektov.
- **@Relation(value = "EntityMailMessage", collectionRelation = "EntityMailMessages"),** určujúcu presné generovanie názvu pomocou rozhrania Spring Data JPA, ktoré by v pôvodnej hodnote nemuselo reprezentovať presný názov triedy. V tomto prípade generované API vracajúce jediný záznam vytvára zápis v zmysle /api/...EntityMailMessage, ak je návratná hodnota kolekcia, resp. skupina záznamov, jedná sa o /api/...EntityMailMessages,

- **@RestResource(rel = "mails", path = "mails"),** určuje kompozíciu odkazov v štruktúre JSON po použitý daného API,
- **@NoArgsConstructor, @RequiredArgsConstructor,** definované v predchádzajúcich kapitolách,

Atribúty triedy reprezentujú stĺpce asociovanej tabuľky v relačnej databáze. Medzi jednotlivé anotácie atribútov patrí:

- **@Id,** určujúca funkciu primárneho kľúča reprezentovanú anotovaným atribútom, v tomto prípade atribútom **id,**
- **@GeneratedValue(strategy = GenerationType.IDENTITY),** definujúca spôsob generovania primárneho kľúča v danej tabuľke. Druh generácie vyjadrený typom z Enum, IDENTITY, určuje generáciu primárneho kľúča automaticky zo strany databázového systému pomocou ním definovaného stĺpcu identity databáze, reprezentujúceho primárny kľuč danej tabuľky,
- **@Column(name = "name" columnDefinition = "Clob"),** anotácia vyjadrujúca názov stĺpca v danej databázovej tabuľke, rovnaký princíp platí aj pre ostatné atribúty definované v danej triede. columnDefinition = "Clob", explicitne určuje použitie dátového typu text s vysokým rozsahom počtu charakterov v rámci PostgresSql.

Medzi anotácie a atribúty určené na tvorbu vzťahom medzi tabuľkami patrí:

- **@OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL),** vyjadrujúca oneToMany vzťah s typom kolekcie definovanej v rámci kolekcie
- **private List<EntityFile> entityFiles.** Dátový typ **EntityFile** vyjadruje objekt druhej triedy entít v rámci mikroslužby. Jedná sa teda o triedu reprezentujúcu druhú tabuľku relačnej databáze, ktorá reprezentuje prílohy e-mailových správ. Prípad existencie vzťahu OneToMany iba na strane triedy EntityMailMessage reprezentuje unirelačný, jednosmerný vzťah medzi danými tabuľkami. Tabuľka **file** preto obsahuje vlastný primárny kľuč jednotlivých príloh a cudzí kľuč, ktorý je tvorený primárnym kľúčom tabuľky mail, reprezentovanej touto triedou.

Metóda **public void addEntityFile(EntityFile entityFile) ,** slúži na priamu inicializáciu nového ArrayListu v ktorom sú ukladané jednotlivé objekty typu EntityFile, teda druhej triedy entity mikroslužby.

Atribút source je určený na ukladanie zdroja prijatej správy, či sa jedná o prijatie pomocou RestControlleru alebo zo strany nástroja RabbitMQ.

4.2.3.2 **EntityFile**

```
@Data
@Entity
@Table(name = "file")
@NoArgsConstructor
@RequiredArgsConstructor
public class EntityFile {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;
   @NonNull
   @Column(name = "name")
   private String name;
   @NonNull
   @Column(name = "path")
   private String path;
}
```

Anotácie triedy sú obdobné ako u triedy EntityMailMessage, pre podrobný popis viz.kap.3.2.3.1

Atribút **private Long id**, reprezentuje primárny kľuč danej tabuľky s rovnakou stratégiou generovania ako v prípade primárneho kľúča triedy EntityMailMessage.

Atribút **name** reprezentuje názov daného súboru, resp. prílohy e-mailu. Atribút path je určený na uloženie absolútnej cesty k danému súboru. Súbor samotný sa do relačnej databáze neukladá, existuje preto záznam o jeho fyzickom mieste uloženia v rámci úložného priestoru servera na ktorom budem daná mikroslužba nasadená.

Definovanie cudzieho kľúča tabuľky **file** relačnej databáze prebehlo v rámci definície triedy **EntityMailMessage**. Cudzí kľuč je reprezentovaný primárnym kľúčom tabuľky **mail**, reprezentovanej triedou **EntityMailMessage**.

4.2.4 JPA repozitár

Jedná sa o repozitár umožňujúci definovať súbor metód a API určených na prácu s objektmi v rámci ich perzistovania do databáze a získavania záznamov objektov pomocou definovaného API primárne vo formáte JSON.

```
@Repository
public interface MailRepository extends CrudRepository<EntityMailMessage, Long>,
PagingAndSortingRepository<EntityMailMessage, Long> {



    List<EntityMailMessage> findByUserToken(@Param("userToken") String userToken);

    List<EntityMailMessage> findByContent(@Param("content") String content);

    List<EntityMailMessage> findByTimestamp(@Param("timestamp")Timestamp
timestamp);
    List<EntityMailMessage> findBySendTo(@Param("sendTo") String sendTo);

    List<EntityMailMessage> findBySubject(@Param("subject") String subject);}
```

MailRepository je anotované **@Repository**, čo je anotácia explicitne označujúca dané rozhrania ako repozitár pre Spring Application Context.

Rozhranie MailRepository rozširuje rozhrania:

- CrudRepository<EntityMailMessage, Long>, rozhranie s generickým parametrami, kde prvý parameter reprezentuje danú primárnu triedu entít a druhý parameter vyjadruje dátový typ primárneho kľúča danej triedy, v tomto prípade Long. Rozšírenie tohto rozhrania zabezpečí MailRepository schopnosť vytvárať, čítať, updatovať a mazať záznamy v relačnej databáze. Reprezentuje relačné príkazy jazyka SQL INSERT, SELECT, UPDATE a DELETE. Rovnako umožňuje detailnejšiu konfiguráciu vystavovaného API popísanú nižšie,

- PagingAndSortingRepository<EntityMailMessage, Long>, zabezpečujúce možnosti stránkovania a zoraďovania výsledkov volaného API, obsahuje rovnaké generické parametre ako CrudRepository.

MailRepository poskytuje po svojom zavedení ako závislosti v rámci Dependency Injection metódu save, ktorá perzistuje objekt EntityMailMessage do relačnej databáze. Táto funkcionalita je použitá triedou MainService, pre bližší popis viz kap.4.2.6, pre uloženie záznamu o prijatej e-mailovej správe a jej jednotlivých prílohách do databáze.

Jednotlivé deklarované metódy rozhrania definujú samotnú tvorbu API. Napríklad deklarácia metódy ***List<EntityMailMessage> findByUserToken(@Param("userToken") String userToken),*** vytvorí možnosť zavolania parametrizovaného API v tvare api/mails/search/findByUserToken?userToken=john , ktoré následne vráti výsledky všetkých záznamov v databáze ktorých hodnota atribútu userToken sa rovná "john".
Ostatné deklarované metódy fungujú rovnako, pomocou samotného názvu metódy je možné deklarovať tvorbu API, jej zoradenie, prípadné reťazenie viacerých inštrukcií.

HTTP metódy požiadavku sú odvodené z logickej charakteristiky daných výrazov. Napríklad vyššie uvedené metódy sú použiteľné metódou GET, keďže vracajú ako výsledok množinu záznamov z databáze.

Pre tvorbu vlastných dotazov jazyka SQL je možné anotovať danú deklarovanú metódu anotáciou **@Query**, ktorá umožňuje priamu deklarácia vlastného špecifického dotazu špecifikovaným štruktúrovaným jazykom HQL(Hibernate Query Language), ktorý obsahuje minimálne odlišnosti od bežne formalizovaného SQL a zabezpečuje automatickú konverziu daných dotazov do rozličných relačných systémov riadenia báze dát podľa implementovaného databázového ovládača mikroslužby a jeho špecifickej konfigurácie.

Zvolené deklarované metódy boli vytvorené na základe požiadaviek o charakteristike prípadného vyhľadávania dát v rámci databáze.

Pre funkcionalitu JPA repozitára je nutné zaviesť v rámci mikroslužby maven závislosť *spring-boot-starter-data-jpa,* slúžiaci na importovanie potrebných knižníc.

4.2.5 **Implementácia RabbitMQ**

Pre implementáciu RabbitMQ je potrebné nadefinovať host a port danej služby. Táto konfigurácia sa nachádza v application.properties, kde je reprezentovaná vlastnosťami *spring.rabbitmq.host* a *spring.rabbitmq.port*

Zároveň je nutné pridať Maven závislosť *spring-boot-starter-amqp,* čo zabezpečí importovanie potrebných knižníc pre prácu s RabbitMQ.

4.2.5.1 **MailMessageRabbitListener**

Trieda MailMessageRabbitListener implementuje základnú logiku prijímania správ z definovanej fronty nástroja RabbitMQ.

```
@Component
public class MailMessageRabbitListener {
   private MainService mainService;
   @Autowired
   public MailMessageRabbitListener(MainService mainService) {
      this.mainService = mainService;
   }
   @Bean
   public MessageConverter messageConverter() {
      return new Jackson2JsonMessageConverter();
   }

   @RabbitListener(queues = "msmail.queue")
   @SendTo
   public void receiveMailMessage(RabbitMailMessage rabbitMailMessage) {
      mainService.createMailMessageDtoFromRabbitMailMessage(rabbitMailMessage);
   }
}
```

Anotácia **@Component** explicitne oznamuje Spring Application Context, že daná trieda reprezentuje komponent.

Ukazovateľovi objekt MainService, ktorý je zavedený pomocou konštruktoru danej triedy technikou dependency injection, slúži na spracovanie prijatej správy z RabbitMQ a následné uloženie záznamu do databáze a odoslanie danej správy na SMTP server.

Preto aby sme mohli konvertovať prijatú správu vo formáte JSON zo strany fronty RabbitMQ na objekt, potrebujeme nástroj na konvertovanie správ.

```
@Bean
public MessageConverter messageConverter(){
return new Jackson2JsonMessageConverter();}
```

Metóda messageConverter slúži na konfiguráciu špecifického konvertoru správ, ktorý je schopný konvertovať prichádzajúcu správu vo formáte JSON na objekt **RabbitMailMessage** a s ním súvisiace objekty **RabbitMailMessage,** pre bližší popis viz. Kap. 3.2.2.2 a Kap.3.2.2.3. Anotácia **@Bean** deklaruje inštanciu daného konvertora ako bean.

Metóda **receiveMailMessage**, obsahuje anotáciu **@RabbitListener(queues = "msmail.queue")**, táto anotácia aplikuje listener na danú frontu nástroja RabbitMQ. Objektový konvertor Jackson následne konvertuje správu z fronty vo formáte JSON na objektový typ parametru uvedený v metóde **receiveMailMessage**, v tomto prípade sa jedná o **RabbitMailMessage**.

Objekt RabbitMailMessage je následne spracovaný pomocou mainService.

4.2.6 MainService

Trieda MainService slúži na implementáciu základnej "business logiky" mikroslužby. Zabezpečuje spracovanie prijatých správ na požadované objekty, uloženie záznamu o nich do databáze a ich odoslanie na triedu zabezpečujúcu samotné odoslanie e-mailu na SMTP server.

Všeobecná špecifikácia triedy MainService obsahuje:

```
@Service
@Slf4j
public class MainService {

private MailService mailService;

   private MailRepository mailRepository;

   private FileConverterService fileConverterService;



   @Autowired
   public MainService(MailService mailService, MailRepository mailRepository,
FileConverterService fileConverterService) {
      this.mailService = mailService;
      this.mailRepository = mailRepository;
      this.fileConverterService = fileConverterService;
   }                    . . .
```

Anotácia @Service, označuje danú triedu pre Spring kontajner ako službu. Služba tvorí základnú "podnikateľskú" logiku danej aplikácie, v tomto prípade zabezpečenie schopnosti odoslať dané správy na stranu SMTP servera.

Ukazovateľové objekty MailService, FileConverterService a MailRepository sú injektované do MainService v rámci procesu dependency injection. MailRepository je využité na persistovanie entity objektov do databáze. MailService je používané na samotné odoslanie správy v správnom formáte na SMTP server využitým balíčka javax.mail.

FileConverterService je služba zabezpečujúca konverziu súborov z formy ich prijatia v rámci RestControlleru alebo nástroja RabbitMQ do formy vhodnej na perzistovanie do databáze.

Jednotlivé metódy tejto triedy popísané v nasledujúcich kapitolách zabezpečujú spracovávanie špecifických správ.

4.2.6.1 **Spracovanie správy prijatej pomocou Rest Controlleru**

```
public ResponseEntity processMessage(String usertoken, String sendTo, String subject,
String content, List<MultipartFile> multipartFileList) {
        List<java.io.File> files = new ArrayList<>();
EntityMailMessage entityMailMessage = new EntityMailMessage(usertoken, sendTo,
subject, content,
                new           java.sql.Timestamp(Calendar.getInstance().getTime().getTime()),
"Http/RestController");
if (!multipartFileList.isEmpty())
        {
            files = fileConverterService.multipartToFile(multipartFileList);
        }       MailMessageDto messageDto = new MailMessageDto(usertoken, sendTo,
subject, content, files);

entityMailMessage.setEntityFiles(createFilesForDatabase(files));

mailRepository.save(entityMailMessage);
return mailService.sendMessageWithAttachment(messageDto);
}
```

Daná metóda spracúva parametrizovanú správu prijatú zo strany RestControlleru prijatú protokolom HTTP, pre bližší popis kontrolóra viz.kap.4.2.1.

EntityMailMessage reprezentuje objekt triedy entity pre ukladanie záznamu do databáze. Samotné perzistovanie do databáze je vykonané pomocou mailRepository, pre bližší popis viz.kap.3.5.4.

Metóda **createFilesForDatabase(List<java.io.File> filesForSavingToDatabase)** sa nachádza v MainService a zabezpečuje tvorbu kolekcie EntityFile. Samostatná metóda je použitá z dôvodu aplikovania princípu znovupoužiteľnosti, kedže rovnakú logiku v konečnom dôsledku používa aj metóda processRabbitMessage, ktorá spracováva správy prijaté pomocou nástroja RabbitMQ.

FileConverterService je služba určená na konvertovanie dátového typu MultipartFile, určeného na prenos protokolom HTTP s MIME(Multipurpose Internet Mail Extensions) typom multipart/form-data na dátovy typ java.io.File.

MailMessageDto je objekt reprezentujúci danú správu odoslanú na smtp server pomocou služby mailService, bližšie popísanej v kap.

4.2.6.2 **Konfigurácia RabbitMQ**

RabbitMQ musíme nakonfigurovať tak, aby sme zabezpečili korektný proces preposielania e-mailových správ pomocou daného nástroja a aby sme dokázali na strane danej mikroslužby správy z tohto nástroja prijímať.

Konfiguráciou vytvoríme novú exchange, s názvom *msmail.exchange*. Musíme vytvoriť aj novú frontu správ, z ktorej budú správy prijímané v rámci mikroslužby. Názov fronty je *msmail.queue*.

Danú frontu následne previažeme s exchange msmail.exchange. V prípade potreby môžeme kedykoľvek definovať novú frontu, ktorá bude rovnako prijímať správy z msmail.exchange.

Komponentom projektu, ktoré budú mať záujem poslať e-mail vzniká nová možnosť posielať správy aj na danú exchange, z ktorej budú správy smerované na našu mikroslužbu, prípadne môžu byť jednoduchý spôsobom smerované na iné koncové body projektu. Spring AMQP poskytuje možnosť prijímania a odosielania asynchrónnych správ pomocou protokolu AMQP a nástroja RabbitMQ.

Medzi exchange a frontou existuje direct(priama) forma viazania, pre bližší popis viz.kap.2.4.

4.2.6.3 Spracovanie správy prijatej pomocou nástroja RabbitMQ

Implementácia uvedenej funkcionality sa nachádza v triede MainService pomocou metódy **processRabbitMessage**.

```
public void createMailMessageDtoFromRabbitMailMessage(RabbitMailMessage
rabbitMailMessage) {

    MailMessageDto rabbitMessageDto = new MailMessageDto();
    rabbitMessageDto.setUsertoken(rabbitMailMessage.getUserToken());
    rabbitMessageDto.setSendTo(rabbitMailMessage.getSendTo());
    rabbitMessageDto.setSubject(rabbitMailMessage.getSubject());
    rabbitMessageDto.setContent(rabbitMailMessage.getContent());
 rabbitMessageDto.setFiles(fileConverterService.rabbitJsonFileToFile(rabbitMailMessage);
    EntityMailMessage entityMailMessage = new
EntityMailMessage(rabbitMailMessage.getUserToken(), rabbitMailMessage.getSendTo(),
        rabbitMailMessage.getSubject(), rabbitMailMessage.getContent(),
        new java.sql.Timestamp(Calendar.getInstance().getTime().getTime()),
"RabbitMQ");

  entityMailMessage.setEntityFiles(createFilesForDatabase(rabbitMessageDto.getFiles()));

    mailRepository.save(entityMailMessage);
    mailService.sendMessageWithAttachment(rabbitMessageDto); }
```

Na začiatku vytvárame základnú formu dto objektu typu **MailMessageDto**. Služba **fileConverterService** prekonvertuje dané súbory prijaté z nástroja RabbitMQ vo formáte Base64 na štandardizovaný súbor java.io.File v rámci programového jazyka Java. Tieto súbory sú následne v rámci entityMailMessage.setEntityFiles(...) modifikované na typ **EntityFile**, určený na ukladanie záznamov o prílohách do databáze.

Prílohy vo forme java.io.file sú samozrejme odoslané priamo na SMTP server v rámci objektu **rabbitMessageDTO**.

5 Záver

Optimalizácia e-mailového messagingu prebehla úspešne. Mikroslužba bola v požadovanej forme vytvorená a následne akceptovaná zadávateľom prvotnej požiadavky na optimalizáciu.

Vytvorená mikroslužba poskytuje možnosť príjmu HTTP požiadavky na vytvorenie e-mailovej správy pomocou technológie REST. Funkcionalita príjmu správ zo strany nástroja RabbitMQ je taktiež implementovaná a reprezentovaná v mikroslužbe.

Prijaté požiadavky na vytvorenie e-mailovej správy, či už formou HTTP požiadavky alebo správy RabbitMQ sú zaznamenané v relačnej databáze, čím sú vytvorené záznamy o parametroch všetkých prijatých správ a ich prípadných súborových prílohách.

Mikroslužba bola úspešne implementovaná a je v súčasnej podobe používaná na komunikáciu formou e-mailov zo strany určitých mikroslužieb projektu Tieto DevOps Space s pracovníkmi projektu a inými internými zákazníkmi projektu.

Rozsah implementácie a využitia funkcionalít mikroslužby môže byť v budúcnosti potencionálne rozšírení, napríklad o e-mailovú komunikáciu so zákazníkmi po priradení nových výpočtových prostriedkov v rámci Tieto DevOps Space. Ďalším možným použitým je rozšírenie rozosielania e-mailových správ medzi pracovníkmi projektu, prípadne ďalšími internými zákazníkmi projektu s rozšíreným využitím asynchrónneho spôsobu preposielania e-mailových správ.

Zoznam použitej literatúry

1. BLOCH, Joshua. Effective Java. Third edition. Boston: Addison-Wesley, 2018. ISBN 0134685997.

2. CARNELL, John. Spring Microservices in Action. Shelter, Island, NY: Manning Publications Co., 2017. ISBN 9781617293986.

3. SCHILDT, Herbert. Java 8: výukový kurs. Přeložil Jakub GONER. Brno: Computer Press, 2016. ISBN 9788025146651.

4. WALLS, Craig. Spring in Action, Fifth Edition. Shelter, Island, NY: Manning Publications Co., 2018. ISBN 9781617294945.

Internetové zdroje

5. Baeldung. Apache Maven Standard Directory Layout [online]. Posledná aktualizácia 22.10.2018 [cit.10.4.2019]. Dostupné z: https://www.baeldung.com/maven-directory-structure

Zoznam skratiek

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
DI	Dependency Injection
FIFO	First In, First Out
HQL	Hibernate Query Language
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JDBC	Java Database Connectivity
JDK	Java Development Kit
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MIME	Multipurpose Internet Mail Extensions
ORM	Object - relational mapping
POM	Project Object Model

REST	Representational state transfer
URI	Uniform Resource Identifier
WAR	Web Application Archive
XML	Extensible Markup Language

Prehlásenie o využití výsledkov bakalárskej práce

Prehlasujem, že

- som bol zoznámený s tým, že na moju bakalársku prácu sa plno vzťahuje

zákon č. 121/2000 Sb. – autorský zákon, najmä § 35 – užití díla v rámci občanských

a náboženských obřadů, v rámci školních představení a užití díla školního a § 60 – školní dílo;

- beriem na vedomie, že Vysoká škola báňská – Technická univerzita Ostrava (ďalej iba VŠB-TUO) má právo nezárobkovo, ku svojej vnútornej potrebe, bakalársku prácu použiť (§ 35 odst.3);

- súhlasím s tým, že bakalárska práca bude v elektronickej podobe archivovaná v Ústrednej knižnici VŠB-TUO. Súhlasím s tým, že bibliografické údaje o bakalárskej práci budú zverejnené v informačnom systéme VŠB-TUO;

- bolo zjednané, že s VŠB-TUO, v prípade záujmu z jej strany, uzavriem licenčnú zmluvu

s oprávnením použiť dielo v rozsahu § 12 odst. 4 autorského zákona;

- bolo zjednané, že použiť svoje dielo, bakalársku prácu, alebo poskytnúť licenciu

k jej využitiu môžem iba so súhlasom VŠB-TUO, ktorá je oprávnená v takom prípade odo

mňa požadovať primeraný príspevok na úhradu nákladov, ktoré boli VŠB-TUO na vytvorenie diela vynaložené (až do ich skutočnej výšky).

V Ostrave dňa 6.5.2019

....

Patrik Polaček

Zoznam príloh

Príloha č. 1: Priložené CD

Príloha č. 1: Priložené CD s programom

Príloha s CD obsahuje:

- Digitálnu verziu bakalárskej práce,
- Zdrojový kód programu.