LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

MIKROSERVISU ARHITEKTŪRAS REALIZĀCIJAS GRŪTĪBU RISINĀŠANA

MAĢISTRA DARBS

Autors: **Daņiils Lučanskis**
Stud. apl. Nr. dl09006
Darba vadītājs: Dr. dat. Maksims Kravcevs

RĪGA 2016

ANOTĀCIJA

Pastāvīgas programmatūras sarežģītības paaugstināšanas, uz veiktspēju virzītas jaunu prasību radīšanas un nepārtrauktās funkcionalitātes izmaiņu veikšanas nepieciešamības dēļ paradās vajadzība projektēt sistēmas, kuras ir iespējams bez lieliem ieguldījumiem mērogot un atbalstīt ilgo laiku, programmkoda izmēra palielināšanas gaitā būtiski nemainot izmaiņu veikšanai vajadzīgas izmaksas.

Viena no klientservera programmatūras veidošanas pieejām ir mikroservisu arhitektūra, kas bija rādījušies nesen, sarežģīto pasaules mēroga tīmeklī bāzēto sistēmu evolūcijas procesā, un piedāvā pieeju visu šo vajadzību apmierināšanai.

Maģistra darba ietvaros ir dots priekšstats par mikroservisu arhitektūras būtību salīdzinājumā ar citām plaši izmantojamām arhitektūrām, definēti šķērsli, kas radās, programmatūras izstrādē izvēloties pielietot mikroservisu arhitektūru, un piedāvāta katra šķēršļa novēršanas stratēģija, kā arī tās pielietošanas apraksts uz darba ietvaros izstrādātas parauga sistēmas pamata.

Atslēgvārdi

Mikroservisu arhitektūra, Docker, Kubernetes, DevOps

ABSTRACT

Theme: Solving microservice architecture development challenges

With permanent increase of software complexity, new performance related requirement creation, necessity to perform continuous functionality changes, appears the need to design systems that are able to scale without significant efforts and to be maintained for a long time, without cost increase during the course of software growth.

One of client/server software development approaches is microservice architecture, which appeared recently, in the process of large worldwide web-based system evolution, and that is able to address all these needs.

Within master thesis the nature of microservice architecture is explained based on comparison with other widely used architectures, obstacles are defined that arise when it is decided to use microservice architecture in software product development, and for each obstacle its elimination strategy is offered, as well as description of its implementation within developed sample system.

Keywords

Microservice architecture, Docker, Kubernetes, DevOps

AUTOREFERĀTS

Mikroservisu arhitektūra ir samērā jauns, nav pamatīgi izpētīts jēdziens un arī latviešu valodā par šo tēmu materiāli vēl nav pieejami.

Autors tika izpētījis lielāko daļu par šo tematu pieejamās literatūras, kas ir redzams izmantojamo avotu sarakstā. No avotiem lielākā daļa ir nerecenzēti tīmekļa avoti, jo tēmas novitātes dēļ publicēto grāmatu un zinātnisku rakstu skaitlis ir ierobežots.

Darba pirmā daļa tika ieguldīta avotu izpētei un svarīgāko jēdziena principu formulēšanai. Otrajā daļā tiek veikts pētījums, kura ietvaros tiek praktiski pielietota aprakstītā programmatūras veidošanas pieeja, un, balstoties uz iegūto pieredzi un zināšanām, tiek definēti mikroservisu arhitektūras pielietošanas šķēršļi, aprakstīta to risināšanas stratēģija un katra šķēršļa novēršanas piemērs.

Darba rezultāti piedāvā lasītājam pamatu piemērotākai sistēmas arhitektūras izvēlei un sistēmas izstrādei izmantojot mikroservisu arhitektūru, balsoties uz darba ietvaros iegūtas un aprakstītas pieredzes.

Darbs tika noformēts izmantojot "Maģistra darba izstrādes un aizstāvēšanas metodiskie norādījumi" dokumentā specificētas vadlīnijas un gramatiskas kļūdas tika pārbaudītas un salabotas ar Tildes Biroja Microsoft Word iebūvēta kļūdu pārbaudītāja palīdzību.

Pie visiem terminu un ideju formulējumiem, kas ir ievietoti darbā, ir pievienotas atsauces ar avotu. Tieši tulkojumi ir attiecīgi noformēti.

SATURS

APZĪMĒJUMU SARAKSTS

API (Application Programming Interface) – funkciju (klašu, struktūru, konstantu) kopums, kuru lietotne vai atsevišķā lietotnes daļa piedāvā klientiem pieprasījumu veikšanai.

BFF (Backends for Frontends) – sistēmu projektēšanas šablons, kurā katram klienta tipam tiek izstrādāts savs API, nevis tiek izmantots viens unificēts API visiem klientu tipiem.

DevOps (Development & Operations) – programmatūras izstrādes metodoloģija, kas pamatojas uz ciešo izstrādātāju un citu informācijas tehnoloģiju speciālistu sadarbību, automatizējot programmatūras piegādes un infrastruktūras izmaiņu procesus.

CI (Continuous Integration) – programmatūras izstrādes prakse, kurā tiek veikta bieža automātiska programmatūras palaišana sistēmas komponentu integrācijas un citu problēmu agrākai atklāšanai un labošanai.

CD (Continuous Delivery) – programmatūras izstrādes prakse, kurā komandas ražo jaunās programmatūru versijas īsos laika intervālos, nodrošinot programmatūras izvietošanas iespēju ražošanā jebkurā laikā.

DDD (Domain-Driven Design) – programmatūras projektēšanas principu kopums, kurā programmatūras vienības tieši attēlo darījumvienības un darījumprocesus vieglākai sapratnei starp ieinteresētām personām.

ESB (Enterprise Service Bus) – savienojošā programmatūra, kas nodrošina centralizēto un unificēto ziņojumu apmaiņu starp vairākām sistēmām vai sistēmas servisiem.

IaaS (Infrastructure as a Service) – mākonī pieejamā infrastruktūra, parasti attālināti izvietoti serveri vai virtuālās mašīnas, kurus atbalsta cita kompānija.

JMS – ziņojumu apmaiņas standarts Java valodā izstrādātai programmatūrai.

IPC (Inter-Process Communication) – starpprocesu tiešās saziņas mehānisms.

JSON (JavaScript Object Notation) – tīmeklī bāzētās sistēmās populārs tekstuāls datu apmaiņas formāts.

PaaS (Platform as a Service) – mākonī pieejams daudzveidīgo programmatūras izvietošanai un palaišanai vajadzīgo sistēmu kopums, kuru atbalsta cita kompānija, un kuru iespējams izmantot, ielādējot programmatūru tās kompānijas serveros.

REST (Representational State Transfer) – komunikācijas pieeja starp tīmekļa lietojumprogrammām vai lietojumprogrammas komponentēm.

RPC (Remote Procedure Call) – starpprocesu komunikācijas tehnoloģija, kas ir pieejama konkrētajās valodās un ļauj datorprogrammai ērti izsaukt procedūru, kura ir pieejama citā adrešu telpā (parasti uz cita datora ar koplietojamu tīklu).

SOA (Service-Oriented Architecture) – pakalpojumu (servisu) orientētā arhitektūra.

SOAP (Simple Object Access Protocol) – strukturēto ziņojumu apmaiņas protokols sadalītajā vidē.

WSDL (Web Services Description Language) – tīmekļa servisu pieejamas funkcionalitātes un pie tās piekļuves iespēju apraksta valoda.

IEVADS

Modernās programmatūras pastāvīgā sarežģītības palielināšana un jauno prasību radīšana prasa jauno pieeju izgudrošanu sistēmu arhitektūras projektēšanai. Tipiskā tīmeklī bāzētā sistēma pašlaik sastāv no vairākiem komponentiem – tīmeklī pieejamas daļas, autorizācijas mehānisma, uzdevumu rindas un to apstrādājošiem servisiem, dažādām datu bāzēm, notikumu reģistrēšanas sistēmas, analītikas sistēmas un citām daļām, kas rāda nepieciešamību to efektīvi pārvaldīt, lai projekts nepārvērstos par pilnībā neuzturamo produktu.

Kā alternatīva eksistējošām klientservera programmatūras arhitektūrām parādījās un ātri guva popularitāti mikroservisu arhitektūra, kā jaunā paradigma, kurā sistēma ir komponēta no vairākiem maziem servisiem, kas tiek neatkarīgi darbināti katrs savā procesā, patstāvīgi mērogoti un atjaunināti, un komunicējas, izmantojot vienkāršus saziņas mehānismus.

Mikroservisu popularitātes galvenais cēlonis ir organizāciju vēlme: cik ātri vien iespējams piegādāt lietotājiem jauno funkcionalitāti, lai palielinātu savu konkurent spēju tirgū; efektīvi mērogot programmatūru; uzlabot programmatūras piecietību pret kļūdām; un palielināt izstrādātāju produktivitāti, pazeminot programmatūras koda sarežģītību.

Tādas pēdējā laika tendences, ka spējās izstrādes metodes, programmatūras piegādes automatizācijas kultūra (DevOps), mākoņu infrastruktūras (PaaS, IaaS) daudzveidīgo risinājumu rašanās, virtualizācijas iespējas izmantojot vieglus konteinerus, programmatūras piegādes un testēšanas biežuma palielinājums (CI, CD), deva pamatu un iespēju realizēt modularizētas lielā mēroga sistēmas.

Lai gan mikroservisu arhitektūra jau tika realizēta vairākos pazīstamos tīmekļa servisos kā Amazon, Google, Netflix, LinkedIn un eBay [1, 2, 3, 4], pastāv vairāki šķēršļi, kurus organizācijām un izstrādātājiem jāņem vērā, apsverot mikroservisu pielietošanu savos projektos. Salīdzinājumā ar pašlaik visizplatītāko monolīta arhitektūru radās problēmas sakarā ar sadalīto izpildi, infrastruktūras atbalstu, izmaiņu vadību.

Jāņem vērā, kā mikroservisu arhitektūra netiek piemērota saskarnes izstrādei, jo vairākas galvenās šī pieejas priekšrocības, tādas, ka ērtākas mērogošanas un izvietošanas iespējas, netiek pielietotas izpildot programmkodu klienta daļā.

Darba mērķis ir izpētīt un salīdzināt modernās lietojumprogrammu arhitektūras veidus; dot paplašinātu priekšstatu par mikroservisu arhitektūru; implementēt sistēmu ar mikroservisu arhitektūras pielietojumu; balstoties uz pieejamo literatūru, pētījumiem un iegūto pieredzi, formulēt problēmas, kas radās mikroservisu arhitektūras pielietošanas procesā; aprakstīt katras problēmas risināšanas stratēģiju un tās realizācijas variantu uz darba ietvaros izstrādātas sistēmas pamata.

Pirmajā darba nodaļā tiek aprakstīti mūsdienā pielietojamie programmatūras arhitektūras veidi un tiek padziļināti apskatīta mikroservisu arhitektūra, definēti katras arhitektūras pielietošanas priekšrocības un trūkumi. Otrajā nodaļā tiek definēti šķērsli, ar kuriem sastapās izstrādātāji, veicot sistēmas realizēšanu, izmantojot mikroservisus, un tiek aprakstīta katra šķēršļa risināšanas pieeja.

1.1. MŪSDIENU ARHITEKTŪRAS

1.1. Ievads

Viens no kritiskajiem jautājumiem jebkuras sarežģītas programmatūras realizācijā ir tās arhitektūras projektēšana. Darbā izmantotais programmatūras arhitektūras jēdziens ir paņemts no IEEE 1774 standarta [5]:

Arhitektūra ir fundamentālā sistēmas komponentu, attiecību starp tiem un vidi, projektēšanu un attīstību vadošo pamatprincipu organizācija.

Arhitektūra ir svarīga, jo tā ietekmē [6]:

1. Komunikāciju starp ieinteresētām personām. Arhitektūra attēlo vispārējo sistēmas abstrakciju, kuru ieinteresētās personas var izmantot par pamatu savstarpējai sapratnei un pārrunām.
2. Agrākus programmatūras projektēšanas lēmumus, kuri savukārt ievērojami ietekmē lēmumus tālākajā izstrādē, izvietošanā un uzturēšanā. Arhitektūras maiņa pēc programmatūras izstrādes sākuma kļūst arvien sarežģītāka.
3. Citas sistēmas, jo esot atkal izmantojamā sistēmas abstrakcija. Arhitektūra ir salīdzinoši neliels, sapratnei viegls modelis, kas attēlo, kā sistēma ir strukturēta un kā tās elementi strādā kopā, tāpēc šīs modelis var būt atkal izmantots turpmākās sistēmās, kuriem ir līdzīgas prasības.

Katram apskatītam arhitektūras veidam nav noteiktas definīcijas, tāpēc darbā visi arhitektūru apraksti ir galveno iezīmju kompilācijas no vairākiem resursiem.

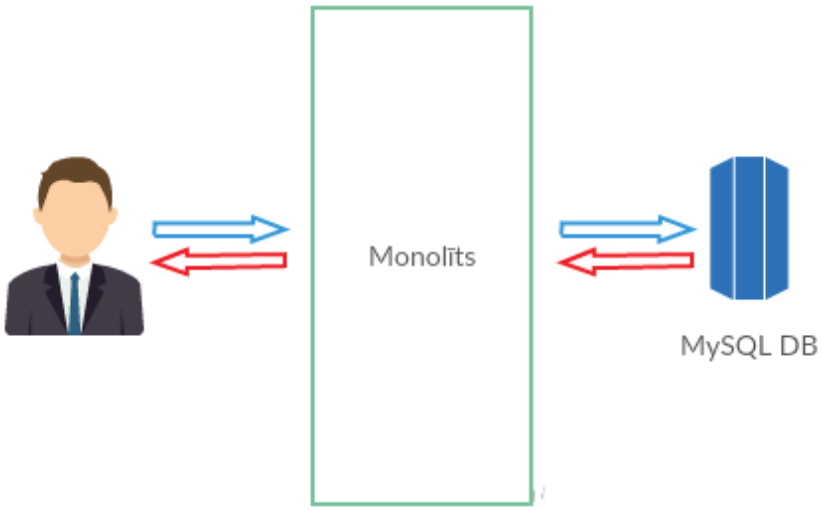
Pašlaik visizplatītākā ir slāņu arhitektūra, kas dod iespēju loģiski (moduļos) vai fiziski (operētājsistēmas procesos) dalīt programmatūru vairākos slāņos, bet būtiski nepalielina infrastruktūras administrēšanas sarežģītību un ir pilnībā piemērota mazā un vidējā apjoma projektiem. Tādu programmatūras dalīšana ir iespējams nosaukt par horizontālu – katram slānim ir atbildība par kādu darbību no sistēmas realizācijas tehniskā viedokļa – piemēram, attēlot datus lietotājam vai saglabāt datus glabātuvē.

Programmatūras izmēram palielinoties, jaunai funkcionalitātei radoties, kad komandā sāk strādāt desmitiem cilvēku, sistēmas uzturēšana kļūst arvien sarežģītāka, jo slāņu skaits būtiski nemainās, tāpēc katrs slānis pakāpeniski izaug. Šajā situācijā piemērotāka kļūst mikroservisu arhitektūra, kurā programmatūras kods tiek sadalīts vertikāli – katrs mikroserviss ir pilnībā atbildīgs par kādu funkcionalitātes daļu, piemēram, par lietotāju datu glabāšanu, attēlošanu un citu saistīto pārvaldi – kas dod iespēju katru funkcionālo vienību izstrādāt izolēti no citiem, tāpēc jaunās funkcionalitātes izstrādes gaitā pati programmkoda sarežģītība nemainās, tomēr radās vairāki jautājumi infrastruktūras līmenī.

1.2. Monolīta arhitektūra

1.2.1. Apraksts

Monolītā arhitektūrā gandrīz visi sistēmas komponenti tiek palaisti kā viena nedalāmā vienība – vienā operētājsistēmas procesā. Sistēmas komponenti komunicējas ar koplietojamas atmiņas palīdzību, tāpēc pieprasījumi starp komponentiem tiek izpildīti sinhroni. Viss pirmkods atrodas vienā vietā un tiek izmantots viens tehnoloģiju kopums, tāpēc izstrāde kopumā ir vienkārša, izstrādātāju programmēšanas un administrēšanas līmenim nav jābūt augstam.



1.1. att. **Monolīta arhitektūras piemērs**

1.2.2. Priekšrocības

Priekšrocību apraksts tika izveidots uz [16, 17] rakstu pamata:

- Viegla projekta izstrādes uzsākšana, jo jāpalaiž tikai vienu komponentu;
- Viegls izstrādes process, kamēr sistēmai ir mazs izmērs;
- Ļoti viegla izvietošana, jo būtībā izvietošanai ir vajadzīgs nokopēt vajadzīgā mašīnā vienu palaižamo vienību;
- Sistēmas mērogošana notiek, palaižot vairākas sistēmas instances un novirzot lietotājus uz instanci izmantojot slodzes balansētāju;
- Vieglā testēšana un atkļūdošana, jo jādarbina tikai vienu palaižamo vienību;
- Programmatūra ir samērā ātra, ja nav pārslogota - tiešie izsaukumi viena procesa ietvaros ir ātrāki nekā starpprocesu komunikācija;
- Vieglā un lētā sistēmas projektēšana.

1.2.3. Trūkumi

Trūkumu apraksts tika izveidots uz [16, 17] rakstu pamata:

- Nav iespējas mērogot atsevišķas sistēmas komponentes, tāpēc mērogošana notiek neefektīvi – ja lēni strādā viena sistēmas komponente, tiek ietekmēta visa sistēma un ātrdarbības palielināšanai ir jāpalaiž vēl vienu visas sistēmas instanci;
- Nav iespējams konkrētai sistēmas komponentei pielāgot savu palaišanas vidi;
- Sarežģītāka un lēnāka sistēmas saprašana un izmaiņu veikšana sistēmas izmēru palielināšanas gaitā;
- Izstrādes vides tiek pārslogotas sistēmas izmēra palielināšanas gadījumā;
- Ļoti sarežģīti mainīt izvēlētas tehnoloģijas (programmēšanas valodu, datu bāzes dzinēju, ietvaru) pēc produkta izstrādes uzsākšanas, jo būtībā ir jāpārstrādā un pēc tam jānotestē visu sistēmas funkcionalitāti;
- Palielinoties sistēmas izmēram, jauno izstrādātāju pievienošana sistēmai kļūst sarežģītāka, jo efektīvam darbam katram izstrādātājam jāzina visu sistēmu;
- Sistēmas atsevišķu komponentu atjaunināšana pieprasa visas sistēmas pārlādēšanu;
- Izstrādātājiem jākoordinē izmaiņu veikšanu un sistēmas izvietošanu, kas padara nepārtraukto produkta piegādi (CD) sarežģītāku;

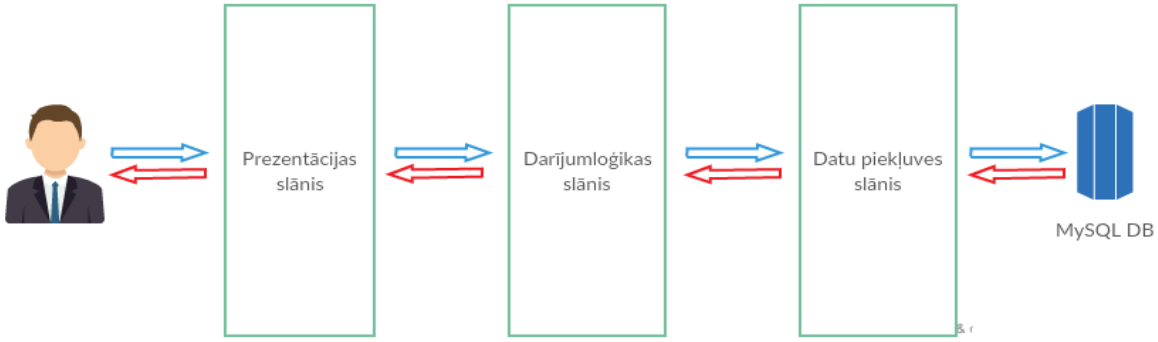
- Testēšana pirms sistēmas piegādes klientam aizņem daudz laika, jo pirms katras piegādes ideāli ir jātestē visu sistēmu, kas arī padara nepārtraukto produkta piegādi (CD) sarežģītāku;
- Automatizēto testa gadījumu palaišana notiek ilgo laiku lielā izmēra sistēmas gadījumā;
- Ja programmatūras daļa sabrūk, tad sabrūk visa sistēmas instance;
- Sistēmas palaišana un pārlādēšana aizņem laiku – jo lielāka kļūst sistēma, jo ilgāks ir laiks;
- Grūti piedalīties izstrādē vairākām izstrādātāju komandām – jo vairāk izstrādātāju, jo iespējamāka ir darba apgabalu pārklāšanas un tajā sakarā starp izstrādātājiem paradās konflikti;
- Sistēmas izstrādes gaitā tajā radās strukturēšanas problēmas, jo sistēmas komponenti nav stingri sadalīti un ar laiku arvien vairāk savienojās.

1.3. Slāņu arhitektūra

1.3.1. Apraksts

Slāņu arhitektūra ir modificēta monolīta arhitektūra, kurā pastāv vairāki slāņi, kas var būt izvietoti atsevišķos operētājsistēmas procesos vai serveros (šajā gadījumā tā ir saukta par vairākrindu arhitektūru, *n-tier architecture*), var būt izstrādāti izmantojot atšķirīgas valodas un komunikācija starp tiem notiek pakāpeniski no augšējā slāņa, ar kuru sazinās lietotājs, uz apakšējo, kas galvenokārt ir datu glabātuve. To var nosaukt par zelta vidusceļu, kad jau paradās sistēma dalīšana moduļos, bet vēl nav jātērē ievērojamus resursus sistēmas infrastruktūras atbalstam. Tipiskie slāņi ir:

1. augšējais prezentācijas slānis, kas nodarbojas ar klienta pieprasījumu sūtīšanu loģikas slānī un loģikas slāņa rezultātu attēlošanu klientam;
2. vidējais loģikas slānis, kas nodarbojas ar prezentācijas slāņa pieprasījumu apstrādi, sazināšanos ar datu piekļuves slāni, un rezultātu sūtīšanu prezentācijas slānim;
3. apakšējais datu piekļuves slānis, kas nodarbojas ar pieprasījumiem datu bāzē.



1.2. att. **Slāņu arhitektūras piemērs**

1.3.2. Priekšrocības

- Vienkārša, saprotama sistēmas struktūra;
- Vienkārša palaišana, jo komponentu skaits ir ierobežots un sistēmas izstrādes gaitā būtiski nemainās;
- Vienkārša mērogošana ierobežota komponentu skaita dēļ un, salīdzinājumā ar monolīta arhitektūru, ir iespējams efektīvāk mērogot atsevišķus slāņus;
- Salīdzinājumā ar monolīta arhitektūru, radās izstrādātāju darbības sfēras sadalījums, izmantojot līgumus starp slāņiem, kas dod iespēju izstrādātājiem nedomāt par citu slāņu realizācijas detaļām;
- Vieglāk nekā monolīta arhitektūrā samainīt tehnoloģisku risinājumu.

1.3.3. Trūkumi

- Slāņi joprojām ir lieli, izstrādātājiem jāņem vērā un jādomā par lielo sistēmas daļu;
- Slāņu izmēram palielinoties, tam kļūst piemēroti visi monolīta arhitektūras trūkumi;
- Mērogošana joprojām ir ierobežota, jo parasti ir vajadzība mērogot konkrēto funkcionalitāti, nevis, piemēram, visu datu bāzes pārvaldības slāni;
- Izstrādātāji galvenokārt strādā vienā līmenī, kompetence citos līmeņos neaug vai samazinās, tāpēc var parādīties konflikti starp vienas funkcionalitātes vairāku slāņu izstrādātājiem;
- Komunikācija starp komandas dalībniekiem notiek pa slāņiem, kas samazina izmaiņu veikšanas ātrumu, jo gandrīz katrai izmaiņu veikšanai ir vajadzīgas izmaiņas katrā slānī.

1.4. Servisu orientēta arhitektūra

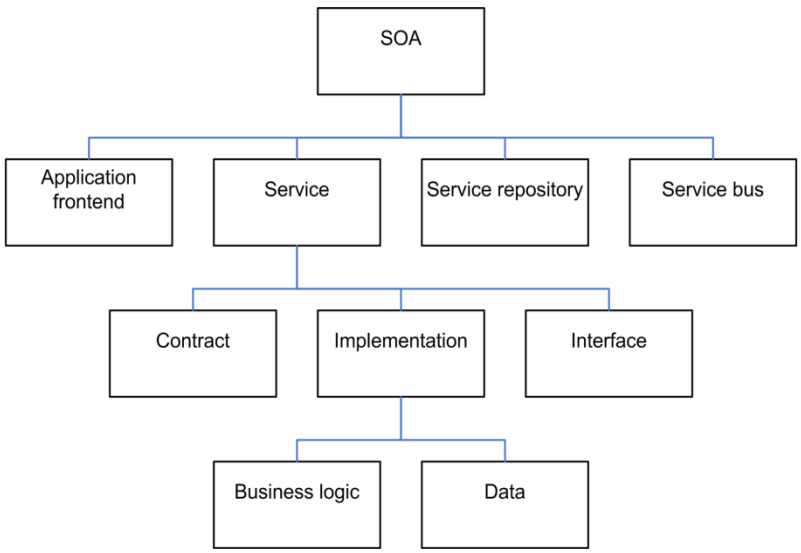
1.4.1. Apraksts

Servisu (Pakalpojumu) Orientēta Arhitektūra (SOA) ir plašs termins, kas būtībā ir nākamais solis programmatūras dalīšanā mazākās daļās, izmantojot uz problēmas apgabalu virzīto projektēšanu (DDD), dalot sistēmu vertikāli pa darījumprocesiem. Viena no SOA definīcijām [11]:

SOA ir arhitektūras stils, kura mērķis ir sasniegt vāji sasieto savienojumu starp mijiedarbotiem programmatūras aģentiem. Serviss ir darba vienība, kuru izpilda servisa nodrošinātājs lai sasniegtu servisa patērētāja vēlamo rezultātu.

Būtiskie elementi, no kuriem sastāv pakalpojumu orientētā sistēma [19]:

- servisi – neatkarīgi izpildāmās vienības ar noteikto līgumu (darbību kopumu);
- servisu direktorija, kurā glabājas servisi un to palaišanas detaļas;
- lietojumprogrammas saskarnes, no kurās notiek visu sistēmas aktivitāšu iniciēšana un kontrole (piemēram, tīmekļa vietne, kā arī publiski pieejams API),
- servisu starpnieks, ar kura palīdzību notiek saziņa starp servisiem.



1.3. att. **Servisu orientētās arhitektūras sastāvdaļas**

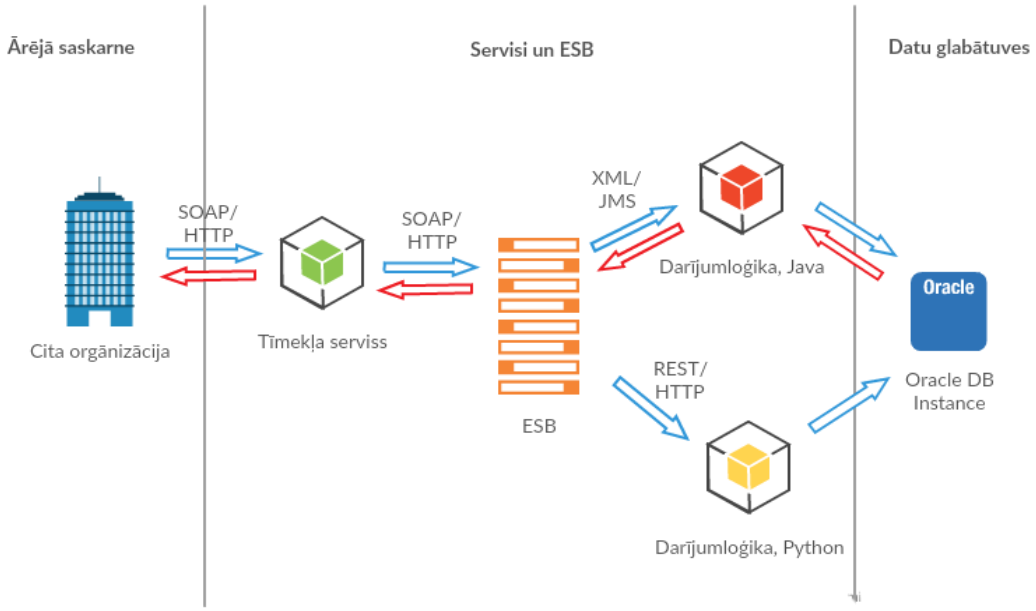
SOA tiek galvenokārt lietota korporatīvo lietotņu pasaulē, jo tās ieviešana aizņem ievērojamus resursus. Starpservisu komunikācijai un servisu direktorijam parasti tiek

izmantots salikts risinājums – uzņēmuma pakalpojumu starpnieks (ESB), kas nodrošina centralizēto un unificēto ziņojumu apmaiņu starp vairākiem servisiem vai sistēmām, glabā servisu atrašanas adreses, veic transakciju un servisu dzīves cikla kontroli un datu konversiju.

Servisu izmērs un būtība arī atbilst darījumprocesiem – ir uzskatīts, kā serviss implementē atsevišķo uzņēmuma darījumpakalpojumu, un servisu kopumi nodrošina darījumprocesu izpildi.

Arī pēc mēroga SOA atšķīrās no citām aprakstītām arhitektūrām – ja citās arhitektūrās ir aprakstīta vienas sistēmas struktūra, SOA strukturē visu organizācijas servisu, kā arī ārējo servisu, mijiedarbību. SOA definē vairākus servisu veidošanas principus [12]:

- Standartizēti servisu līgumi. Sazināšanai starp servisiem jābūt standartizētai, piedāvājot iespēju servisa patērētājam saņemt informāciju par servisa piegādātājā izmantošanas iespējām (piemēram, WSDL un SOAP standarti).
- Jāveido servisus ar atbilstošo granularitāti. Granularitāti jāizvēlas katram servisam atsevišķi, lai tas varētu piedāvāt vislabāko elastīgumu priekš servisa patērētājam, neietekmējot veiktspēju un drošību
- Servisiem jābūt vāji savienotiem. Servisus jāizmanto tikai caur definētiem publiski pieejamiem līgumiem.
- Servisiem jābūt abstraktiem. Servisa patērētājiem nav jāzina tehniskās servisa realizācijas detaļas.
- Servisiem jābūt komponējamiem. Jābūt iespējai veidot saliktus servisus.
- Servisiem nav jābūt stāvokļa. Katrai servisu izsaukšanai jābūt neatkarīgai, padodot visu nepieciešamo darbības izpildei informāciju katrā pieprasījumā.
- Servisiem jābūt neatkarīgiem un jākontrolē savu izpildes vidi.
- Servisiem jābūt atkalizmantojamiem. Piemēram, atskaišu veidošanas serviss varētu būt izmantojams vairākos darījumprocesos.



1.4. att. **Servisu orientētās arhitektūras piemērs**

1.4.2. Priekšrocības

- Salīdzinājuma ar monolīta arhitektūru, nav vajadzīgs pārlādēt visu sistēmu vienas funkcionālās vienības atjaunināšanas gadījumā – pietiek ar konkrēta servisa pārlādēšanu;
- Labākā sistēmas dalīšana moduļos, servisi kļūst vājāk savienoti, tāpēc vieglāk veikt izmaiņas;
- Standartizēta komunikācija starp servisiem;
- Efektīvāka mērogošana, jo sistēma ir vairāk sadalīta tieši funkcionālajos apgabalos;
- Ciešāka saskaņotība starp IT un darījumprocesiem, jo servisi attēlo darījumprocesu struktūru.

1.4.3. Trūkumi

- Radās atbalsta sarežģītība infrastruktūras līmenī;
- Pielāgota uzņēmuma sistēmām, jo ieviešanai ir vajadzīgi ievērojami resursi un noteiktā izstrādātāju kompetence;
- Servisa lielums nav definēts un būtībā serviss var būt tikpat liels kā monolīta lietojumprogrammas, ar tiem piemērotiem trūkumiem.

1.5. Mikroservisu arhitektūra

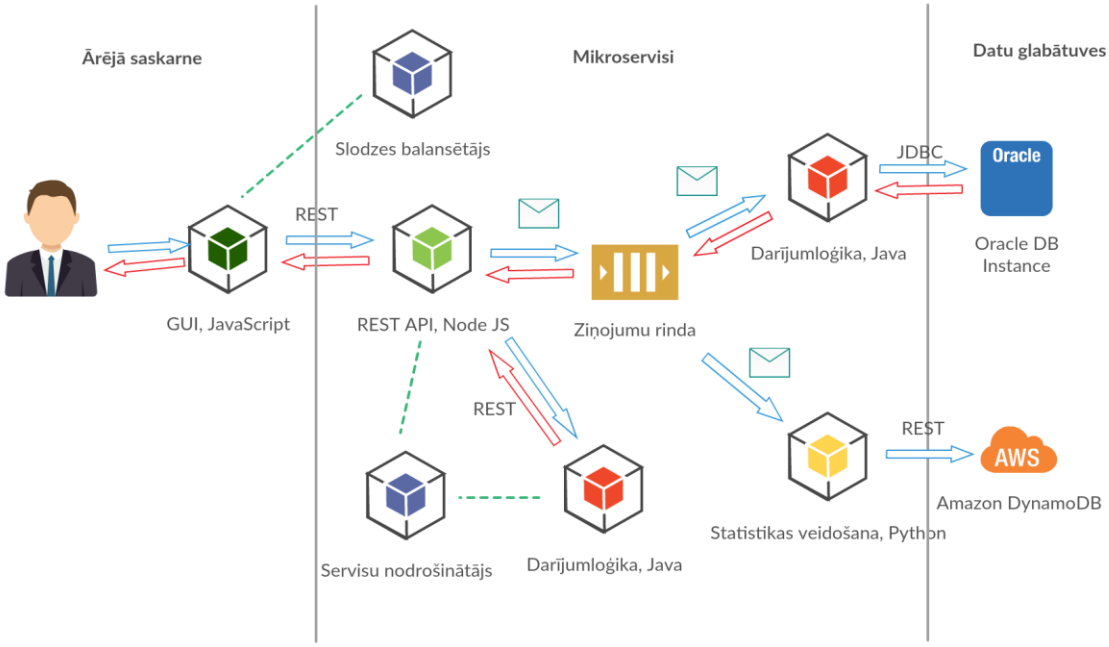
1.5.1. Apraksts

Mikroservisu arhitektūra ir SOA paveids ar atšķirību, kā mikroservisi ir mazāki nekā SOA, ziņojumu apmaiņai tiek izmantoti vieglie apmaiņas protokoli kā REST nevis SOAP, tiešie asinhronie ziņojumu sūtīšanas mehānismi vai RPC, bez starpnieka kā ESB. Ziņojumu apmaiņai vajadzīgā loģika tiek izvietota pašos mikroservisos.

Būtiskāka mikroservisu arhitektūras atšķirība no monolīta arhitektūras ir tas, kā programmas sarežģītība ir pārvietota no sistēmas koda uz infrastruktūras līmeni, un veiksmīgs infrastruktūras līmeņa ierīkojums ir atslēga priekš arhitektūras pielietošanas veiksmes. Tā kā tagad izstrādātājiem kļūst pieejami vairāki un vairāki infrastruktūras administrēšanas rīki, infrastruktūras uzstādīšana un sistēmas komponentu palaišana kļūst vieglāka.

Mikroserviss ir pilnībā izolēta izpildāmā vienība (process) ar vienu skaidri saprotamo darbību. Tas dod iespējas mikroservisiem būt uzrakstītiem atsevišķās valodās, izmantot dažādus ietvarus un datu glabātuves. Tie var būt neatkarīgi palaisti, pārtraukti un mainīti.

Nav tieši definēts, cik lielam jābūt mikroservisam, bet galvenokārt pastāv vienošanās, kā mikroservisa darbības jēgu ir jāspēj pilnībā izteikt vienā teikumā.



1.5. att. **Mikroservisu arhitektūras piemērs**

1.5.2. Priekšrocības

Mikroservisu priekšrocību saraksts tika izveidots uz [13] grāmatas un [18, 20, 21] rakstu pamata:

- Neatkarīgā, efektīvā mērogošana, jo katrs mikroserviss var mērogoties atsevišķi un iespējams palaist servisus piemērotākajā aparatūrā;
- Neatkarīgā komponentu atjaunošana, jo, atjaunojot mikroservisu, citus pārlādēt nav nepieciešams;
- Iespēja nepārtraukti piegādāt produktu, jo nav vajadzīgs koordinēties ar citiem komandas biedriem un komandām par viena mikroservisa pārlādēšanu, ja netiek mainīts API;
- Vienkārša izmaiņu veikšana, jo darba konteksts ir ierobežots;
- Izstrādes vides netiek pārslogotas, jo ir iespējams katru mikroservisu izstrādāt atsevišķi un to izmērs ir mazs;
- Iespēja izmantot vairākas valodas un datu bāzes, viegli mainīt tos viena mikroservisa ietvaros, jo katram mikroservisam tiek izmantota sava palaišanas vide un no citiem mikroservisiem izolēta datu bāze;
- Kļūdu izolācija, jo kļūda ideālajā gadījumā ietekmēs tikai pašu mikroservisu;
- Izstrādātājiem nav specializācijas uz vienu izstrādes procesu;
- Ātrāks viena mikroservisa palaišanas process;
- Viegla moduļu testēšana;
- Iespējams pievienot vairākās izstrādātāju komandas sistēmas izstrādei, jo tie strādās neatkarīgos koda gabalos;
- Programmkoda apjoma palielināšana nepalielina koda sarežģītību – katrs mikroserviss paliek mazs.

1.5.3. Trūkumi

Mikroservisu trūkumu saraksts tika izveidots uz [18] raksta pamata:

- Sarežģītāka kopējā sistēmas testēšana;
- Jārealizē un jāatbalsta komunikācijas mehānismus starp mikroservisiem;
- Salīdzinot ar monolīta un citām ciešāk savienotiem arhitektūrām, komunikācijas mehānismi starp mikroservisiem ir lēnāki par pieprasījumu veikšanu vienā procesā;

- Sarežģīti realizēt sadalītās transakcijas;
- Izstrādātāju profesionālisma līmenim jābūt augstam;
- Jāprojektē mikroservisus tā, lai tie būtu toleranti pret citu servisu kļūdām;
- Jāprojektē datu sadalīšanu starp servisiem, lai katrā servisā glabātos optimālais datu kopums;
- Jāadministrē katram mikroservisam vajadzīgas tehnoloģijas;
- Sarežģītāka sistēmas izvietošana, jo katram mikroservisam ir savs konfigurējums;
- Jākoordinējas ar citiem komandas dalībniekiem un komandām, ja servisam tiek mainīts API.

1.5.4. Analoģijas

Līdzīgas idejas, kurus izpauž mikroservisu arhitektūra, ir atrodamas UNIX operētājsistēmu filozofijā [9]:

1. Katrai programmai jādara vienu lietu labi;
2. Jāprojektē programmu tā, lai būtu iespējams palaist to cik vien iespējams ātri;
3. Nav jāvilcinās mest prom neefektīvas daļas un pārrakstīt tos;
4. Jārēķinās, kā katras programmas izejas dati kļūst par citas programmas ieejas datiem;
5. Pievienojot jauno funkcionalitāti, jāizveido jauno programmu, nevis jāapgrūtina veco programmu ar jauno funkcionalitāti.

Aktoru paralēlskaitļojumu modeļa koncepti arī ir līdzīgi mikroservisu arhitektūras principiem [10]:

1. Aktori ir primitīvie skaitļošanas kopumi, kuri saņem ziņojumus un veic kādu skaitļošanu, ir pilnīgi izolēti viens no otra, tāpēc sazinās tikai ar ziņojumu apmaiņu;
2. Aktori tiek izpildīti izolēti, tāpēc tie ir bojājumpiecietīgi;
3. Pastāv aktoru uzraugi, kuri vajadzības gadījumā veido jaunus aktorus.

1.5.5. Pielietošanas piemēri

Ir pazīstami jau vairāki tīmekļa servisi, kas tika pārveidoti no monolīta uz mikroservisu arhitektūru: eBay [1], Amazon [2], LinkedIn [3], Netflix [4], Soundcloud [7].

Netflix sistēma saņem vairāk par vienu miljardu pieprasījumu katru dienu no vairāk par 800 ierīču tipiem uz video straumēšanas API. Katrs API pieprasījums izraisa apmēram 5 izsaukumus aizmugures mikroservisiem.

Amazon arī tika migrējuši mikroservisu arhitektūrā. Viņi saņem neskaitāmo pieprasījumu skaitu no vairākām lietotnēm – ieskaitot lietotnes kas pārvalda tīmekļa API, kā arī pašu tīmekļa vietni – kas nebūtu iespējams vecajā divu slāņu arhitektūrā.

eBay izsole ir cits piemērs, kurā sistēma sastāv no vairākām atsevišķām lietotnēm, un katra no tiem izpilda loģiku no atsevišķām funkcionalitātes zonām [8].

2. IZSTRĀDĀTA SISTĒMA

Darba ietvaros piedāvāto risinājumu testēšanai tika daļēji izstrādātā sistēma, kas ir veidota uz mikroservisu arhitektūras pamata.

2.1. Apraksts

2.1.1. Sistēmas darbības sfēra

Sistēma ir tīmekļa serviss, kas ir orientēts uz mazām kompānijām un individuāliem komersantiem, kuriem ir vajadzība izrakstīt, automātiski sūtīt un kopumā pārvaldīt kā periodiskus, tā arī neperiodiskus rēķinus, un nav līdzekļu nodarbināt grāmatvedi kompānijas naudas lietu pārvaldībai.

2.1.2. Sistēmas funkcionalitāte

Sistēmas lietotājam ir iespējas:

- ātri veidot jaunus rēķinus;
- sūtīt periodiskus vai neperiodiskus rēķinus pa e-pastu, sūtīt atgādinājuma vēstules;
- apskatīt un automātiski sūtīt pa e-pastu vairāku veidu atskaites: neapmaksātie rēķini, apgrozījuma statistika pa nedēļām un mēnešiem;
- glabāt pastāvīgu klientu datus ar iespēju piešķirt tiem atlaidi;
- katram uzņēmumam pievienot vairākus lietotājus;
- integrēt sistēmu ar uzņēmuma citām sistēmām: eksportēt rēķinus JSON vai PDF formātā.

Sistēma nodrošina:

- drošu piekļuvi pie datiem, izmantojot vairāklīmeņu datu piekļuves pārvaldību;
- drošu datu glabāšanu, izmantojot šifrēšanas tehnoloģijas;
- atbilstību LR nodokļu likumiem rēķinu veidošanā.

2.1.3. Pirmkods

Sistēmas pirmkoda eksemplāri ir publiski pieejami GitHub programmkoda glabātuvē pēc adreses *https://github.com/dlouchansky/msc-microservices*.

2.2. Projektējums

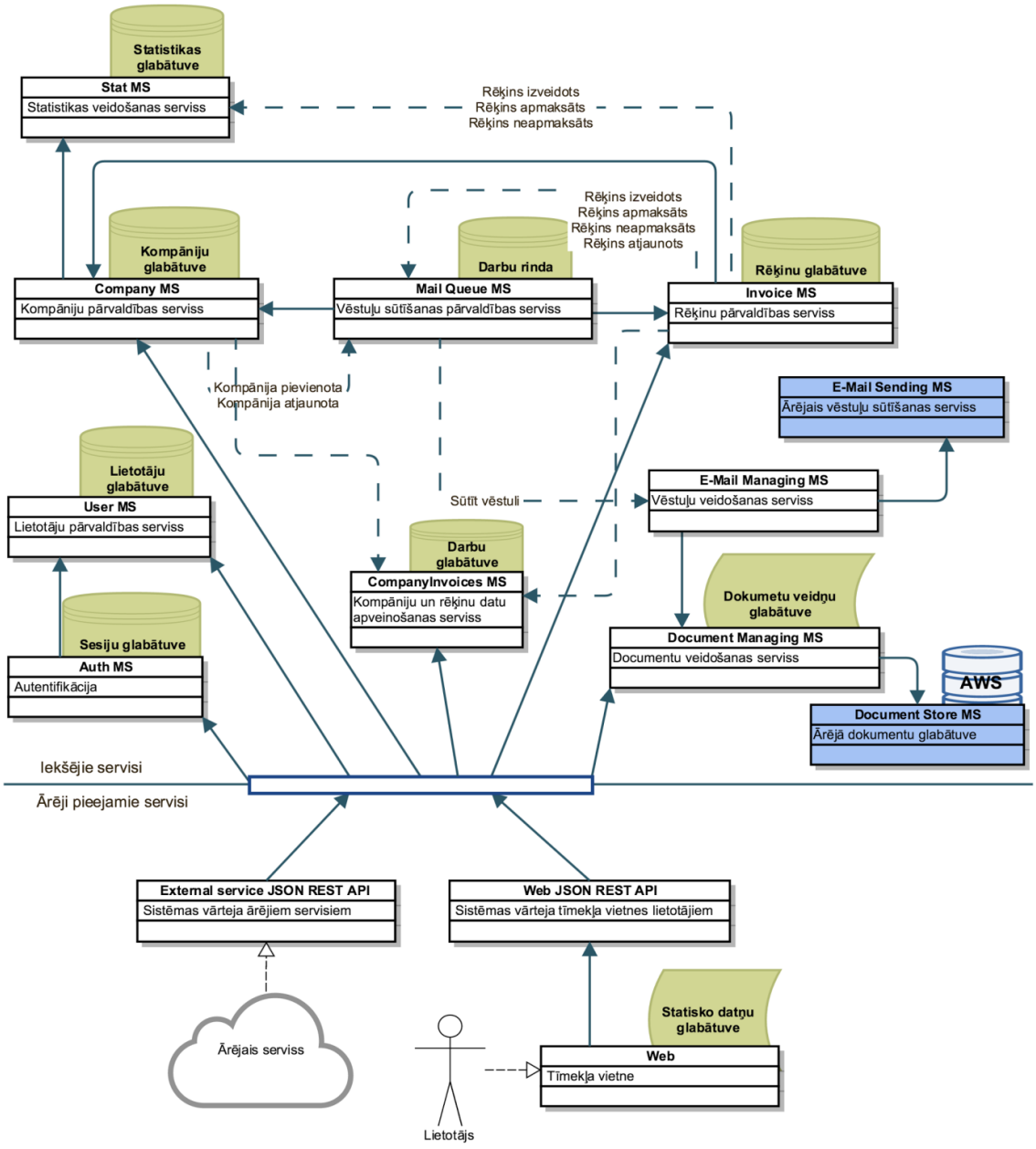
Sistēma sastāv no 12 mikroservisiem un ir integrēta ar 2 ārējām sistēmām. Mikroservisus iespējams sadalīt uz tiem, kuri ir publiski pieejami sistēmas lietotājiem no tīmekļa, un tiem, kuri ir pieejami tikai no lokālā tīkla.

Par datu attēlošanu ir atbildīgi paši pieprasījumus veicošie klienti (piemēram, pārlūkprogrammā izpildāmais JavaScript programmkods), tāpēc mikroservisos nav saskarnes veidošanas loģikas, un servisu pieprasījumu sūtīšana un atbilžu saņemšana notiek tikai JSON formātā.

Visiem mikroservisiem nav iekšējā stāvokļa – stāvoklis tiek glabāts datu bāzēs vai citās glabātuvēs – kas būtiski atvieglo servisu mērogošanu, jo starp viena servisa instancēm nav vajadzības sinhronizēt atmiņā glabātus datus.

Publiski pieejamie mikroservisi ir izstrādāti pēc projektēšanas šablona "aizmugursistēma priekšgalsistēmai" (BFF) [22] – katram sistēmas klienta veidam tiek izstrādāts savs API mikroserviss, lai neradītu atšķirīgo klientu loģikas savienošanos, jo bieži katram klienta viedam atšķīrās gan attēlošanai vajadzīgie dati, gan nepieciešamā funkcionalitāte. Tika izstrādāti sekojošie publiskie mikroservisi:

- *Web JSON REST API* mikroserviss – serviss, kurš ir pieejams pieprasījumiem no klientiem, kuri piekļuvei pie sistēmas izmanto pārlūkprogrammu. Servisā ir ievietota pieprasījumiem vajadzīgo datu apvienošanas un formatēšanas loģika;
- *External service JSON REST API* mikroserviss – serviss, kuru izmanto ārējās klientiem piederošas sistēmas, lai saņemtu atskaites un citus datus JSON formātā. Vienīgā strukturālā atšķirībā no Web JSON REST API – API tiek versionēts;
- *Web* mikroserviss – būtībā ir saskarnes veidošanas datņu glabātuve un to piegādātājs klientam. Datu saņemšanai no sistēmas izmanto Web JSON REST API mikroservisu.



2.1. att. **Izstrādātas sistēmas projektējums**

Privāti pieejamus mikroservisus ir iespējams sadalīt uz sinhroni un asinhroni pieejamiem:

- *Document Managing* mikroserviss satur dokumentu glabāšanas loģiku un sazinās ar ārējo dokumentu glabātuvi;
- *E-mail managing* mikroserviss formē vēstules un sazinās ar ārējo vēstuļu sūtīšanas servisu;

- *Mail queue* mikroserviss nolemj, kad un kāda tipa sūtīt vēstules – periodiskus un neperiodiskus rēķinus, kompāniju statistiku, atgādinājuma vēstules par neapmaksātiem rēķiniem;
- *Invoice* mikroserviss glabā un pārvalda rēķinus;
- *Company* mikroserviss glabā un pārvalda kompānijas;
- *CompanyInvoices* mikroserviss glabā apvienotus rēķinu un kompāniju datus sistēmas lietotāju pieprasījumu ātrākai apstrādei;
- *Stat* mikroserviss savāc un glabā statistiku par neapmaksātiem rēķiniem un kompāniju apgrozījumu;
- *User* mikroserviss glabā un pārvalda lietotāju kontus, kā arī satur datus par lietotājiem pieejamām darbībām;
- *Session* mikroserviss glabā autentificēto lietotāju sesijas.

2.1. attēlā sinhronie servisu izsaukumi ir attēloti ar nepārtrauktām bultām, asinhronie – ar pārtrauktām bultām. Tika attēloti tikai būtiskie izsaukumi.

Vēstuļu sūtīšanai un dokumentu glabāšanai tika nolemts izmantot ārējos servisus, jo abas darbības ir resursu ietilpīgās operācijas, un tīmeklī ir pieejamas vairākas šo funkciju realizācijas ar spēju vienkārši tās integrēt ar izstrādāto sistēmu.

1. pielikumā ir iespējams apskatīt projektējumu ar augstāko detalizāciju.

2.3. Izvēlētie rīki

2.3.1. Izvēles kritēriji

Katram sistēmas izstrādei un administrēšanai vajadzīgam rīku veidam pastāv vairākas alternatīvās realizācijas, tāpēc bija nolemts novērtēt un izvēlēties katru rīku pēc sekojošiem parametriem:

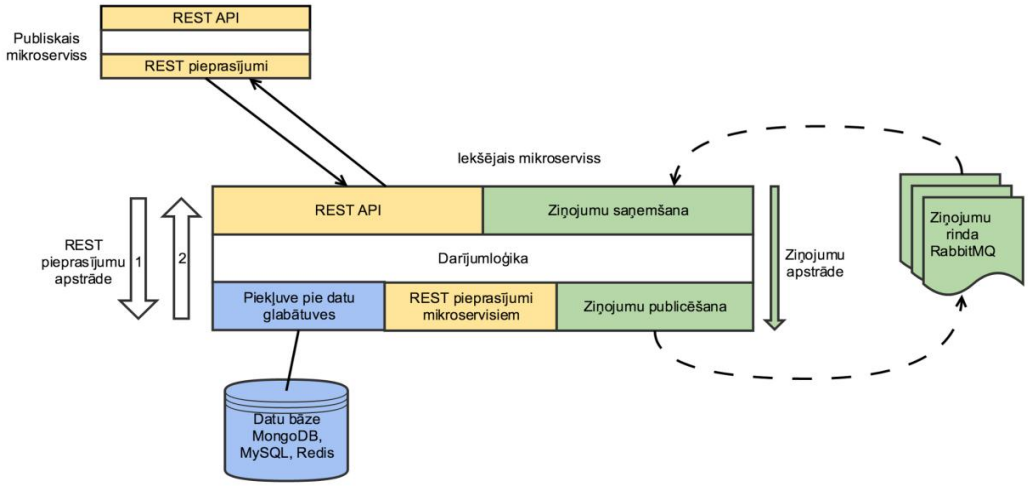
1. Vai rīks ir brīvās programmatūras un atvērta koda risinājums;
2. Cik rīks ir viegls lietošanā;
3. Vai rīks ir stabils un jau tiek izmantots citu sistēmu izstrādē;
4. Cik kvalitatīva un pilna ir dokumentācija;
5. Kādā mērā ir iespēja nokonfigurēt risinājumu savām vajadzībām.

2.3.2. Implementēšanas rīki

Šajā sadaļā tiek aprakstīti ar programmēšanu saistītie rīki, kas bija izvēlēti sistēmas izstrādei.

Par pamata programmēšanas valodu tika izvēlēta Java valoda. Tieši Java programmētāju kopiena bija viena no pirmajiem, kas sāka popularizēt mikroservisu arhitektūru, un pašlaik jau ir pieejami vairāki ietvari un palīgbibliotēkas, kas ir tieši orientēti sistēmas izstrādei izmantojot mikroservisus.

Visi sistēmas Java mikroservisi bāzējās uz Spring ietvara mikroservisu orientētās atvieglotās modifikācijas – Spring Boot – kas dod iespēju viegli programmēt integrēšanas loģiku ar citiem mikroservisiem, veidot REST API, ērti izveidot mikroservisa izpildāmo datni un pēc tam ātri palaist servisu. Spring Boot izmanto iekšējo tīmekļa serveri Apache Tomcat. Apache Maven tiek izmantots kā programmatūras atkarību pārvaldības un kompilēšanas automatizācijas rīks.



2.2. att. **Mikroservisu struktūra**

Par otru valodu tika izvēlēts JavaScript. Tas tiek izmantots kā saskarnē, tā arī publisko API mikroservisu programmkodā, mikroservisos izmantojot Node.JS dzinēju. Sistēmas noklusētam datu apmaiņas formātam JSON ir dzimts atbalsts Javascript valodā, kas Node.JS bāzētiem mikroservisiem padara datu saņemšanu no iekšējiem servisiem, to pēcapstrādi, apvienošanu un sūtīšanu klientam ļoti vieglu.

Node.JS mikroservisi izmanto Restify ietvaru, kas fokusējas tieši no ārpuses pieejamo API izstrādē. Node.JS dzinējā ir iekļauta tīmekļa servera veidošanas iespēja. Npm tiek izmantots kā programmatūras atkarību pārvaldības un kompilēšanas soļu automatizācijas rīks.

Programmkods tiek glabāts GIT versiju kontroles sistēmā.

Dati tiek glabāti MongoDB un MySQL datu bāzēs vai Redis glabātuvē atkarībā no konkrētā mikroservisa vajadzībām. Redis tiek arī izmantots kešoto datu glabāšanai.

Ziņojumu apmaiņai tika izvēlēta RabbitMQ programmatūra.

2. 2. attēlā ir paradīta katra izstrādāta mikroservisa iekšējo moduļu struktūra.

2.3.3. Administrēšanas rīki

Šajā sadaļā tiek aprakstīti ar mikroservisu administrēšanu saistītie rīki, kas bija izvēlēti sistēmas izstrādei.

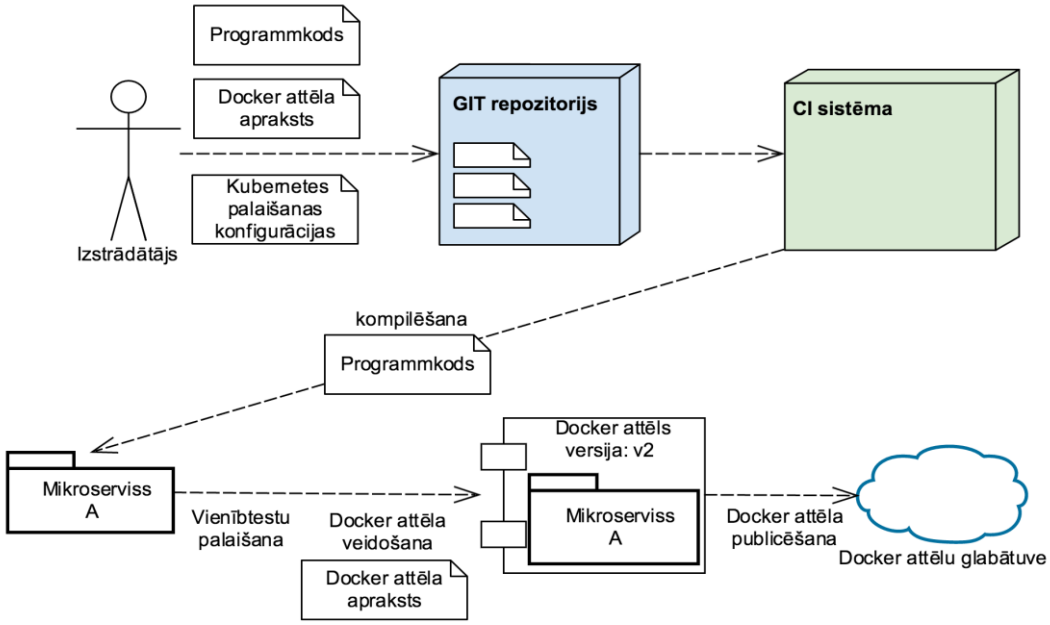
Par pamatu sistēmas administrēšanai tika izvēlētas Docker un Kubernetes tehnoloģijas.

Docker ir programmatūra, kas dod iespēju automatizēt atsevišķo sistēmas daļu izvietošanu, dalīšanos un pārvaldību. Galvenā Docker struktūrvienība ir konteineri, kuros tiek palaista jebkura programmatūra (darba ietvaros – mikroservisi, datu glabātuves, mikroservisu pārvaldošā un cita infrastruktūras programmatūra). Konteineris ir izolēts, salīdzinājumā viegls virtuālās mašīnas analogs, kas var būt palaists dažu sekunžu laikā. Konteineri ir veidoti pēc šablona – nokompilētā Docker attēla. Attēls ir aprakstīts speciālajā datnē Dockerfile un pēc kompilēšanas var būt saglabāts speciālajā attēlu glabātuve, no kurienes citi izstrādātāji varēs to ielādēt.

Attēli tiek veidoti uz jau esošo attēlu pamata – piemēram, par pamatu ir iespējams izvēlēt Linux sistēmu ar jau instalēto Node.JS dzinēju, un konfigurācijas datnē aprakstīt tikai mikroservisa programmkoda kopēšanas un palaišanas komandas.

Docker pats optimizē attēlu glabāšanu – ja vairāki izveidotie attēli ir izstrādāti pēc kopējā paraugattēla, sistēmā glabājas viens paraugattēls un katrā izveidotā attēla tajā veiktās izmaiņas. Operatīvās atmiņas ziņā arī notiek optimizācijas – konteineriem vajadzīgas atmiņas mērījums ir pieejams 15. pielikumā.

Mikroservisu izvietošana un palaišana Docker konteineros dod iespēju unificēti tos pārvaldīt, katram servisam veidojot tam piemērotāko palaišanas vidi un bez riska konfliktēt ar citiem mikroservisiem par kopīgām palaišanas vides atkarībām.



2.3. att. **Paraugs automatizētai Docker attēla publicēšanai pēc izmaiņas publicēšanas GIT**

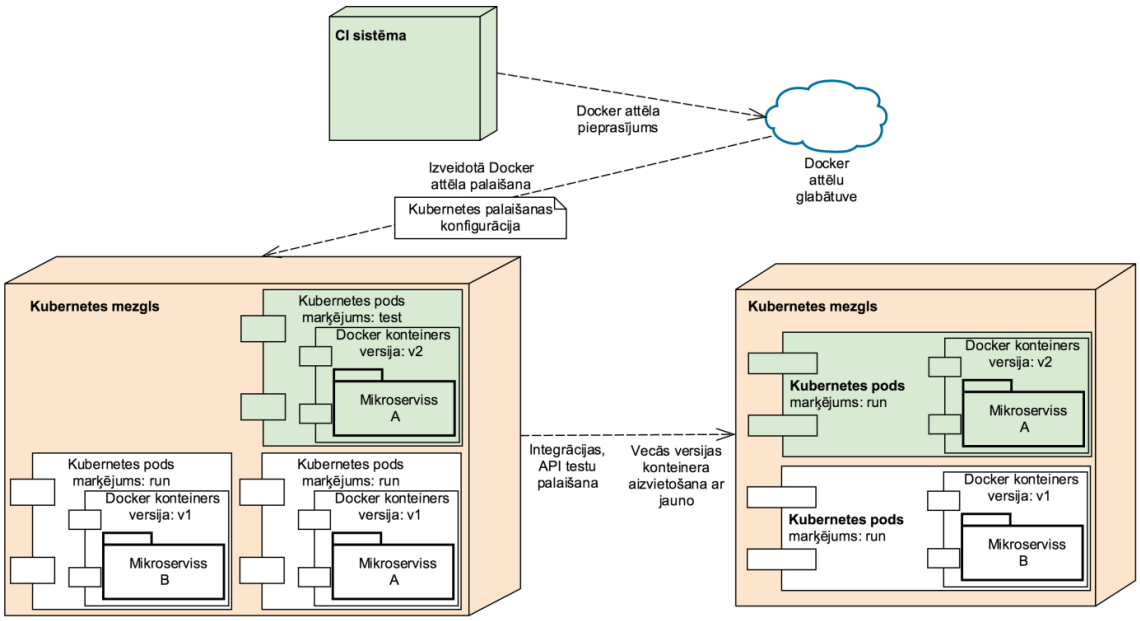
Kubernetes ir izstrādātais Google kompānijā Docker konteineru pārvaldnieks, kas atvieglo konteineru izvietošanu un uzraudzību kā vienā, tā arī vairākos serveros vai PaaS piegādātāju mašīnās, ar iespēju pārvaldīt līdz tūkstošiem mašīnu [23]. Pašlaik tas atrodas aktīvās izstrādes stadijā, bet jau pastāv stabilas, ražošanā strādājošas versijas. Atšķirībā no citiem pārvaldības rīkiem, Kubernetes ir universālais mikroservisu pārvaldības rīks, kas mēģina iekļaut sevī visu mikroservisu palaišanai vajadzīgo pamatfunkcionalitāti. Kubernetes ievieš vairākas Docker konteineru pārvaldības abstrakcijas:

- Pods (Pod) – konteineru kopums, kurus vienmēr ir jāizvieto vienā serverī, piemēram, ja vienā konteinerī ir tīmekļa servētāja programmatūra un otrā ir atkarīgā sistēma, kas to izmanto. Ir rekomendēts vienā podā izvietot vienu konteineri, tāpēc izstrādātā sistēmā vienā podā vienmēr tiek izvietot viens konteiners;
- Replikācijas pārvaldnieks (Replication Controller) – podu uzraugs, kurš kontrolē, ka sistēmā katrā laika momentā ir palaists iepriekš definēts noteiktā veida podu skaits, un vajadzības gadījuma startē jaunus poda instances;
- Serviss (Service) – vajadzīgs ērtai piekļuvei pie Kubernetes klasterī palaistiem resursiem un slodzes balansēšanai vairāku viena poda veida instanču darbošanās gadījumā;

- Marķējums (Label) – identifikators, ar kuru var būt aprakstīta viena vai vairākas iepriekš aprakstītas Kubernetes vienības;
- Mezgls (Node) – serveris, kurā tiek palaista Kubernetes rīka uzraudzības process;
- Klasteris (Cluster) – mezglu kopums, kurā pastāv vismaz viens galvenais mezgls ar Kubernetes darbības vajadzīgiem servisiem.

Citi darba ietvaros nozīmīgie rīki, kas tika izmantoti problēmu risināšanai, ir aprakstīti attiecīgajās sadaļās.

2.3. un 2.4. attēlā ir iespējams redzēt automātiskās izvietošanas soļus sistēmai, kura izmanto Docker mikroservisu palaišanai un Kubernetes konteineru pārvaldībai. 2.3. attēlā ir redzamas darbības, kuras notiek pēc pirmkoda publicēšanas versiju kontroles sistēmā, 2.4. attēlā ir radīti mikroservisa izvietošanas un palaišanas soļi.



2.4. att. **Paraugs automatizētai Docker attēla palaišanai Kubernetes klasterī**

3. MIKROSERVISU ARHITEKTŪRAS PROBLĒMAS

3.1. Ievads

Pirms programmatūras izstrādes uzsākšanas mikroservisu arhitektūrā vai esošas sistēmas arhitektūras mainīšanas uz mikroservisu arhitektūru ir jāsaprot, ar kādiem šķēršļiem būs jāsastapās izstrādes gaitā, jo, nezinot tās, paaugstinās risks projekta neveiksmīgai pabeigšanai, tā kā problēmu risināšana var patērēt milzīgus resursus.

Problēmas tika sadalītas divās sadaļās – implementēšanas problēmās, ar kuriem sastapās programmatūras izstrādātāji un testētāji, un administrēšanas problēmās, ar kuriem tradicionāli sastapās galvenokārt citi informācijas tehnoloģiju speciālisti. Problēmas tika galvenokārt formulētas uz [13] grāmatas, [14, 15] rakstu bāzes, autora pieredzes monolīta programmatūras implementēšanā, kā arī darbā apskatītās sistēmas realizācijas gaitā saņemtas pieredzes.

Katrai problēmai ir pieejama vispārēji aprakstīta risināšanas stratēģija un konkrētais stratēģijas pielietošanas gadījums. Risinājumos tiek pieņemts, kā sistēmas izstrādē ir iesaistīti vairāki cilvēki, lai būtu vieglāk piemērot to sistēmas veidošanai reālajā vidē.

Pieņemtais Docker un Kubernetes administrēšanas veids ir izmantojot komandrindu, tāpēc tālāk tekstā ir pieņemts, kā slīprakstā uzrakstītas komandas ir jāpilda OS X vai Linux operētājsistēmu komandrindā. Tiek arī pieņemts, kā darba lasītājam jau ir instalēts Docker rīks.

3.2. Mikroservisu administrēšana

3.2.1. Versionēšana

3.2.1.1. Apraksts

Mikroservisu arhitektūra ir domāta priekš pastāvīgiem neatkarīgiem mikroservisu atjauninājumiem. Atjaunināšanas procesā bieži tiek mainīta funkcionalitāte un ārējais servisa izmantošanas līgums.

Ja servisa līgums tika mainīts, jaunās versijas izvietošanas procesam ir jāpievērš lielāku uzmanību, nekā vienkārši jāpalaiž jauno mikroservisa versiju. Pat mazākā izmaiņa līgumā var

salauzt sistēmas daļu vai visu sistēmu, jo no gandrīz katra mikroservisa ir atkarīgs kāds cits serviss.

Jāpastāv pieejai, kā kontrolēt un izsekot līguma izmaiņas, nelaužot mikroservisus, kuri ir pielāgoti vecās līguma versijas izmantošanai, vai pārdomāt izziņošanas mehānismu līguma maiņas gadījumā.

3.2.1.2. Risināšanas stratēģija

API versionēšanas izmantošana projektā palielina izstrādes, testēšanas, atbalsta un pārvaldes sarežģītību. Izstrādātājiem ir jāatbalsta vairākas mikroservisa versijas, kas nozīmē lielāko pirmkoda apjomu un algoritmu komplikāciju. Testēšanas inženieriem ir jātestē visas pieejamas versijas, tāpēc testēšanas apjoms pastāvīgi palielinās. Jāpārvalda atkarīgo servisu izziņošanu par jauno versiju palaišanu, bet pašlaik nav pieejams standartizēts izziņošanas mehānisms. Tādēļ pastāv viedoklis, kā pēc iespējas ir jāizvairās no versionēšanas [24].

Apskatītam projektam tika nolemts analizēt katru mikroservisu atsevišķi un pēc iespējas ir jāatteicas no versionēšanas.

Aizvietot versionēšanu nolemts pavadot vairākas aktivitātes:

- publiskot rokasgrāmatu ar API veidošanas un mainīšanas noradījumiem, pie kuriem ir jāturas izstrādājot mikroservisus, lai servisa attīstības gaitā būtu viegli veikt izmaiņas līgumā;
- definēt klienta pieprasījumu veikšanas pieeju, lai līguma negaidītas mainīšanas gadījumā klients varētu to apstrādāt bez vajadzības veikt izmaiņas savā servisā – būt par tā saucamo “toleranto lasītāju” [25];
- izmantot līgumu testēšanu – pieeju, kad testus raksta servisa lietotāji, nevis paši servisa izstrādātāji, un izstrādātājiem ir jāveido programmatūru tā, lai tos nesalauzt – kas aizsargās atkarīgus servisus no kļūdām līguma mainīšanas gadījumā, jo tie nebūs ietekmēti, vai būs laicīgi brīdināti;
- dot iespēju ērtā veidā apskatīt katra mikroservisa esošā API dokumentāciju.

Versionēšanu tika nolemts izmantot tajos mikroservisos, kuriem API nav iespējas notestēt ar līgumu testēšanu. Izstrādātajā sistēmā vienīgais serviss, kuram ir jāversionē API, ir

mikroserviss, kurš ir publiski pieejams ārējām sistēmām – External REST API – jo nav iespējas komunicēt ar visiem klientiem.

Versionēšanas pārvaldībai tika nolemts izveidot atsevišķo dokumentu, kur jābūt aprakstītam, kā notiek piekļuve pie atsevišķām servisa versijām, kāds ir veco versiju atbalsta periods, kā tiek sasniegta atgriezeniskā saderība.

3.2.1.3. Risinājuma piemērs

Versionēšanas rokasgrāmata ir pieejama 3. pielikumā.

Par mikroservisu dokumentēšanas rīku tika izvēlēts Swagger ietvars, kas dod iespēju automātiski ģenerēt REST API dokumentāciju. Dokumentācijas ģenerēšanas iestatīšanas piemērs mikroservisam uz Spring Boot ietvara ir pieejams 4. pielikumā. Node.JS mikroservisiem ir arī pieejamas Swagger ģenerējošas bibliotēkas – apskatītā projektā tiek lietota swagger-restify bibliotēka, kas ir tieši piemērota apskatītas sistēmas mikroservisos izmantotām Restify ietvaram. Automātiski ģenerētas dokumentācijas glabāšanai API atjaunošanas gadījumā tika izveidots atsevišķais repozitorijs versiju kontroles sistēmā.

Ja mikroserviss izmanto versionēšanu, klientiem ir iespēja repozitorijā saņemt dokumentāciju visām versijām, kuras pašlaik ir atbalstītas.

Pašlaik nav pieejams rīks dokumentācijas veidošanai mikroservisiem, kuri publicē vai parakstās uz asinhroniem ziņojumiem, tāpēc tika izveidots atsevišķais repozitorijs versiju kontroles sistēmā, kur katram ziņojumu publicējošam servisam ir sava mape un servisa izstrādātājiem ir pienākums aprakstīt katrā ziņojumā iekļautus datus, skaidrojumu par ziņojuma sūtīšanas cēloni, un rindas nosaukumu, kurā ziņojums tiek publicēts. Katram ziņojumam jābūt savā datnē.

Līgumu testēšanas pieeja tiek aprakstīta 3.3.1. sadaļā.

API rokasgrāmata ar veidošanas un mainīšanas noradījumiem un pieprasījumu veikšanas pieejas aprakstu ir pieejama 2. pielikumā.

3.2.2. Izvietošana

3.2.2.1. Apraksts

Tā kā mikroservisu skaits var būt milzīgs, izvietošanas procesam jābūt automatizētam, jo manuālā izvietošana tērē daudz izstrādātāju un administratoru laika un tāpēc nebūs iespējams sasniegt nepārtrauktas integrācijas (CI) un izvietošanas (CD) pieeju mērķus.

Izvietošanai arī ir jābūt viegli konfigurējamai, lai varētu bez sarežģījumiem uzstādīt mikroservisus neatkarīgi no tehnoloģiju kopuma, kuru tie izmanto.

Izstrādātāju vidē jābūt iespējai izvietot vienu mikroservisu vai mikroservisu kopumu, lai, piemēram, varētu notestēt mikroservisa integrāciju ar atkarīgiem servisiem.

3.2.2.2. Risināšanas stratēģija

Pašlaik pastāv vairākas lietojumprogrammu izvietošanas pieejas, kas atšķīrās ar palaišanas vides konfigurācijas abstrakcijas līmeni:

- Tradicionālākais izvietošanas veids ir tieši pārvaldot fiziskus serverus. Parasti tiek pirkts vai īrēts serveru kopums, katrā serverī tiek instalēta programmas palaišanai vajadzīgā programmatūra. Risinājums ir ērts, ja izstrādātājiem pastāv vajadzība palaist vienā serverī tikai vienu lietojumprogrammu, jo vairāku programmu palaišana kļūst grūtāka, tā kā var parādīties konflikti servera operētājsistēmas, palīgprogrammatūras un globāli pieejamo bibliotēku koplietošanā. Serveru automatizētai pārvaldībai ir pieejama Chef un Puppet programmatūra.
- Jau ilgu laiku ir populārā programmatūras izvietošana virtuālajās mašīnās, kas dod iespēju vienā serverī izvietot vairākas sistēmas, katru savā palaišanas vidē. Atsevišķās palaišanas vides atrisina koplietojamo rīku konfliktus. Par trūkumiem iespējams uzskatīt virtuālo mašīnu prasīgumu pret sistēmas resursiem, jo tie būtībā neatšķīrās no parastajām operētājsistēmām, un samērā ilgo palaišanas laiku.
- Pēdējo pāris gadu laikā ieguva popularitāti konteineru izmantošana programmatūras izvietošanai, kurai piemīt virtuālo mašīnu lietošanas priekšrocības, bet konteineri ir neprasīgie pret sistēmas resursiem un startējas dažu sekunžu laikā. Pašlaik visizmantotākā konteineru realizācija ir Docker. Ir pieejams liels konteineru pārvaldības rīku kopums, populārāki ir Docker Swarm, Kubernetes, Apache Mesos. Tīmeklī Docker konteinerus ir iespējams izvietot Google Cloud Platform, Amazon

EC2 servisos. Lielākais trūkums ir jauns vecums, sakarā ar to pārsvarā rīki vēl ir izstrādes stadijā, tāpēc programmatūru ir jāizvēlas ar piesardzību.

- Mākoņskaitļošanas popularitātei palielinoties, parādījās iespēja izvietot programmatūru pie PaaS piegādātājiem, kuri paši izvēlās, kā pilnībā pārvaldīt programmatūru un tās infrastruktūru. No trūkumiem jānorāda pieķeršanās pie konkrēta piegādātāja un to tehnoloģiju kopuma; neiespējamību optimizēt resursu izmantošanu, jo nav, vai ir vienkāršota piekļuves iespēja pie palaišanas vides; vajadzība izveidot atsevišķu programmatūras lokālās palaišanas konfigurāciju. Populārāki PaaS piegādātāji ir Amazon Beanstalk, Heroku, Google App Engine, RedHat Openshift.

3.2.2.3. Risinājuma piemērs

Darbā apskatītai sistēmai tika izvēlēts izmantot izvietošanu Docker konteineros ar Kubernetes pārvaldnieku, jo mikroservisu arhitektūras lietošanas gadījumā pirmās divas pieejas nav efektīvas resursu ziņā, bet PaaS rīku prasība izmantot konkrētās tehnoloģijas infrastruktūras pārvaldībai ir neapmierinoša.

Mikroservisa izvietošanas soļi:

1. Mikroservisa palaižamā programmkoda ievietošana Docker attēlā un saglabāšana lokālajā glabātuvē. Ir jāizveido mapi, kur jāieliek attēlu aprakstošo Dockefile un programmas pirmkodu, un jāizpilda komandu:
   docker build -t attela_nosaukums .
   Dockerfile datnes piemērs Node.JS un Java bāzētiem mikroservisiem ir pieejams 5. pielikumā.
2. Attēla izvietošana attēlu glabātuvē (apskatītam projektam tika izmantota publiski pieejamā glabātuve Docker Hub, bet pastāv iespēja instalēt privāto Docker konteineru glabātuvi):
   docker push -t dlouchansky/msc-api-web:latest .
   Iespējams noradīt attēla versiju (piemērā versija ir *latest, dlouchansky/msc-api-web* ir attēla nosaukums), lai dažādās vidēs varētu palaist atšķirīgas versijas.
   Attēla ielāde serverī notiek konteinera palaišanas brīdī, ja attēla vajadzīga versija jau nav saglabāta serverī.

3.2.3. Palaišana

3.2.3.1. Apraksts

Palaišana ir process, kas nāk tieši pēc mikroservisa izvietošanas. Tam arī jābūt automatizētām un vieglam, lai varētu vienkārši palaist vienu vai vairākus mikroservisus vienlaicīgi, un jāspēj izdarīt to neaizskarot citus strādājošus mikroservisus, lai sasniegtu neatkarīgas mikroservisu izvietošanas mērķi.

Tā kā mikroservisi tiek palaisti pilnīgi izolēti, jābūt izveidotam mehānismam, kas, pēc mikroservisa palaišanas, paziņo atkarīgiem mikroservisiem par jaunā mikroservisa parādīšanos, kā arī reģistrē mikroservisu pārraudzības un citos administrēšanas mehānismos.

Mikroservisa darbības beigšanas gadījumā visus atkarīgus servisus arī ir jābrīdina par notikumu.

3.2.3.2. Risināšanas stratēģija

Tā kā mikroservisi tiek izvietoti kā abstrakti Docker konteineri, to palaišanas instrukcija ir aprakstīta Dockerfile datnē un vienkārša konteinera palaišana notiek ar vienas komandas izpildīšanu serverī. Konteineriem nav savstarpējo atkarību, tāpēc palaišana notiek atomāri un jau strādājošie konteineri nekā nav ietekmēti.

Darba ietvaros izstrādātai sistēmai ir jādarbina vairākus konteinerus un jāpaziņo atkarīgus konteinerus par palaišanas notikumu, tāpēc palaišanas automatizēšanai tika nolemts izmantot konteineru pārvaldnieku Kubernetes. No pieejamiem risinājumiem tika izvēlēts tieši Kubernetes pārvaldnieks, jo tajā jau ir iekļauti vairāki mikroservisu administrēšanai vajadzīgie rīki un tas jau ir izmantots lielā mēroga tīmekļa sistēmu darbībā.

Kubernetes pārvaldnieks atvieglo vairāku konteineru vienlaicīgu palaišanu, mērogošanu vairākos serveros, slodzes balansēšanu, konteineru atjaunošanu, glabā un piedāvā saites uz visiem palaistiem konteineriem.

Aprakstītai sistēmai pastāv trīs palaišanas vides – izstrādātāju (pieejama konteineru palaišanai sistēmas izstrādātājiem lokāli), testēšanas (pieejama sistēmas izstrādē iesaistītiem cilvēkiem un automatizētai integrācijas testēšanai), un produkcijas (pieejama publiski visiem lietotājiem) vides. Katrai videi ir savs konfigurācijas datņu kopums. Lokāli Kubernetes klasteris tiek palaists bez pārraudzības un citiem palīgservisiem, jo to lietošanai ir vajadzīgi ievērojami sistēmas resursi.

Mikroservisam vajadzīgie konfigurācijas parametri, piemēram, citu palaisto mikroservisu adreses, piekļuves dati pie datu glabātuvēm, ir ievietoti konfigurāciju datnēs un nodoti programmkodam vides mainīgo veidā, tāpēc nav vajadzības pielāgot pirmkodu katrai palaišanas videi.

Privāto konfigurācijas datu glabāšanai tika izvēlēta specializētā Kubernetes sastāvdaļa – Secrets – privāto datu glabātuve, kas dod iespēju konfigurāciju datnēs noradīt tikai datu identifikatoru, pēc kura palaišanas brīdī Kubernetes saņem vajadzīgos datus no glabātuves.

3.2.3.3. Risinājuma piemērs

Docker konteinera vienkāršā palaišana bez Kubernetes pārvaldības notiek ar vienu komandu:

docker run -p 8008:8008 --name msc-api-web -d dlouchansky/msc-api-web,

kas nozīmē, ka tiks palaists *msc-api-web* mikroserviss, un tā konteinera 8008 ports būs piesaistīts 8008 servera portam, tāpēc pieprasot servera 8008 portu, pieprasījums tiks automātiski novirzīts konteineram.

Katra mikroservisa palaišanai ar Kubernetes parasti tiek izveidotas divas konfigurāciju datnes – Kubernetes replikācijas pārvaldniekam (piemēru iespējams apskatīt 6. pielikumā) un Kubernetes servisam (piemērs pieejams 7. pielikumā). Vairāk par šīm Kubernetes sastāvdaļām un to konfigurāciju datnēm ir aprakstīts 3.2.4. un 3.2.5. sadaļās. Konteineru palaišana notiek, izmantojot *kubectl* komandrindas palīgprogrammatūru saziņai ar Kubernetes klasteri:

kubectl create -f msc-api-web-controller.yaml

kubectl create -f msc-api-web-service.yaml,

kur *msc-api-web-controller.yaml* un *msc-api-web-service.yaml* ir konfigurāciju datnes, uz kuru bāzes tiks izveidots gan replikācijas pārvaldnieks ar podiem, gan serviss, gan konteineri ar mikroservisiem tajās.

Lai pārliecinātos, kā vajadzīgas vienības ir palaistas, un apskatīt katras izveidotās instances identifikatoru, eksistē komanda

kubectl get vienības_tips

kur *vienības_tips* ir serviss (*srv*), pods (*po*), replikācijas pārvaldnieks (*rc*).

Lai atjaunotu konteinerus, nav vajadzības vispirms atslēgt vecās versijas palaistas instances, ja konteineriem ir sakonfigurēts replikācijas pārvaldnieks:

kubectl rolling-update msc-api-web-controller -f msc-api-web-controller.yaml,

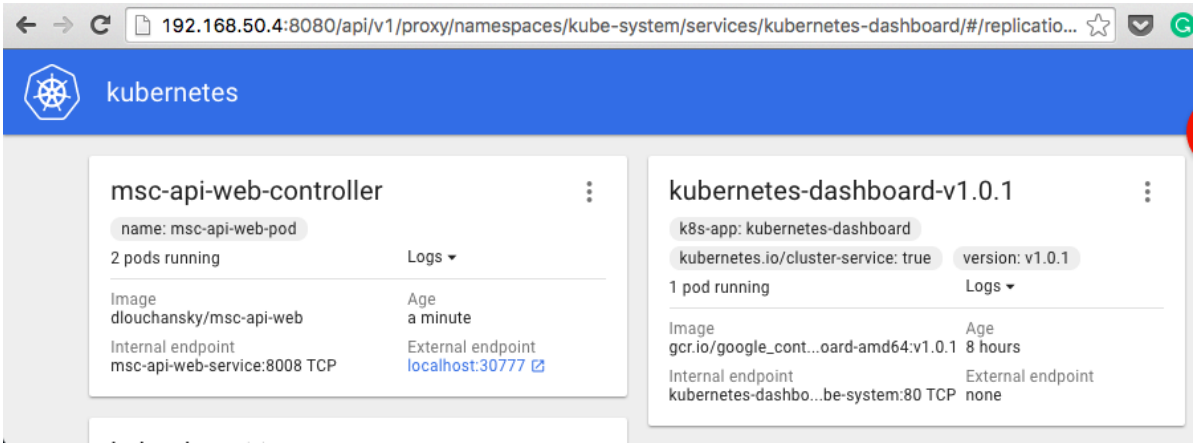
šī komanda pa vienam pievieno jaunās versijas instances, kuras definētas *msc-api-web-controller.yaml* replikācijas pārvaldnieka konfigurācijas datnē, un atslēdz vecas, kuras pašlaik tiek pārvaldītas ar *msc-api-web-controller* pārvaldnieku.

Kubernetes klasteris tika arī palaists lokāli, un tika ievietots virtuālajā mašīnā, lai negaidījumu gadījumā cita sistēma būtu izolēta. Tā kā lokāli arī ir izmantots Kubernetes, ir iespējams visām palaišanas vidēm veidot līdzīgas konfigurācijas.

Konfigurācijas datu slēpšana ir aprakstīta 3.2.8. sadaļā.

Tā kā mikroservisu izvietošana un palaišana var būt panākta ar nelielo operāciju skaitu, ir iespējams vienkārši automatizēt šo procesu ar nepārtrauktās integrācijas rīkiem kā Jenkins, Bamboo un orientētam tieši automatizētai palaišanai Kubernetes klasterī Spinnaker.

Kubernetes piedāvā iespēju ērti ielādēt konfigurācijas datnes un palaist attiecīgus resursus caur administratora paneli, kā arī apskatīt informāciju par podiem, kuri ir palaisti ar replikāciju pārvaldniekiem, apskatīt konkrēta poda notikumu žurnālu un citu informāciju. 3.1. attēlā redzams, ka klasterī ir palaisti 2 *msc-api-web* podi, kuri ir pieejami pēc 30777 porta un kurās ir palaists Web REST API mikroserviss.



3.1. att. **Kubernetes administratora panelis**

15. pielikumā ir pieejams Docker konteinera palaišanai vajadzīgas operatīvās atmiņas mērījums. 8. pielikumā ir aprakstīta Kubernetes klastera palaišana AWS EC2 serveros.

3.2.4. Slodzes balansēšana

3.2.4.1. Apraksts

Ja tiek palaistas vairākas viena tipa mikroservisu instances, ir jāpārvalda pieprasījumu virzīšanu, lai pieprasījumi tika virzīti pie instancēm vienmērīgi, ņemot vērā katras instances

noslogotību, un slodzes paaugstināšanas gadījumā tiktu atsūtīts ziņojums mērogošanas mehānismam jaunās instances veidošanai. Balansētājam jābūt piemērotam pieprasījumu virzīšanai vairākos serveros.

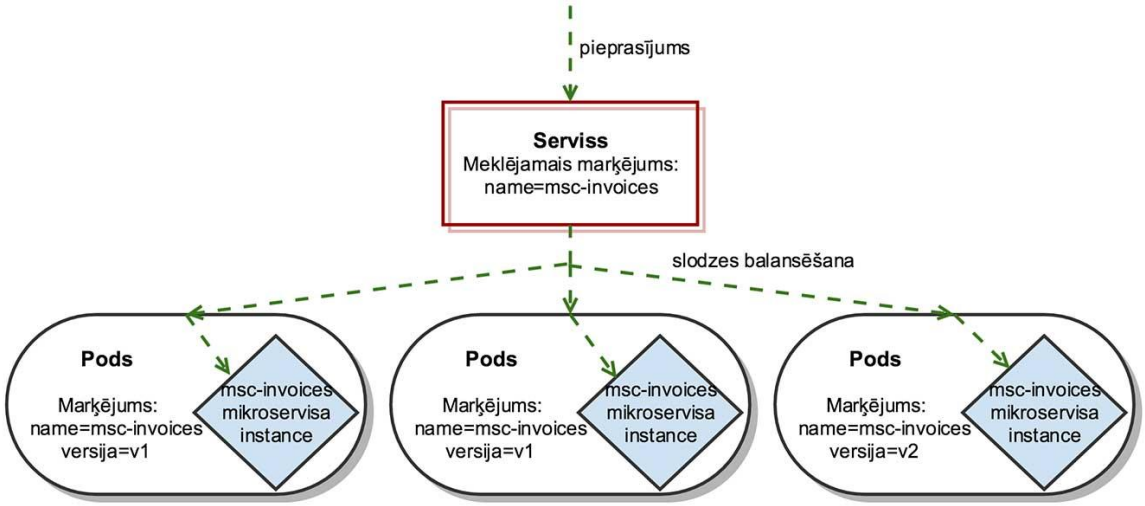
3.2.4.2. Risināšanas stratēģija

Slodzes balansēšana būtībā ir mikroservisa instances izvēle no pašlaik palaistiem klasterī. Pastāv divi mikroservisu pasaulē izplatītie instances saites iegūšanas veidi [26]:

- klienta puses pakalpojumu atklāšana – šajā gadījumā klients (mikroserviss vai lietotāja saskarne) saņem no specializētā pakalpojumu reģistra, kurā tiek glabātas saites uz visiem strādājošiem mikroservisiem, pieejamo instanču saišu sarakstu un pats nolemj, kādai instancei veikt pieprasījumu;
- servera puses pakalpojumu atklāšana – šajā gadījumā klients nav atbildīgs par precīzo saišu iegūšanu un pieprasīšanu, bet veic pieprasījumu slodzes balansētājam, kurš jau pieprasa specializēto pakalpojumu reģistru. Klients veic pieprasījumu, izmantojot abstrakto mikroservisa adresi, un balansētājs pats izvēlas, kuru no instancēm jāpieprasa.

Tika nolemts slodzes balansēšanas loģiku cik vien iespējams pārvietot uz infrastruktūras līmeni, tāpēc apskatītā sistēmā tiek pielietota servera puses pakalpojumu atklāšana. Servera puses slodzes balansēšanai tika izmantoti Kubernetes servisi.

Kubernetes serviss ir abstrakcija, kas dod iespēju piekļūt pie viena no līdzīgiem podiem, kuri var būt izvietoti vairākos serveros, izmantojot loģisko nemainīgo adresi – marķējumu. Katram podam tiek definēts marķējums, un serviss atlasa no visiem pieejamiem podiem tādus, kuriem marķējums atbilst vajadzīgam, un pēc tam pieprasījuma apstrādei izvēlās vienu no tiem.



3.2. att. **Slodzes balansēšana izmantojot Kubernetes servisu**

Pastāv trīs Kubernetes servisu veidi:

- iekšējais, kas ir pieejams tikai citām Kubernetes vienībām;
- ar piekļuvi no ārpuses, kas ir pieejams pieprasījumiem ārpus Kubernetes klastera;
- ar ārējo slodzes balansētāju, ja ir vēlme aizvietot standarta slodzes balansēšanas mehānismu.

Kubernetes servisu slodzes balansēšanai tiek izmantots vienkāršais apļkārtes (*round-robin*) algoritms, kas ir pieņemams apskatītā projekta realizācijai.

3.2.4.3. Risinājuma piemērs

Katram Kubernetes servisam ir piešķirts savs nosaukums, pēc kura ir iespējams ērti pieprasīt vajadzīgo mikroservisu, ja klients atrodas Kubernetes klasterī. Pieprasījuma piemērs:

http://msc-web-api-service/

kur *msc-web-api-service* ir konfigurācijas datnē noradītais Kubernetes servisa nosaukums.

Ja klients atrodas ārpus Kubernetes klastera, ir jālieto servisu ar iespēju piekļūt no ārpuses. Šiem servisiem konfigurācijā ir jānorāda portu, ar kuru būs iespējams pieprasīt servisu ārpus klastera, *nodePort*. Ārējā tipa serviss kļūs pieejams pieprasījumiem no jebkura Kubernetes mezgla pēc noradītā porta.

Kubernetes servisa konfigurēšanas piemērs ir pieejams 7. pielikumā. Piemērā tiek konfigurēts no tīmekļa pieejams serviss, kas balansē slodzi starp Web REST API mikroservisa instancēm.

3.2.5. Mērogošana

3.2.5.1. Apraksts

Ja tiek konstatēts, kā kāda mikroservisa instances ir pilnībā noslogotas un ir jāpievieno jaunās, vai, pretējā gadījumā, instances nav noslogotas un tos iespējams noņemt, jābūt automātiskām mehānismam, kurš šo procesu pārvaldīs. Jaunās instances pievienošanas gadījumā jādomā par tās izvietojumu, jo mikroservisi var būt darbināti vairākos serveros un jāizvēlas vismazāk noslogoto piemēroto serveri. Vecās instances noņemšanas gadījumā jābeidz jauno pieprasījumu sūtīšanu instancei, un gaidīšanu, kamēr visi uzsāktie pieprasījumi tiek apstrādāti.

3.2.5.2. Risināšanas stratēģija

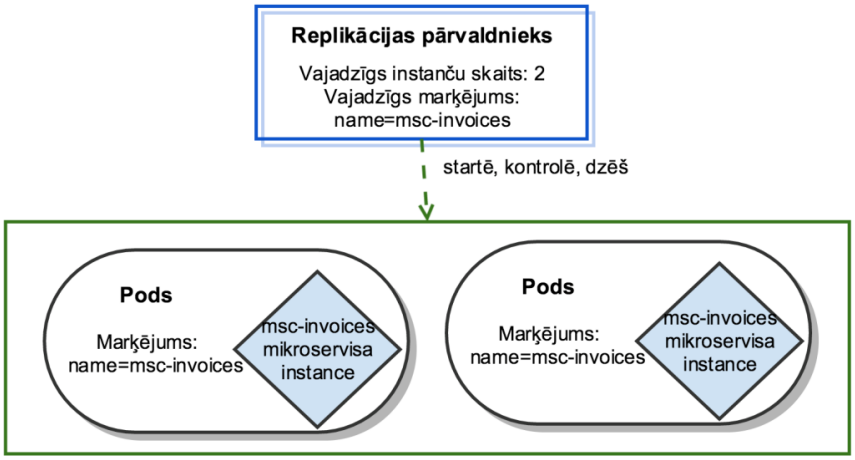
Mērogošanu ir iespējams sadalīt divos veidos:

- Vertikālā mērogošana – palielināt programmatūras instancei pieejamo resursu daudzumu. To ir iespējams sasniegt, atjaunojot servera komponentes – procesoru un atmiņu, vai, ja viena serverī darbojas vairāk par vienu programmu, iedodot programmatūras instancei lielāku resursu daļu, atņemot resursus no citām programmām;
- Horizontālā mērogošana – palielināt programmatūras instanču skaitu. To ir iespējams sasniegt, pievienojot jaunus serverus klasterim, vai, ja vēl ir neizmantotie resursi esošos serveros, vienkārši palaist tur jaunās programmatūras instances.

Pielietojot mikroservisu arhitektūru sistēmas izstrādē, kļūst iespējams efektīvākai horizontālai mērogošanai izmantot funkcionālo dekompozīciju – sadalīt programmatūru mazākās daļās un mērogot tās daļas atsevišķi.

Kubernetes rīkam pastāv vairākas iespējas konfigurēt mikroservisu mērogošanu. Galvenais mērogošanas mehānisms ir replikācijas pārvaldnieks. Pārvaldniekam ir definēta instrukcija, kurus un cik podus ir jāizveido. Pēc podu izveidošanas pārvaldnieks kontrolē, lai vienmēr klasterī būtu palaists definētais podu skaits. Vienas instances sabrukšanas gadījumā

replikācijas pārvaldnieks izveido jaunu. Instances tiek maksimāli izplatītas pa mezgliem, lai, viena mezgla sabrukšanas gadījuma, varbūtība, kā kāda tipa poda vispār nepaliek, būtu minimāla.



3.3. att. **Instanču replikācijas kontrole izmantojot Kubernetes replikācijas pārvaldnieku**

3.2.5.3. Risinājuma piemērs

Replikācijas pārvaldnieks ir veidots uz konfigurācijas datnes pamata, palaišana notiek ar vienu komandu:

kubectl create -f msc-api-web-controller.yaml

kur *msc-api-web-controller.yaml* ir konfigurācijas datne.

Ja pēc instanču palaišanas ir nepieciešams manuāli palielināt instanču skaitu, ir pieejama komanda:

kubectl scale rc msc-api-web-controller --replicas=5

kur *msc-api-web-controller* ir palaistā pārvaldnieka nosaukums un *5* ir cik podu instancēm jābūt palaistiem.

Replikācijas pārvaldnieka konfigurācijas datnes piemērs ir pieejams 6. pielikumā.

Pārvaldniekiem ir iespējams nokonfigurēt automātisku mērogošanu, izmantojot Kubernetes iekļauto Heapster pārraudzības rīku, kas tiek automātiski palaists klastera veidošanas brīdī (par Heapster vairāk informācijas ir pieejams 3.2.6. sadaļā):

kubectl autoscale rc msc-api-web-controller --max=5 --cpu-percent=80

kur *msc-api-web-controller* ir pārvaldnieka nosaukums, *5* ir maksimālais palaisto podu skaits un 80% ir procesora noslogotība podiem no konfigurācijā definētas. Pirms ieslēgt

automātisku mērogošanu, podiem ir jābūt uzstādītai maksimālai procesora resursu izmantošanas robežai.

Mērogošana notiek nemainot mezglu skaitu. Ja serveru klasterim tika pievienots jauns serveris, tajā ir jāstartē Kubernetes palīgprocesu lai tas pārvērstos par Kubernetes mezglu:

kubelet --api_servers=http://123.45.67.8:8080 --v=2 --enable_server --allow-privileged

kube-proxy --master=http://123.45.67.8:8080 --v=2

kur *123.45.67.8* ir galvenās Kubernetes instances IP adrese.

3.2.6. Pārraudzība

3.2.6.1. Apraksts

Jābūt īsta laika detalizētai darbināto mikroservisu un serveru statistikai, lai saņemtu informāciju par sistēmas slodzi un citiem parametriem. Novērošana var notikt gan automātiski, izveidojot kritiskās vērtības dažādām metrikām, piemēram, brīvai vietai serveru atmiņā, un paziņot ieinteresētas personas problēmu gadījumā, gan manuāli, izveidojot administrēšanas paneli ar diagrammām un citām informācijas uztveršanas uzlabošanas veidiem.

3.2.6.2. Risināšanas stratēģija

Par katru palaisto mikroservisu jāzina:

- cik tas aizņem procesora resursu;
- cik tas aizņem atmiņas;
- cik ātri izpildās pieprasījumi;
- cik daudz pieprasījumu nāk mikroservisā;
- vai ir neveiksmīgie pieprasījumi.

Par mikroservisu klasteri jāzina:

- cik katrā mikroservisa instanču ir pieejams;
- cik aizņem pieprasījumi viena tipa mikroservisu instancēm;
- mikroservisu avārijas gadījumu vēsturi.

Tā kā mikroservisi tiek palaisti konteineros, mikroservisa resursu pārraudzība ir vienāda ar konteineru resursu pārraudzību. Docker konteineru pārraudzībai ir pieejams liels rīku kopums. Kubernetes arī piedāvā vairākus palīgrīkus to klastera un podu pārraudzībai.

Izstrādājot apskatīto sistēmu tika nolemts vienā podā izvietot vienu Docker konteineri, tāpēc tika nolemts izmantot rīkus tieši podu pārraudzībai.

Par konteineru resursu pārraudzības risinājumu tika izvēlēta Grafana saskarne ar InfluxDB datu glabātuvi, jo šie rīki ir ļoti plaši konfigurējami un var būt vienkārši palaisti Kubernetes klasterī. Datus no podiem saņem Heapster rīks, kas ir Kubernetes dzimtais risinājums datu savākšanai.

Pieprasījumu statistika tiek savākta ar Netflix Hystrix bojājumpiecietības rīku, kuram ir pieejams administrācijas panelis ar savākto datu vizualizāciju. Mikroservisa ātrdarbības mērīšanai ir iespējams izmantot slodzes testa gadījumus.

Visi rīki tiek instalēti Kubernetes klasterī kā atsevišķi konteineri.

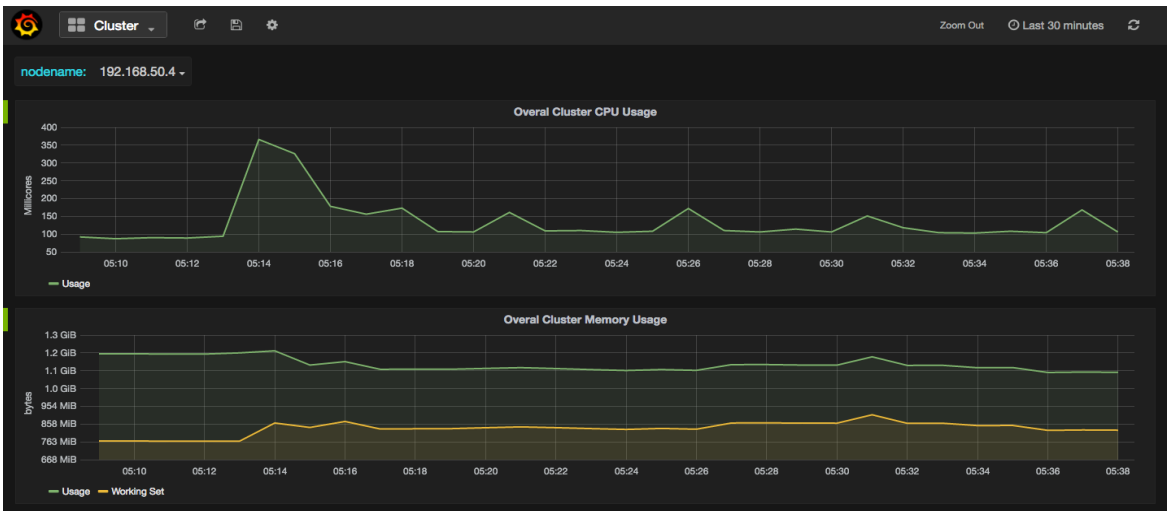
Pašlaik nav pieejama iesaistīto personu apziņošana avāriju gadījumā, bet ir iespējams instalēt citus rīkus, kā Nagios serveru resursu izmantošanas novērošanai, kas atbalsta ziņojumu sūtīšanu, ja serveris tika atslēgts, vai tam nepietiek resursu pieprasījumu apstrādei.

3.2.6.3. Risinājuma piemērs

Pārraudzības rīku iestatīšanas pamācības ir pieejamas 8. (klastera palaišana Amazon EC2 serveros) un 10. (Hystrix konfigurēšana) pielikumos. 3.4. attēlā ir redzama izsaukumu statistika no Company mikroservisa – pašlaik ir pieejami *hystrixCheck* metodes izsaukšanas dati. Metodē ir cita mikroservisa izsaukums, kurš pašlaik nestrādā, tāpēc Hystrix panelī ir redzams "Circuit Open" teksts, kas nozīmē, kā pašlaik neviens Company mikroservisa pieprasījums pie hystrixCheck netiks pildīts. Vairāk par bojājumpiecietības risinājumu ir pieejams 3.3.1. sadaļā.

3.4. att. **Hystrix pārraudzības paneļa saskarne**

3. 5. attēlā ir redzama procesoru un atmiņas resursu izmantošanas statistika Grafana rīkā visās klastera mašīnās par pēdējām 30 minūtēm. InfluxDB ar Heapster palīdzību tiek importēti dažāda tipa datu, kurus Grafana rīkā ir iespējams dināmiski atlasīt un pašiem veidot vajadzīgo statistiku.



3.5. att. **Grafana pārraudzības paneļa saskarne**

3.2.7. Žurnalēšana

3.2.7.1. Apraksts

Monolīta arhitektūrā žurnalēšana notiek triviāli, jo visi notikumi tiek rakstīti vienā plūsmā no viena avota, tāpēc ir iespējams viegli izsekot pieprasījumu apstrādes ceļu.

Mikroservisu arhitektūrā katram mikroservisam ir savi notikumi un jābūt ērtai notikumu reģistrēšanas un pārvaldības funkcionalitātei, kas varētu apvienot, apstrādāt un kategorizēt masīvo datu skaitu. Jābūt iespējai atlasīt konkrēta lietotāja pieprasījumu, kā arī aplūkot lietotāja viena pieprasījuma gaitu – lai uzzinātu, kādi mikroservisi tika pieprasīti, un sašaurināt meklēšanas lauku problēmu radīšanas gadījumā. Notikumu reģistrēšanas mehānismam jābūt integrētam ar pārraudzības sistēmu, kuru dažādos gadījumos būs jāpaziņo.

3.2.7.2. Risināšanas stratēģija

Pašlaik kļūst plaši izmantojama žurnalēšanas pieeja, kad tai tiek izmantoti vairāki instrumenti – katrs savai notikumu reģistrēšanas daļai – notikumu glabāšanai un meklēšanai tiek lietots meklēšanas dzinējs; notikumu vizualizēšanai un analīzei tiek lietotas datu attēlošanas bibliotēkas, notikumu sākotnējai pirmsapstrādei, apvienošanai un pierakstīšanai tiek lietota atsevišķā programmatūra.

Docker konteineru un Kubernetes pārvaldnieka gadījumā par noklusējuma tehnoloģijām tiek uzskatīti Elasticsearch meklēšanas dzinējs, Kibana saskarne un fluentd ziņojumu lasītājs un savienotājs. Darbā apskatītā projektā tika nolemts izmantot šo tehnoloģiju kopumu, jo rīki piedāvā visu žurnalēšanas apskatei vajadzīgo funkcionalitāti.

Viena pieprasījuma notikumu reģistrēšanai tika nolemts katram pieprasījumam izveidot unikālo identifikatoru, kas tiek padots kā parametrs katrā pieprasījumā un ievietots katrā notikumā, un dod iespēju ātri sameklēt pieprasījuma notikumus.

3.2.7.3. Risinājuma piemērs

Elasticsearch, Kibana un fluentd iestatīšanas pamācība ir pieejama 8. pielikumā. Rīki tiek automātiski instalēti kā atsevišķi konteineri Kubernetes klasterī.

Java bāzētiem mikroservisiem notikumu rakstīšana notiek izmantojot log4j bibliotēku:

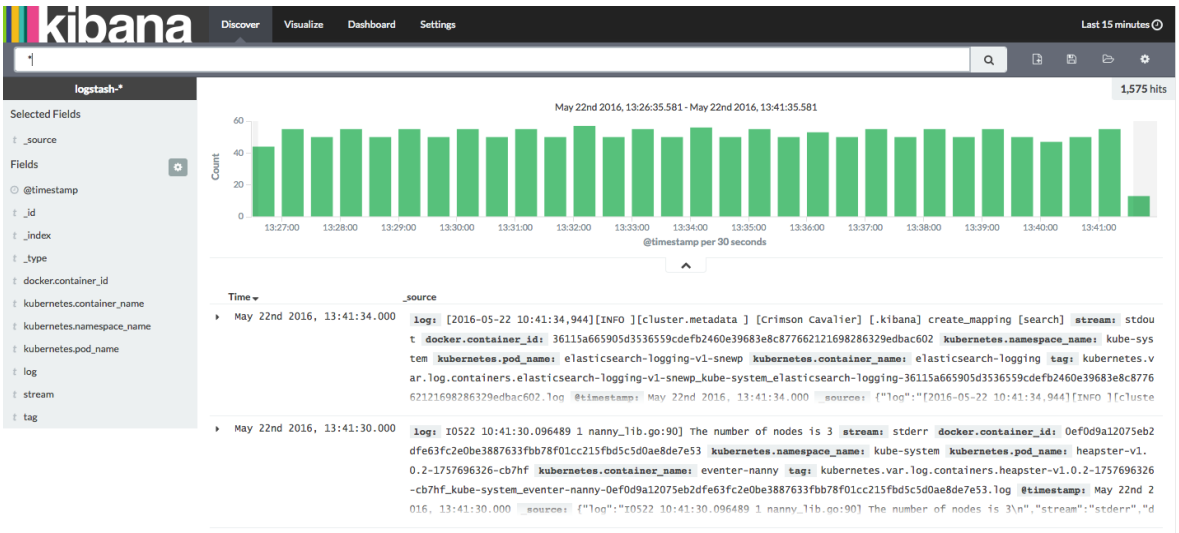
logger.info(requestId + " notikuma ziņojums");

Node.js bāzētiem mikroservisiem notikumu rakstīšana notiek izmantojot intel bibliotēku:

require('intel').info(requestId + " notikuma ziņojums");

kur *requestId* ir pieprasījuma identifikators.

Fluentd rīks automātiski savāc notikumus no visiem konteineriem un padod tos glabāšanai Elasticsearch servisā. Kibana attēlo datus no Elasticsearch vēlamā formātā. Kibana administratora paneli ir iespējams apskatīt 3. 6. attēlā.



3.6. att. **Kibana žurnalēšanas paneļa saskarne**

3.2.8. Drošība

3.2.8.1. Apraksts

Mikroservisu arhitektūras pielietošanas plānošanā radās jautājumi drošības ziņā:

- Kur ir jāglabā un kā ir jānodod privātus konfigurāciju datus mikroservisiem? Pirmkods un visas konfigurācijas datnes tiek glabātas versiju kontroles sistēmā, tāpēc tos var apskatīt ne tikai tie cilvēki, kuriem ir tiesības veikt privāto datu pārvaldību.
- Kā paslēpt privātos mikroservisus no lietotājiem un dot iespēju pieprasīt tikai publisko REST API mikroservisus?

3.2.8.2. Risināšanas stratēģija

Tika nolemts visus datus, kuri ir vajadzīgi piekļuvei pie datu bāzēm un citiem servisiem, padot mikroservisiem kā vides mainīgus. Šī pieeja deva iespēju noradīt visus privātus datus Kubernetes konfigurāciju datnēs.

Kubernetes rīkam ir pieejama ērta funkcionalitāte noslēpumu glabāšanai, Secret, kas dod iespēju konfigurācijas datnēs privāto datu vietā noradīt marķējumu, pēc kura mikroservisa palaišanas laikā Kubernetes savienojas ar glabātuvi un no tās pēc marķējuma saņem vajadzīgus datus.

Kubernetes klasterī visi podi un tajos izvietotie mikroservisi ir pēc noklusējuma paslēpti. Vienīgais veids kā dot piekļuvi pie poda – izveidot ārēja tipa Kubernetes servisu, un tiks publiskots tikai viens ports.

3.2.8.3. Risinājuma piemērs

Privāto datu saglabāšanas piemēru iespējams apskatīt 9. pielikumā. Noslēpumu izmantošana ir pieejama 6. pielikumā.

Kubernetes servisa konfigurēšana ir pieejama 7. pielikumā.

3.3. Mikroservisu implementēšana

3.3.1. Sistēmas testēšana

3.3.1.1. Apraksts

Mikroservisu arhitektūrā sistēmas testēšana ir netriviālā, jo, kopā ar monolīta arhitektūras ierastiem testēšanas veidiem, ir arī jātestē sistēmas bojājumpiecietību un mērogošanas mehānismu, starpservisu sadarbību.

3.3.2.2. Risināšanas stratēģija

Tā kā viens no mikroservisu nozīmīgiem labumiem ir iespēja ātri izstrādāt un izvietot jauno vai pārrakstīt esošo funkcionalitāti, tika nolemts maksimāli izmantot automatizēto testēšanu, jo tas dod iespēju iespaidīgi paātrināt testēšanas stadiju.

Testēšanas aktivitātes iespējams sadalīt divās daļās – funkcionālajā un nefunkcionālajā testēšanā. Pie funkcionālās testēšanas attiecās tieši darījumloģikas pārbaude – to ir iespējams testēt vairākos detalizācijas līmeņos – sākot ar nelieliem programmkoda blokiem un beidzot ar lietotāju darbplūsmu testēšanu. Pie nefunkcionālās testēšanas attiecās sistēmas veiktspējas, bojājumpiecietības, slodzes, mērogošanas, lietojamības testēšana.

Funkcionālās testēšanas detalizācijas līmenis tiek definēts katram mikroservisam atsevišķi, atkarībā no servisa sarežģītības un svarīguma.

Vienīgais obligātais funkcionālo testu veids, kurš tiek izmantots visos servisos, ir vienībtesti. Tie tiek izpildīti automātiski pēc katra mikroservisa programmkoda atjauninājuma, izmantojot nepārtrauktās integrācijas sistēmu.

Nākamais līmenis ir integrācijas (veselā servisa) testi – būtībā tā ir vesela mikroservisa testēšana izmantojot tā API. Integrācijas testos tiek automatizēti pārbaudīta kā servisa funkcionalitāte, tā arī integrācija ar citiem mikroservisiem, no kuriem ir atkarība, un nefunkcionālās testēšanas veidi – veiktspēja (mērot pieprasījuma izpildīšanas laiku), slodze

(sūtot servisam vairākus pieprasījumus vienlaicīgi) un bojājumpiecietība (atslēdzot atkarības un pārbaudot kā serviss uz to situāciju reaģē). Integrācijas testi tiek izpildīti katru dienu un pirms katras programmkoda izvietošanas produkcijas vidē.

Mikroservisu API mainīšanas kontrolēšanai tiek veikta līgumu testēšana – no servisa atkarīgie klienti – citi servisi – paši definē šīm mikroservisam integrācijas līmeņa testus ar aprakstiem, kā viņi to izmanto – kādus veic pieprasījumus un ko vēlas redzēt atbildēs. Šie testi tiek izpildīti kopā ar citiem integrācijas testiem.

Sistēmas testēšana ir vissmagākais un ietilpīgākais testēšanas veids, un ar to tiek pārbaudītas nozīmīgas lietotāju darbplūsmas un lietotāju saskarne.

Kopējā sistēmas bojājumpiecietības, veiktspējas un ātrdarbības testēšana notiek ar tiem pašiem integrācijas testiem, testējot publiski pieejamus REST API mikroservisus.

Mērogošanas testēšana tiek veikta manuāli, izmantojot integrācijas līmeņa slodzes testus – testu izpildes laikā testēšanas inženieris aplūko sistēmas mērogošanas mehānismus izmantojot pārraudzības programmatūru [sk. 3.2.6.].

Pašlaik kļūst populāra pieeja, "kanārijputnu testēšana" [27], kad jaunā funkcionalitāte tiek vispirms piedāvāta nelielai lietotāju kopai. Tika nolemts izmantot šo pieeju jauno mikroservisu versiju testēšanai. Kubernetes rīkam ir iespējams, izmantojot replikāciju pārvaldnieku un servisus, vienkārši izvietot un pēc testēšanas dzēst jaunās mikroservisu versijas.

Kanārijputnu testēšanas pielietošanas gadījumā pastāv problēma, ja mikroservisam notiek izmaiņas datu bāzes līmenī. Piemēram, ja tika nolemts izdzēst vienu kolonnu vai samainīt kolonnas datu tipu, pēc jaunās mikroservisa versijas palaišanas testēšanai pastāv vajadzība atbalstīt gan kanārijputnu instanci, gan vecās instances. Šī problēmas risināšanai ir iespējams pielietot "paralēlo izmaiņu" izstrādes veidni [28]. Kolonnas dzēšanas gadījumā dzēšana notiek pēc testēšanas beigšanas un vecās versijas konteineru izslēgšanas. Kolonnas nosaukuma vai kolonnas datu tipa maiņas gadījumā pirms jaunās mikroservisa versijas palaišanas ir jāizveido kolonnas kopiju ar izmaiņām, kurā automātiski tiks kopēti un, ja ir vajadzīgs, pārveidoti dati no vecās kolonnas. Pēc jaunās mikroservisa versijas palaišanas kanārijuputnu testēšanai, vecās mikroservisu instances izmantos vecās kolonnas. Veco kolonnu dzēšanai jānotiek pēc visu mikroservisa instanču atjaunošanas.

3.3.2.3. Risinājuma piemērs

Katrai populārai programmēšanas valodai eksistē vienībtestēšanas ietvari. Apskatītā sistēmā tiek izmantots JUnit priekš mikroservisiem, uzrakstītiem Java valodā, un QUnit priekš mikroservisiem, uzrakstītiem JavaScript valodā.

Par nepārtrauktās integrācijas sistēmu, kurā tiek palaisti automatizēti testi, tika izvēlēts Jenkins.

Sistēmas saskarnes un darbplūsmu testēšanai tika nolemts izmantot Selenium ietvaru.

Kontraktu, veiktspējas, un integrācijas testēšanai tiek izmantots Cucumber ietvars BDD testu rakstīšanai, un cukes-rest rīks, kas palīdz rakstīt tieši testa gadījumus tieši REST API testēšanai.

Veiktspējas testa piemērs ir pieejams 16. pielikumā. Katram mikroservisam ir definēti veiktspējas kritēriji – cik ātri jāatbild uz pieprasījumiem.

Visiem klientiem, kas izmanto servisu, ir piedāvāta iespēja veidot līgumu testus kā lietošanas piemērus, izmantojot Cucumber ietvaru. Testi tiek palaisti jaunās servisa versijas palaišanas laikā. Ja kāds tests tiek salauzts, par to tiek brīdināts klients, kuram pieder tests. Visi testi tiek glabāti atsevišķajā repozitorijā, katram mikroservisam savā mapē. Līguma testa paraugs ir pieejams 12. pielikumā.

Kanārijpuntu testēšanai tiek izmantoti Kubernetes marķējumi. Ja ir nepieciešamība startēt mikroservisa jauno versiju kopā ar vecajām, jaunā versija tiek palaista ar atsevišķo replikācijas pārvaldnieku. Pieņemsim, kā mikroservisa instances tiek pieprasītas izmantojot Kubernetes servisu ar meklējamo instanču marķējumu “msc-invoices”. Tad, pieņemsim, kā vecajiem mikroservisiem marķējums ir “msc-invoices:v1”. Jaunām mikroservisam marķējums ir “msc-invoices:v2”. Kubernetes serviss sameklēs gan “msc-invoices:v1”, gan “msc-invoices:v2”. Ja ir nepieciešamība, lai pie jaunā mikroservisa piekļūtu 25% pieprasījumu, tad vecās versijas mikroservisiem jābūt palaistiem 3 instancēs, jaunās versijas – 1 instancei.

3.3.2. Datu sadale starp servisiem

3.3.2.1. Apraksts

Jābūt pārdomātam kā starp mikroservisiem ir visefektīvāk realizēt sadalīto datu saņemšanu pieprasījumiem. Pieprasījumu izpilde kļūst lēnāka proporcionāli mikroservisu skaitam, kurus ir vajadzīgs pieprasīt atbildes veidošanai.

Vienādi dati var būt glabāti vairākos mikroservisos – piemēram, var eksistēt mikroserviss, kurš nodarbojas ar statistiku veidošanu – tāpēc jāpārdomā sinhronizēšanu starp tiem.

Mikroservisu sadalītā arhitektūrā nav iespējams veidot transakcijas tādā sapratnē kā monolīta lietotnēs. Mikroservisi darbojas pēc eventuālas saskaņotības principiem – jāskaitās, kā galu galā dati sistēmā tiks sinhronizēti, bet tiešais laiks, kad tas var notikt, nav konstants. Iespējams realizēt sadalīto transakciju mehānismu, bet šajā gadījumā sistēma zaudē bojājumpiecietību.

3.3.2.2. Risināšanas stratēģija

Sadalīto datu efektīvāku un ātrāku saņemšanu un savienošanu tika nolemts sasniegt divos soļos:

- izmantot kešdarbi – katrs mikroserviss ir pats atbildīgs par savu datu glabāšanu kešatmiņā un saglabāto datu atjaunošanu;
- izmantot apvienoto datu glabāšanas mikroservisus – datu atjaunošanai šie mikroservisi parakstās uz datu avota servisu ziņojumiem.

Datu sinhronizēšanai tika nolemts izmantot asinhrono ziņojumu apmaiņu. Servisi tika projektēti tā, lai lietotājam vajadzētu atsūtīt pieprasījumu datu mainīšanai tikai vienā servisā, kurš ir atbildīgs par noteikto datu glabāšanu. Pēc datu mainīšanas tiek atsūtīts ziņojums visiem atkarīgiem mikroservisiem par šo notikumu.

Tā kā ziņojumi ir asinhroni, var paiet laiks, kamēr visi atkarīgi mikroservisi būs atjaunoti, bet tā kā lietotājs var tikai lasīt datus no atkarīgiem mikroservisiem, nebūs problēmu ar starpservisu datu sinhronizācijas konfliktiem.

Vienīgā nopietna ar šo pieeju saistīta problēma ir iespējamība novecojušo datu izmantošanai – kā piemērs, apskatītā projektā rēķina pievienošanas gadījumā Invoice serviss pēc rēķina saglabāšanas sūta asinhrono ziņojumu Stat mikroservisam par rēķinu statistikas datu maiņu, bet ziņojuma sūtīšanas laikā Stat mikroserviss tiek pieprasīts un pieprasītājs saņem novecojušo statistiku. Tika nolemts katram nevienādības gadījumam novērtēt tās negatīvas sekas un, ja situācija nav pieejama, tad šiem darbībām ir jāpieprasa tieši datu glabāšanas servisu.

Tika nolemts izvairīties no sadalītām transakcijām – ja eksistē dati, kurus ir jāatjauno vienā transakcijā, šos datus ir jāievieto vienā mikroservisā.

3.3.2.3. Risinājuma piemērs

Kešoto datu glabāšanai tika nolemts izmantot Redis glabātuvi, kas dod iespēju sinhronizēt kešotos datus starp visiem jebkura mikroservisa instancēm.

Spring Boot ietvaram ir iespējams nokonfigurēt atsevišķo metožu kešdarbi atkarībā no metodē padotiem parametriem. Redis glabātuves iestatīšanai Spring Boot projektiem ir vajadzīgs pievienot spring-boot-starter-redis bibliotēku.

Node.JS mikroservisiem tiek lietota express-redis-cache bibliotēka.

Redis bāzētās kešdarbes iestatīšanas pamācība Node.JS mikroservisiem ir pieejama 13. pielikumā.

Apskatītā sistēmā par piemēru apvienoto datu glabāšanai tika izveidots *CompanyInvoices* mikroserviss, kas ir vajadzīgs datu attēlošanai lietotāju galvenajā panelī. Rēķinu vai kompāniju datu mainīšanas notikumu laikā no *Invoice* un *Company* mikroservisiem tiek sūtīti asinhronie ziņojumi *CompanyInvoices* servisam ar servisam vajadzīgiem datiem.

3.3.3. Asinhronā sazināšanas

3.3.3.1. Apraksts

Asinhronās sazināšanas realizācija ir viena no sarežģītākām problēmām, ar kuriem sastapās izstrādātājs. Tā kā ļoti iespējams, kā pieprasījuma pildīšanai mikroserviss pieprasīs citus mikroservisus, jābūt iespējai izmantot sistēmas resursus kamēr tiek pildīta darbība cita mikroservisā, nevis gaidīt atbildes atnākšanu, un asinhrono ziņojumu apmaiņa dod šo iespēju.

3.3.3.2. Risināšanas stratēģija

Tika nolemts sazināties asinhroni ar:

- servisiem, kuri nav tieši izmantoti lietotāju pieprasījumu pildīšanai (vēstuļu sūtīšanas mikroserviss);
- infrastruktūras servisiem (notikumu reģistrēšanas sistēma);
- datu apvienošanas servisiem (CompanyInvoicess un Stat servisi tiek atjaunoti asinhroni, bet lietotāji pieprasa datus sinhroni).

Visu lietotāju pieprasījumu pildīšanai tika nolemts izmantot sinhrono sazināšanas veidu, jo tas ir ātrāks (nav starpnieka – ziņojumu rīka) un ir iespējams ātri uzzināt, ja pieprasītais

serviss nestrādā (asinhronās sazināšanas gadījumā ir jāpārtrauc gaidīšanu pēc noteiktā laika intervāla), Problēmas rada arī asinhrono ziņojumu dokumentēšana, jo netika atrasts ziņojumu dokumentēšanas standarts ar implementāciju.

3.3.3.3. Risinājuma piemērs

Ziņojumu sūtīšanas un saņemšanas piemērs Spring Boot ietvaram ir pieejams 14. pielikumā.

3.3.4. Bojājumpiecietība

3.3.4.1. Apraksts

Jāpārdomā iespējas realizēt mikroservisus tā, lai atkarīgo mikroservisu bojājumu gadījuma mikroserviss turpinātu darbu, ja atkarīgais mikroserviss nebija būtisks un darbības turpināšana ir iespējama, vai vairākkārtējai pieprasījuma sūtīšanai atkarīgam mikroservisam, cerot, kā pieprasījums tiks galu galā izpildīts.

3.3.4.2. Risināšanas stratēģija

Bojājumpiecietību ir iespējams aplūkot no diviem skatpunktiem:

- Lietotāja puses – ja mikroserviss nav sasniedzams, lietotājam ir jāspēj izmantot citu sistēmas funkcionalitāti vai paņemt datus no kešatmiņas; mikroservisa palaišanas gadījumā lietotājam automātiski jāparada atveseļoto funkcionalitāti;
- Izstrādātāju puses – ja mikroserviss tika salauzts, jāpalaiž jauno instanci; ja mikroserviss ilgi atbild uz pieprasījumiem, jābeidz sūtīt jaunus kamēr tas sāks strādāt ātrāk.

Lietotāja puses bojājumpiecietība ir sasniedzama izmantojot asinhronus nebloķējušus servisu izsaukumus. Saskarne tika projektētā tā, lai lietotājs, pirmo reizi pieprasot sistēmu, saņemtu vismazāk iespējamo vajadzīgo datu skaitu no vismazāk iespējamā mikroservisu skaita. Visi citi pieprasījumi notiek asinhroni. Ja atnāk kļūdas ziņojums uz pieprasījumu, lietotājs tiek brīdināts par sistēmas kļūdu, saskarnes komponente tiek bloķēta un saskarne sāc atkārtoti sūtīt pieprasījumus mikroservisam noteiktā laika intervālā. Kad atnāc atbilde, sistēmas komponente tiek atbloķēta un atjaunota. Jāsaprot, kā, ja serviss beidz strādāt, tas beidz strādāt visiem klientiem un visi klienti sāc sūtīt periodiskus pieprasījumus, tāpēc

pieprasījumu intervālam jābūt maksimāli lielam. Pieprasījumu veikšanas piemēru ir iespējams apskatīt 17. pielikumā.

Izstrādātāju puses bojājumpiecietība ir sasniedzama izmantojot Kubernetes replikācijas pārvaldniekus, kuri instances laušanas gadījumā izvieto jaunus, un izmantojot specializēto risinājumu – Netflix Hystrix – kas kontrolē izsaukumus starp mikroservisiem.

Hystrix darbošanās princips – ja mikroserviss neatbild uz klientu pieprasījumiem, klienti, atsūtot pieprasījumu un kādu noteiktu reižu skaitu nesaņemot pareizo atbildi (Hystrix noklusēti ir 20 pieprasījumi 5 sekundēs) vai nesaņemot atbildi kāda laika intervālā, vispār beidz sūtīt pieprasījumu servisam, aizvietojot atbildi ar tukšo objektu. Šī laikā Hystrix pats ar noteiktu laika intervālu pieprasa problemātisko servisu līdz tas atveseļosies. Kad serviss sāc strādāt normāli, klienti beidz aizvietot atbildi un turpina sūtīt pieprasījumus atveseļotam mikroservisam.

3.3.4.3. Risinājuma piemērs

Hystrix bibliotēkas eksistē gan Spring Boot ietvaram (spring-cloud-starter-hystrix bibliotēka), gan Node.JS bāzētiem projektiem (hystrixjs bibliotēka).

Hystrix iestatīšanas pamācība ir pieejama 10. pielikumā. Bojājumpiecietības testēšana izmantojot Hystrix tīku ir pieejama 11. pielikumā.

REZULTĀTI

Maģistra darbā tika:

1. dots paplašināts priekšstats par mikroservisu arhitektūru salīdzinājumā ar citiem moderniem lietojumprogrammu arhitektūras veidiem;

Nozīmīgāko arhitektūras īpatnību kopsavilkums

	Monolīta	Slāņu	SOA	Mikroservisu
Izstrādātāja zināšanu līmenis	*zems*	*zems*	*vidējs*	*augsts*
Infrastruktūras sarežģītība	*zema*	*zema*	*vidēja*	*augsta*
Modularitāte	*nav*	*zema*	*vidēja*	*augsta*
Mērogošanas optimalitāte	*zema*	*vidēja*	*augsta*	*augsta*
Tehnoloģiju izvēles brīvība	*nav*	*zema*	*zema*	*augsta*
Reaģētspēja uz klientu prasībām	*zema*	*zema*	*vidēja*	*augsta*

2. noformulētas un atrisinātas problēmas, kas radās mikroservisu arhitektūras pielietošanas procesā;

Risinājumu rezultātu kopsavilkums

Problēma	*Risinājuma novērtējums*
Versionēšana	Versionēšanas problēmas tika atrisinātas ar versionēšanas pielietošanas standartu definēšanu, kā arī tika nolemts un tika piedāvāta pieeja maksimāli atteikties no versionēšanas
Izvietošana	Docker konteineru izmantošana palīdz ievērojami atvieglot mikroservisu neatkarīgās izvietošanas procesu

Palaišana	Docker konteineru un Kubernetes pārvaldnieka lietošana dod iespēju palaist mikroservisus vairākās izvietošanas vidēs, dodot pārliecību, kā sistēmas uzvedība visās vidēs būs vienāda
Slodzes balansēšana	Ir pieejami vairāki slodzes balansēšanas rīki, kas var būt palaisti Kubernetes klasterī, bet noklusējuma balansētājs var būt pietiekošs vairākos gadījumos
Mērogošana	Kubernetes automātiskās mērogošanas iespējas ir ierobežotas, bet manuālā mikroservisa mērogošana, kā arī jaunu serveru pievienošana notiek vienkārši
Pārraudzība	Pārraudzībai pieejama programmatūra dod paplašināto mikroservisu darbības statistiku
Žurnalēšana	Aprakstīta žurnalēšanas pieeja, izmantojot Elasticsearch, Kibana un fluentd rīkus, kopā ar pārraudzības risinājumiem dod iespēju ērti veikt sistēmas darbības problēmu izsekošanu
Drošība	Kubernetes pēc noklusējuma palaiž mikroservisus tikai iekšējā tīkla un piedāvā iespēju glabāt konfigurācijas datus šifrētā veidā, kas definētām atbilst drošības prasībām
Testēšana	Tika izveidota testēšanas stratēģija, kurā tika definēti visi mikroservisu testēšanas veidi, katram veidam tika izvēlēti vajadzīgie rīki
Datu sadale	Datu sinhronizēšanas problēma tika atrisināta, definējot datu atjaunošanas stratēģiju un aizliedzot sadalītās transakcijas; pieprasījumu optimizēšanai tika nolemts veidot mikroservisus-datu savienotājus
Asinhronā sazināšanas	Tika nolemts izmantot asinhronus ziņojumus iekšējai datu pārraidei, jo lietotāju pieprasījumu veikšanai svarīga ir ātrdarbība; sazināšanas palīgrīki ir pieejami visām programmēšanas valodām
Bojājumpie-cietība	Bojājumpiecietības implementēšanai tika izstrādāta pieeja, kā apstrādāt katra konkrēta mikroservisa kļūdu gadījumus, un kādā veidā uz šiem notikumiem jāreaģē saskarnei

3. tika daļēji izveidota uz mikroservisu arhitektūras balstīta tīmekļa lietojumprogramma, kurā tika izmēģināti visi darbā aprakstītie problēmu risinājumi.

SECINĀJUMI

Mikroservisu arhitektūra dod vairākas nozīmīgas priekšrocības salīdzinājumā ar citām pielietojamām arhitektūrām – problēmas apgabalu dekompozīciju vairākās mazākās daļās, efektīvāku mērogošanu, ātrāku izmaiņu piegādi – bet ir jāatceras, ka, izstrādājot sistēmu ar mikroservisiem, implementēšanas gaitā paradās nopietni jautājumi, kas varētu krietni palielināt izmaksas, ja tiem nepievērsa uzmanību pirms izstrādes uzsākšanas.

Sistēmas arhitektūras izvēles procesā izstrādātāju komandai ir jāpieņem lēmumu, balstoties gan uz aprakstītiem plusiem, gan mīnusiem, atkarībā no tā, cik sagatavota ir pašreizējā infrastruktūra (vai cik resursu ir iespējams atvēlēt piemērotas infrastruktūras veidošanai), kāds kompetences līmenis komandas dalībniekiem ir sadalīto sistēmu izstrādē, cik sarežģītas un apjomīgas ir sistēmas prasības, kāds ir iespējamais sistēmas dzīvības laiks, cik skaidri ir pazīstama darbības sfēra un kāda ir iespēja sfērai mainīties, kāds ir iedomāts lietotāju skaits.

Pašlaik, analizējot darba ietvaros aprakstītas arhitektūras implementēšanai vajadzīgas darbības, ir iespējams secināt, kā mikroservisu pielietošana var būt noderīga esošiem lieliem projektiem, kuros ir problēmas ar atbalstu pārmērīgas pirmkoda sarežģītības vai apjoma dēļ, un sistēmām, kurās piedalās vairākas izstrādātāju komandas.

Veiksmīgai arhitektūras pielietošanai, izstrādātājiem ir jābūt gataviem un motivētiem būt pilnībā atbildīgiem par izstrādātiem mikroservisiem, jo viņiem ir jāspēj veikt arī mikroservisu palaišanu un pārvaldību.

Mikroservisu arhitektūra ir samērā jauns jēdziens, kas vēl atrodas savas attīstības sākumā – pieejamo pētījumu un veiksmes stāstu nav pietiekami daudz – tāpēc vēl pastāv vajadzība veikt arhitektūras pielietošanas izmēģinājumus dažādās vidēs, vērtējot tas tiesību kļūt par atzīto un efektīvo programmatūras izstrādes pieeju.

IZMANTOTĀ LITERATŪRA UN AVOTI

1 – Todd Hoff, "Deep Lessons From Google And EBay On Building Ecosystems Of Microservices," 1 Dec. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html

2 – Alan Ho, "Microservices at Amazon," 11 Nov. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://apigee.com/about/blog/developer/microservices-amazon

3 – Josh Clemm, "A Brief History of Scaling LinkedIn," 20 Jul. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin

4 –Tony Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design," 19 Feb. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

5 – ANSI/*IEEE 1471-2000, Recommended Practice for Architecture Description of Software-Intensive Systems*, IEEE, 2001

6 – Len Bass, Paul Clements, Rick Kazman, "*Software Architecture in Practice, Second Edition*," Addison Wesley, 2003.

7 – Phil Calçado, "How we ended up with microservices," 8 Sep. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html

8 – Tom Huston, "WHAT IS MICROSERVICES ARCHITECTURE?" [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: https://smartbear.com/learn/api-design/what-are-microservices/

9 – D. L. McIlroy, *"Unix time-sharing system forward,"* Bell System Technical Journal, 1978

10 – Brian Storti, "The actor model in 10 minutes," 9 Jul. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://www.brianstorti.com/the-actor-model/

11 – Hao He, "What Is Service-Oriented Architecture," 30 Sep. 2003 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

12 – Thomas Erl, "*SOA: Principles of Service Design*," Prentice Hall, 2008

13 – Sam Newman, "*Building Microservices*," OReilly Media, 2015.

14 – Martin Fowler, "Microservice Trade-Offs," 01 Jul. 2015 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://martinfowler.com/articles/microservice-trade-offs.html

15 – Benjamin Wootton, "Microservices - Not A Free Lunch!," 8 Apr. 2014 [tiešsaiste]. – [atsauce 28.01.2016]. Pieejams: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

16 – Ozzie Gooen, Tom Ash,"Advantages and Disadvantages of a Monolith Application" [tiešsaiste]. [atsauce 28.01.2016]. Pieejams: https://impact.hackpad.com/Advantages-and-Disadvantages-of-a-Monolith-Application-ZlrQRl3LHCg

17 – Arun Gupta, "Microservices, Monoliths, and NoOps," 30 Mar. 2015 [tiešsaiste]. [atsauce 28.01.2016]. Pieejams: http://blog.arungupta.me/microservices-monoliths-noops/

18 – Mrityunjay Kumar, "Microservices Architecture: What, When, and How," 05 Jan. 2016 [tiešsaiste]. [atsauce 28.01.2016]. Pieejams: https://dzone.com/articles/microservices-architecture-what-when-how

19 – Dirk Krafzig, Karl Banke, Dirk Slama, "*Enterprise SOA: Service-Oriented Architecture Best Practices,"* Pearson Education, 2004

20 – Chris Richardson, "Microservices: Decomposing Applications for Deployability and Scalability," 25 May 2014 [tiešsaiste]. [atsauce 28.01.2016]. Pieejams: http://www.infoq.com/articles/microservices-intro

21 - Arun Gupta, "Getting Started with Microservices" [tiešsaiste]. [atsauce 28.01.2016]. Pieejams: https://dzone.com/refcardz/getting-started-with-microservices

22 - Sam Newman, "Pattern: Backends for frontends," 18 Nov. 2015 [tiešsaiste]. [atsauce 07.05.2016]. Pieejams: http://samnewman.io/patterns/architectural/bff/

23 - Wojciech Tyczynski, "1000 nodes and beyond: updates to Kubernetes performance and scalability in 1.2," 28 Mar. 2016 [tiešsaiste]. [atsauce 07.05.2016]. Pieejams: http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-12.html

24 - Brandon Byars, "Enterprise Integration Using REST," 18 November 2013 [tiešsaiste]. [atsauce 07.05.2016]. Pieejams: http://martinfowler.com/articles/enterpriseREST.html#versioning

25 - Martin Fowler, "TolerantReader," 9 May 2011 [tiešsaiste]. [atsauce 07.05.2016]. Pieejams: http://martinfowler.com/bliki/TolerantReader.html

26 - Chris Richardson, "Service Discovery in a Microservices Architecture," 12 Oct. 2015 [tiešsaiste]. [atsauce 07.05.2016]. Pieejams: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

27 - Danilo Sato, "CanaryRelease," 25 Jun. 2014 [tiešsaiste]. [atsauce 07.05.2016]. Pieejams: http://martinfowler.com/bliki/CanaryRelease.html

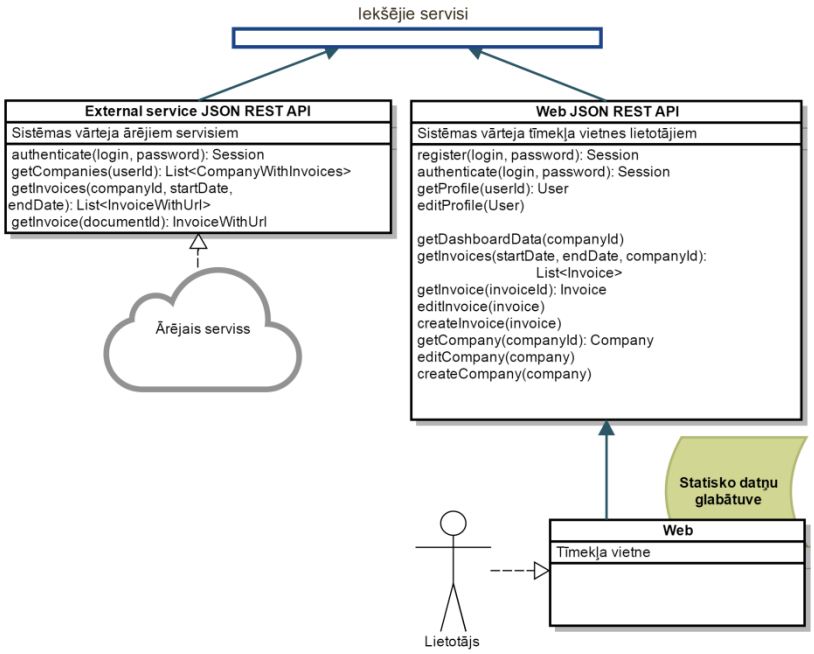
28 - Danilo Sato, "ParallelChange," 13 May 2014 [tiešsaiste]. [atsauce 21.05.2016]. Pieejams: http://martinfowler.com/bliki/ParallelChange.html

PIELIKUMI

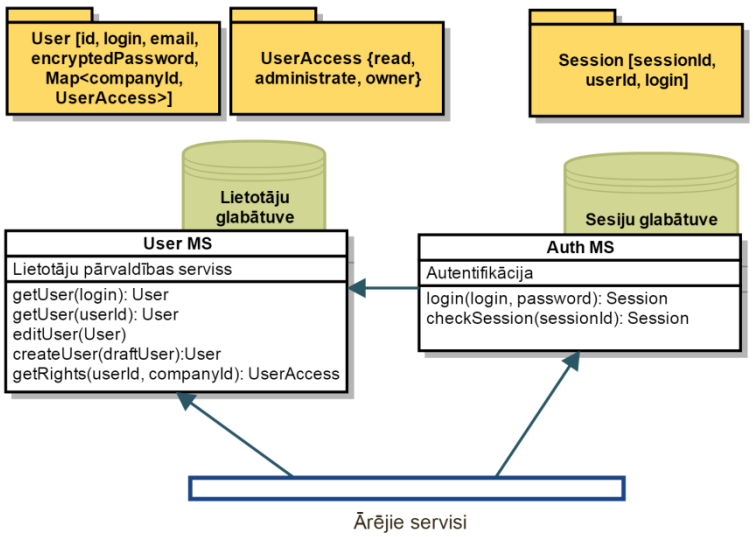
1. pielikums

Detalizēts izstrādātas sistēmas projektējums

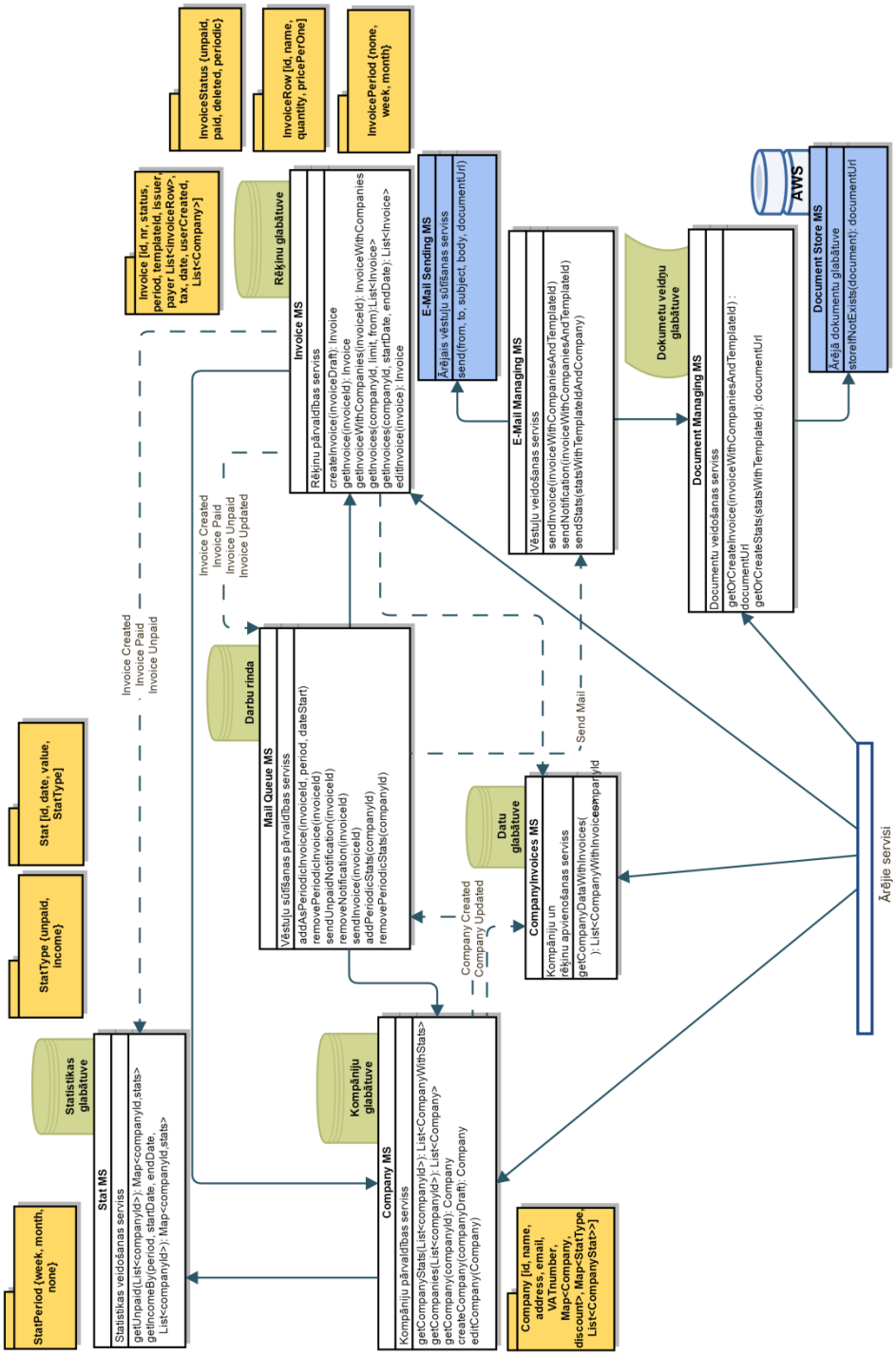
Pirmajā attēlā ir apskatāmi ārējie mikroservisi, kuri veic pieprasījumus visādiem iekšējiem mikroservisiem, paslēptiem no ārējās pasaules.



Otrajā attēlā ir redzami ar lietotāju pārvaldību saistīti iekšējie mikroservisi.



Trešajā attēlā ir redzami visi pārējie mikroservisi, kas realizē tieši darījumloģiku.



2. pielikums

API rokasgrāmata

Lai samazinātu grūtības klientu pārejai uz jaunām servisa versijām, ir definēti saderības norādījumi jauno servisa versiju izstrādātājiem:

1. Ir jāizvairās no servisa metodes nosaukuma vai parametru nosaukumu mainīšanas;
2. Servisa metodes nosaukuma vai parametru nosaukumu mainīšanas gadījumā vecai metodes versijai ir jābūt pieejamai izmantošanai ne mazāk par mēnesi pēc jaunās versijas izvietošanas;
3. Servisa metodes dzēšanas gadījumā klientam jābūt atgriezts pieprasījuma noraidījums;
4. Ja servisa metodei ir jāpievieno jaunu obligāto parametru, ir jāizveido jauno metodi;
5. Atbildes obligāto vērtību nosaukumus vai izvietojumu nedrīkst mainīt. Vērtība skaitās par obligāto, ja tā ir izmantota līgumu testa gadījumos;
6. Jebkuras līguma izmaiņas ir iespējams veikt tikai gadījuma, ja visi līguma testi strādā korekti. Ja līgumtests izkrīt, jākontaktējas ar testa īpašniekiem un jāgaida kamēr viņi pielāgosies jaunam līgumam un salabos testu.

Saderības norādījumi klientiem:

1. Klientam jāpieņem, kā servisa atbildē var parādīties papildus parametri vai izzust neobligātie parametri;
2. Klientam ir pienākums veidot līgumtestus izmantojamam servisam.

Pašlaik nav pieejams ērts mehānisms atkarīgo servisu apziņošanai par jauno mikroservisa versiju radīšanu, tāpēc:

- Atkarīgo servisu un par tiem atbildīgo personu sarakstam jābūt pieejamam kompānijas iekšējā sistēmā;
- Jauno versiju radīšanas notikumiem jābūt paziņotiem visiem klientiem pa e-pastu nedēļu pirms versijas izvietošanas;
- Klientiem jāpaziņo pa e-pastu par izmantotās versijas mainīšanu, jānorāda izmaiņu sarakstu un saiti ar jaunās versijas dokumentāciju.

3. pielikums

Versionēšanas rokasgrāmata

Šajā dokumentā tiek aprakstīts, kā jānotiek piekļuvei pie atsevišķām servisa versijām, kādam jābūt veco versiju atbalsta periodam un kādā veida klientiem jāpaziņo par jauno versiju radīšanu.

Darbā apskatītā sistēmā visā sinhronā sazināšanas starp servisiem notiek izmantojot REST pieeju pa HTTP slāni, tāpēc katram servisam ir jābūt unikālam URL. Apskatītai sistēmai tika izvēlēta visplašāk un vieglāk izmantojamā versionēšanas pieeja – norādīt vajadzīgo servisa versiju URL segmentā:

mikroserviss/v2/companies/1

kur *v2* ir API versija, *companies* ir servisa metode un *1* ir obligātais pieprasījuma parametrs. Versijas noradīšana klientam ir obligāta. Versiju numurēšana notiek pēc Semantic Versioning 2.0.0 standarta.

Katras servisa versijas metožu lietošanas statistikai jābūt savāktai ar žurnalēšanas sistēmu – katram REST API metodes izsaukšanas notikumam ir jābūt saglabātam. Ar servisa lietošanas statistikas palīdzību jābūt iespējams kontrolēt veco versiju izmantošanas datus, un uz to pamata secināt, vai tos ir iespējams atslēgt. Ir jāpārtrauc vecās versijas atbalstu un jāizdzēš atbilstošo programmkodu tad, kad versijas pieprasījumu skaits ir mazāk kā 10% no jaunās versijas pieprasījumu skaita, bet ne agrāk par pusgadu no nākamās versijas izvietošanas datuma. Mēnesi pirms versijas atslēgšanas, katrā pieprasījuma atbildē laukā *error* jāsūta brīdinājumu par versijas atslēgšanu.

Lai samazinātu grūtības klientu pārejai uz jauno servisa versiju, tika definēti norādījumi jauno servisa versiju izstrādātājiem:

- nedrīkst mainīt esošo pieprasījuma parametru nosaukumus un datu tipu;
- veco pieprasījuma parametru saņemšanas gadījumā tos ir jāignorē;
- nedrīkst mainīt esošo pieprasījumu metožu nosaukumus, ja netika mainīta to būtība.

API dokumentācijas ģenerēšanas iestatīšana Spring Boot ietvarā

Mikroservisiem, kuri ir izstrādāti uz Spring Boot ietvara, Swagger dokumentācijas ģenerēšanai ir vajadzīgs:

1. Iekļaut trīs Springfox bibliotēkas Maven konfigurācijas datnē *pom.xml*. Pirmā bibliotēka savāc dokumentācijai vajadzīgus datus no pieejamiem API klasēm. Otrā bibliotēka ģenerē ērtu saskarni. Trešā bibliotēka eksportē dokumentāciju ASCII vai Markdown formātā. Jābūt arī pievienotas *org.springframework:spring-test, junit:junit* un *org.apache.commons:commons-lang3* palaišanai vajadzīgas atkarības.

```
<dependency>
  <groupId>io.springfox</groupId><artifactId>springfox-swagger2</artifactId>
  <version>2.2.2</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId><artifactId>springfox-swagger-ui</artifactId>
  <version>2.2.2</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId><artifactId>springfox-staticdocs</artifactId>
  <version>2.2.2</version>
  <scope>test</scope>
</dependency>
```

2. Izveidot Spring konfigurēšanas klasi, kurā ir iespējams noradīt vairākus saskarnes ģenerēšanas parametrus, piemēram, API klašu izvietojumu, ja ir vajadzīgs ģenerēt dokumentāciju tikai konkrētiem API klasēm. Saskarne ir ģenerēta mikroservisa palaišanas gadījumā un ir pieejama pēc adreses *mikroservisa_adrese/swagger-ui.html*.

```
@Configuration @EnableSwagger2
public class SwaggerConfig {
  @Bean public Docket api() {
    return new Docket(DocumentationType.SWAGGER_2)
        .select().apis(RequestHandlerSelectors.any())
```

```
            .paths(PathSelectors.any()).build();
}}
```

3. Ja ir vajadzīgs eksportēt dokumentāciju, jāizveido testēšanas klasi ar eksportēšanas konfigurāciju. Dokumentācija tiks eksportēta testu palaišanas laikā (jāpalaiž Maven komandu *mvn test*). Piemērā tiks ģenerēti dati Markdown formātā, un eksportēti *../msc-swagger-docs/msc-companies/* direktorijā:

```
@WebAppConfiguration @RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = DefaultApplication.class, loader =
SpringApplicationContextLoader.class)
public class Swagger2MarkupTest {
   @Autowired private WebApplicationContext context;
   private MockMvc mockMvc;
   @Before public void setUp() {
      this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context).build();
   }
   @Test public void convertSwaggerToMarkdown() throws Exception {
      this.mockMvc.perform(get("/v2/api-docs")
            .accept(MediaType.APPLICATION_JSON))
            .andDo(Swagger2MarkupResultHandler.outputDirectory("../msc-swagger-docs/msc-
companies/")
                  .withMarkupLanguage(MarkupLanguage.MARKDOWN).build())
            .andExpect(status().isOk());
}}
```

4. Izveidot REST API klases, izmantojot standarta Spring REST anotācijas:

```
@RestController @RequestMapping({"/v1", "/"})
public class InvoiceControllerV1 {
   @Autowired InvoiceService invoiceService;
   @RequestMapping(path = "/invoice/{invoiceId}/companies", method = RequestMethod.GET)
   public Invoice getInvoiceWithCompanies(@PathVariable(value = "invoiceId") String invoiceId) {
      return invoiceService.getWithCompanies(invoiceId);
}}
```

Dockerfile datņu paraugi Java un Node.JS bāzētiem mikroservisiem

Visas Dockerfile datnes ir glabātas mikroservisa saknes direktorijā.

Dockerfile piemērs mikroservisam, kurš tika izstrādāts uz Node.JS dzinēja pamata:

```
FROM node:slim # publiski pieejamā pamatattēla nosaukums
RUN mkdir -p /usr/src/msc-api-web # attēlā ir jāizveido msc-api-web mapi
WORKDIR /usr/src/msc-api-web # cd komandas analogs
COPY . /usr/src/msc-api-web # jākopē mikroservisa pirmkodu direktorijā
CMD ["npm", "start"] # pēc konteinera palaišanas ir jāizpilda npm start komandu
```

Dockerfile piemērs Java mikroservisam:

```
FROM frolvlad/alpine-oraclejdk8:slim
VOLUME /tmp
ADD target/msc-companies-1.0-SNAPSHOT.jar app.jar # tiek ievietots jau nokompilēts mikroserviss
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"] # palaišana
```

Kubernetes Replication Controller datnes paraugs

Piemērā tiek izveidots replikācijas pārvaldnieks ar nosaukumu msc-api-web-controller, kas pēc palaišanas izveidos 2 *msc-api-web-pod* podus (katrā ir viena *msc-api-web* mikroservisa instance):

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: msc-api-web-controller
spec:
  replicas: 2
  selector:
    name: msc-api-web-pod # marķējums, pēc kura tiks atrasti podi kurus pārvaldīt
  template:
    metadata:
      name: msc-api-web-pod
      labels:
        name: msc-api-web-pod # katrā izveidota poda marķējums
    spec:
      containers:
        - name: msc-api-web
          image: dlouchansky/msc-api-web # attēls tiek ielādēts no Docker Hub
          env: # vides mainīgie
            -
              name: "STATS_SERVICE_ADDRESS"
              value: "stats-service" # Stat mikroservisa adrese izmantojot Kubernetes servisus
            -
              name: "SERVICE_PORT"
              value: "8008" # Adrese, no kura mikroserviss saņems pieprasījumus
            -
              name: "REDIS_HOST"
              valueFrom: # noslēpuma ievietošanas paraugs
                secretKeyRef:
                  name: msc-api-web-redis # noslēpuma nosaukums
                  key: redishost # viena no noslēpuma vērtībām
            -
```

```
    name: "REDIS_PORT"
    valueFrom:
      secretKeyRef:
        name: msc-api-web-redis
        key: redisport
  -
    name: "REDIS_EXPIRE_SEC"
    value: "60"
resources: # mērogošanai vajadzīgie parametri
  limits: # cik maksimāli pods var tērēt resursus
    cpu: 300m # ne vairāk par 30% no pieejamiem procesora resursiem
    memory: 200Mi # ne vairāk par 200 mebibaitiem
  requests: #
    cpu: 200m
    memory: 100Mi
ports:
  - containerPort: 8008 # ports, pēc kura ir pieejams konteineris
```

Kubernetes Service datnes paraugs

Pieņemsim, kā Kubernetes klasterī eksistē serveris ar IP adresi 123.45.67.8. Piemērā nokonfigurētais Kubernetes serviss novirza pieprasījumus no tīmekļa uz podiem, kuru marķējums ir *msc-api-web-pod*, kad tiek pieprasīta adrese 123.45.67.8:30766. Pieprasījums tiek novirzīts no ārēja 30766 porta uz iekšējo 8008 portu, tāpēc podiem jāklausās portu 8008.

Lai šī pieeja strādātu, serverim ir jāatver portu 30766 pieprasījumiem no tīmekļa. No Kubernetes klastera iekšpuses, servisu ir iespējams pieprasīt pēc adreses *http://msc-api-web-service/*. Konfigurācija ir ievietota *msc-api-web-service.yaml* datnē:

```
apiVersion: v1
kind: Service
metadata:
  name: msc-api-web-service
spec:
  type: NodePort
  ports:
    - port: 8008
      nodePort: 30766
  selector:
    name: msc-api-web-pod
```

8. pielikums

Kubernetes palaišanas uz Amazon Web Services pamācība

Lai instalētu un palaistu Kubernetes klasteri kopā ar pārraudzības rīkiem Grafana, Heapster, InfluxDB un žurnalēšanas rīkiem Elasticsearch, Kibana un fluentd pie virtuālo serveru nodrošinātāja AWS EC2, ir vajadzīgs:

1. izveidot lietotāja kontu Amazon Web Services tīmekļa servisā;
2. ieiet AWS administratoru panelī un pievienot speciālo IAM lietotāju, kurš ir iekļauts lietotāju grupā ar iespēju administrēt EC2 instances (AdministratorAccess loma), un saglabāt izveidotā lietotāja piekļuves datus (Access Key Id, Secret Access Key);
3. instalēt savā datorā AWS CLI, lai būtu iespējams no komandrindas pārvaldīt EC2 instanču administrēšanu:
   curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"
   unzip awscli-bundle.zip
   sudo ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws
4. nokonfigurēt instalēto AWS CLI ar reģistrētā IAM lietotāja piekļuves datiem:
   aws configure
5. nokonfigurēt Kubernetes klastera palaišanas iestātījumus:
   export KUBE_AWS_ZONE=eu-west-1c # kurā serveru zonā izvietot klasteri
   export NUM_NODES=4 # cik serveru palaist
   export MASTER_SIZE=m3.medium # galvenā Kubernetes mezgla izmērs
   export NODE_SIZE=m3.medium # citu Kubernetes mezglu izmērs
6. palaist Kubernetes klastera izvietošanas komandu:
   KUBERNETES_PROVIDER=aws; curl -sS https://get.k8s.io | bash

Pēc komandas palaišanas, tiks izveidoti 4 EC2 serveri un tajos tiks instalēts Kubernetes klasteris. Pēc serveru palaišanas terminālā tiks paradītas saites, pēc kurām ir iespējams piekļūt instalētiem rīkiem.

Kubernetes Secret lietošanas paraugs

Lai izvietotu noslēpumu Kubernetes klasterī, visērtāk izmantot konfigurācijas datni. Noslēpuma saglabāšana notiek trīs soļos. Piemērā tiek ievietoti Redis glabātuves piekļuves adrese un ports. Sākuma dati:

adrese: redis-service

ports: 6379

Izvietošanas soļi:

1. Nokodēt vērtības ar base64 šifrēšanas algoritmu. Ir pieejama komanda:

   echo "redis-service" | base64

   Rezultāts: cmVkaXMtc2VydmljZQo=

2. Izveidot konfigurācijas datni, piemēram *redis-secret.yaml*, un noslēpuma nosaukumu *msc-api-web-redis*:

```
apiVersion: v1
kind: Secret
metadata:
  name: msc-api-web-redis
type: Opaque
data:
  redishost: cmVkaXMtc2VydmljZQo=
  redisport: NjM3OQo=
```

3. Ievietot noslēpumu Kubernetes klasterī:

   kubectl create -f ./redis-secret.yaml

Noslēpuma lietošanu ir iespējams apskatīt 6. pielikumā.

Hystrix iestatīšanas pamācība

Hystrix rīka iestatīšana ietver 3 soļus:

1. Administratora paneļa palaišana:

   Administratora paneli ir iespējams palaist kā atsevišķu Docker konteineri. Kubernetes konfigurācijas datne, kurā tiek izveidots pods un serviss, un pēc palaišanas administratora panelis būs pieejams, pieprasot 30771 servera portu:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: hystrix-dashb-service
  name: hystrix-dash-service
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 30771
  selector:
    name: hystrix-dash-pod
--
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: hystrix-dash-pod
  name: hystrix-dash-pod
spec:
  containers:
    -
      image: arthurtsang/docker-hystrix-dashboard #viens no brīvi pieejamiem attēliem
      name: hystrix-dash
      ports:
        -
          containerPort: 8080
```

2. Hystrix palaišana mikroservisā:

   Lai administratora panelis saņemtu datus par mikroservisā veiktiem pieprasījumiem, mikroservisam ir jāeksportē atjaunojamo stāvokļa datni. Spring Boot mikroservisam datne pēc iestatīšanas ir pieejama pēc *http://mikroservisa_adrese/turbine.stream* saites. Maven atkarības:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-hystrix</artifactId>
   <version>1.0.0.RELEASE</version>
</dependency>
```

   Jāpievieno anotāciju *@EnableCircuitBreaker* programmas palaišanas klasei, lai Hystrix bibliotēka sāktu pārraudzīt pieprasījumus. Palaišanas klasi var atrast pēc *@SpringBootApplication* anotācijas.

   Metodei, kuru Hystrix rīkam ir jāpārraudzē, arī ir jāpievieno anotāciju *@HystrixCommand* ar noradītu metodes nosaukumu, kurš tiks palaists pieprasījuma kļūdu gadījumā. Abu metožu parametriem un atgrieztai vērtībai ir jābūt vienādiem:

```
@HystrixCommand(fallbackMethod = "hystrixServiceOffline")
public String hystrixCheck() {
   RestTemplate restTemplate = new RestTemplate();
   return restTemplate.getForObject("http://192.168.50.4:30777/hystrixTest", String.class);
}
String hystrixServiceOffline() {
   return "error";
}
```

3. Ienākt administratoru panelī un ieviest pārraudzīta mikroservisa stāvokļa datnes adresi.

Bojājumpiecietības testēšana

Hystrix rīka darbības testēšanai tika izmantoti divi mikroservisi. Otrajām no pirmā mikroservisa tika veikti divi pieprasījumu kopumi. Vispirms tika atsūtīti 10 pieprasījumi ar 0.1 sekundes intervālu. Pēc tam otrais serviss tika atslēgts, un no pirmā servisa tika atsūtīti 50 pieprasījumi ar 0.1 sekundes intervālu.

Pirmajā mikroservisā tika žurnalētas visas pieprasījuma atbildes. Otrajā mikroservisā tika žurnalēts no pirmā mikroservisa pienākto pieprasījumu skaits. Bija sagaidīts, kā kopumā pirmajā mikroservisā būs žurnalēti visu 60 pieprasījumu rezultāti (10 – ar otrā mikroservisa datiem, 50 – ar iestatītiem kļūdu gadījuma datiem), bet otrajā mikroservisā būs žurnalēts krietni mazākais veikto pieprasījumu skaits.

Rezultātā pirmajā mikroservisā tika žurnalēti 60 pieprasījumu rezultāti, bet otrajā mikroservisā tika žurnalēts, kā pienāca 10 korekto pieprasījumu un 13 nekorekto pieprasījumu, kas atbilst sagaidītam.

Pirmā mikroservisa pieprasījuma pirmkods (Java valoda):

```
public Long testHystrix() throws InterruptedException {
    // serviss strādā
    List<String> responses = new ArrayList<>();
    for (long i = 0; i < 10; i++) { // 10 pieprasījumi
        Thread.sleep(100); // 0.1 sekundes instervāls
        String response = statsService.hystrixCheck(); // iepriekšējā pielikumā ievietotas metodes izsaukšana
        responses.add(response);
    }
    System.out.println("Jābūt visiem ok: " + responses);
    // serviss nestrādā
    System.out.println("set error: " + setError());
    responses = new ArrayList<>();
    for (long i = 0; i < 50; i++) {
        Thread.sleep(100);
        String response = statsService.hystrixCheck();
        responses.add(response);
    }
    System.out.println("Jābūt visiem error: " + responses);
    return new Integer(responses.size()).longValue();
```

```
 }
 public String setError() {
    RestTemplate restTemplate = new RestTemplate();
    return restTemplate.getForObject("http://192.168.50.4:30777/setError", String.class);
 }
```

Otrā mikroservisa pieprasījumu apstrādes pirmkods (Node.JS dzinējs):

```
var requestCount = 0;
var hystrixError = false;
server.get('/hystrixTest', function (req, res, next) {
     requestCount++;
     if (!hystrixError) {
       console.log((new Date().toUTCString()) + " hystrix ok: " + requestCount);
       res.send("ok");
       return next();
     } else {
       console.log((new Date().toUTCString()) + " hystrix not ok: " + requestCount);
       return next(new restify.InternalServerError("Hystrix text"));
 }}
server.get('/setError', function (req, res, next) { // kļūdas iestātīšana
     if (hystrixError) hystrixError = false;
     else hystrixError = true;
     requestCount = 0;
     res.send(hystrixError);
     return next();
   },
```

Līguma testa paraugs

Testa gadījumā tiek ir pārbaudīts, kā vajadzīgie lauki ir pieejami, pieprasot šī rēķina datus no Invoice mikroservisa. Šī testa darbībai ir vajadzīgs palaist Invoice mikroservisu un atsevišķo mikroservisu ar līguma testa gadījumiem, kā arī testēšanas datu bāzi, kurā glabājas testēšanai izmantojama rēķina informācija. Testa gadījums tiek izpildīts Invoice mikroservisa izvietošanas laikā.

```
Feature: It is able to get Invoice records with required fields
  Scenario: Should get Invoice object data
    When the client performs GET request on http://localhost:8080/invoice/
    Then status code is 200
    And response contains property "id" with value other than "2000"
    And response contains property "number" with value "Rekins-5"
    And response contains property "templateId" with value "3"
    And response contains property "createdDate" of type "long"
```

Redis kešdarbes iestatīšanas pamācība Node.JS mikroservisam

Redis kešdarbei ir lietota express-redis-cache bibliotēka. Vispirms ir jānorāda Redis glabātuves adresi:

```
var cache = require('express-redis-cache')({
  host: 'redis-service', port: 6379
});
```

Piemērā no Redis tiek pieprasīta kešota informācija par kompāniju. Ja kompānijas dati nav kešoti, tiek pieprasīts Company mikroserviss un to atbildes dati tiek saglabāti atmiņā un atsūtīti klientam.

```
var companyId = req.params.id;
cache.get('company-' + companyId, function (error, entries) {
  // pieprasam datus no Redisa pēc atslēgas "dashboard"
  if (error || !entries || !entries.length) { // ja Redis kešatmiņā nekā nav
    // pieprasam datus no vajadzīgā mikroservisa
    ccompanyService.get('company/' + companyId, function (err, serviceRequest, serviceResponse, obj) {
      if (err) { // ja Company mikroserviss neatbildēja, sūtam kļūdu lietotājam
        res.writeHead(500, responseType);
        res.write(serviceResponse);
        return next();
      } else {
        // ja viss ok, saglabājam redisā saņemtus datus, glabāsim tos 60 sekundes
        cache.add('company-' + companyId,
                  JSON.stringify(serviceResponse),
                  {type: 'json', expire: 60},
                  function (error, added) {
          res.send(serviceResponse);
          return next();
        });
      }
    });
  } else { // ja redis kešatmiņā ir saglabāti dati
    res.send(entries[0].body); // sūtam saglabātus datus
    return next();
  }
});
```

14. pielikums

Asinhrono ziņojumu sūtīšana Spring Boot ietvarā

Apskatītās sistēmas ziņojumu apmaiņas rīks ir RabbitMQ. Ziņojumi tiek sūtīti un saglabāti iepriekš definētā RabbitMQ ziņojumu rindā. Asinhronai ziņojumu sūtīšanai mikroservisiem uz Spring Boot ietvara pamata tiek izmantota Spring AMQP bibliotēka, kuru ir jāinstalē izmantojot Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Pirms implementēt ziņojumu sūtīšanas un apstrādes loģiku, ir jānokonfigurē autentifikācijas datus. Piemērā RabbitMQ ziņojumu sistēmas adrese, lietotājs un parole tiek paņemti no vides mainīgiem RABBIT_MQ_ADDRESS, RABBIT_MQ_USER, RABBIT_MQ_PASSWORD, kuri ir ievadīti replikācijas pārvaldnieka konfigurācijas datnē:

```
@Configuration public class RabbitMqConfig {
   @Value("${RABBIT_MQ_ADDRESS}")
   private String RABBIT_MQ_ADDRESS;
   @Value("${RABBIT_MQ_USER}")
   private String RABBIT_MQ_USER;
   @Value("${RABBIT_MQ_PASSWORD}")
   private String RABBIT_MQ_PASSWORD;
   @Bean public ConnectionFactory connectionFactory() {
      CachingConnectionFactory connectionFactory = new
CachingConnectionFactory(RABBIT_MQ_ADDRESS);
      connectionFactory.setUsername(RABBIT_MQ_USER);
      connectionFactory.setPassword(RABBIT_MQ_PASSWORD);
      return connectionFactory;
   }
   @Bean public MessageConverter jsonMessageConverter() {
      return new Jackson2JsonMessageConverter(); // kādā formātā sūtīt un saņemt ziņojumus
}}
```

Ziņojumu saņemšanai vai sūtīšanai ir jāparakstās uz konkrēto ziņojumu rindu. Piemērā notiek parakstīšana uz *company.invoice.queue* ziņojumu rindu kā ziņojumu saņemšanai, tā arī sūtīšanai:

```
@Configuration public class CompanyInvoiceQueueConfig extends RabbitMqConfig {
```

```
    public final String companyInvoiceQueue = "company.invoice.queue";
    @Autowired private QueueConsumer queueConsumer;
    @Bean public RabbitTemplate rabbitTemplate() { // metode tiek izmantota ziņojumu sūtīšanai
        RabbitTemplate template = new RabbitTemplate(connectionFactory());
        template.setRoutingKey(this.companyInvoiceQueue);
        template.setQueue(this.companyInvoiceQueue);
        template.setMessageConverter(jsonMessageConverter());
        return template;
    }
    @Bean public SimpleMessageListenerContainer listenerContainer() { // parakstīšanas uz ziņojumiem
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        container.setConnectionFactory(connectionFactory());
        container.setQueueNames(companyInvoiceQueue);
        container.setMessageListener(messageListenerAdapter());
        return container;
    }
    @Bean public MessageListenerAdapter messageListenerAdapter() {
        return new MessageListenerAdapter(queueConsumer, jsonMessageConverter());
}}
```

Ziņojumu saņemšanas klase:

```
@Component public class QueueConsumer {
    public void handleMessage(CompanyDataUpdatedEvent scrapingResultMessage) {
        System.out.println("Message received: " + scrapingResultMessage.getMsg());
 }}
```

Ziņojuma sūtīšanas piemērs:

```
@Component
public class QueueProducer {
    @Autowired  private CompanyInvoiceQueueConfig cfg;
    public void sendNewTask() {
        CompanyDataUpdatedEvent companyDataUpdatedEvent = new CompanyDataUpdatedEvent();
        companyDataUpdatedEvent.setMsg("MessageText");
        companyInvoiceQueueConfig
                .rabbitTemplate()
                .convertAndSend(cfg.companyInvoiceQueue, companyDataUpdatedEvent);
 }}
```

15. pielikums

Docker konteineru atmiņas lietošana

Docker konteineru operatīvās atmiņas prasīguma mērīšanai tika izmantots free rīks, kas ir instalēts visās Linux operētājsistēmās un piedāvā informāciju par pašlaik pieejamo un aizņemto operētājsistēmai atmiņu. To ir iespējams palaist ar komandu:

watch -n 5 free –m

Rezultāta piemērs:

	total	*used*	*free*
Mem	1497	1259	238
-/+ buffers/cache		719	778

Pirmajā rindā tiek radīts, cik no pieejamās atmiņas operētājsistēma uzreiz var izmantot lietojumprogrammu palaišanai. Raidījumā operētājsistēma ir rezervējusi 1259 MB no 1497 MB, 238 MB var tikt norezervēti nepieciešamības gadījumā.

Otrajā rindā tiek radīts programmatūras palaišanai pieejamais un aizņemtais atmiņas daudzums. Pēc raidījuma datiem, ir pieejams 778 MB un aizņemts 719 MB.

Eksperimentā tika mērīta Web REST API mikroservisa operatīvās atmiņas patēriņš. Mikroserviss tika izvietots Docker konteinerī Kubernetes klasterī. Izvietošanai tika izmantots replikācijas pārvaldnieks, lai varētu izmērīt atmiņas patēriņu dažādam mikroservisa instanču skaitam. Tika veikti 4 mērījumi:

1) pirms mikroservisa palaišanas konteinerī;
2) pēc 1 konteinera instances palaišanas;
3) pēc 2 konteinera instanču palaišanas;
4) pēc 5 konteinera instanču palaišanas.

Izmantotā replikācijas kontroliera konfigurācijas datne ir pieejama 6. pielikumā. Docker attēla konfigurācija ir pieejama 5. pielikumā.

Kopējais mikroservisa pirmkodu apjoms ir 11MB.

Replikācijas kontrolieris tika palaists ar komandu:

kubectl create -f msc-api-web-controller.yaml

Instanču skaita mainīšana notika, izmantojot komandu:

kubectl scale rc msc-api-web-controller --replicas 5

Rezultāti:

1) sistēmas raidījumi pirms mikroservisa palaišanas

	total	*used*	*free*
Mem	1497	1259	238
-/+ *buffers/cache*		716	781

Pašlaik palaista programmatūra aizņem 716 MB atmiņas.

2) sistēmas raidījumi pēc 1 konteinera instances palaišanas

	total	*used*	*free*
Mem	1497	1313	184
-/+ *buffers/cache*		770	727

Viena mikroservisa instance patērē 770 - 716 = 54 MB.

3) sistēmas raidījumi pēc 2 konteinera instanču palaišanas

	total	*used*	*free*
Mem	1497	1371	126
-/+ *buffers/cache*		827	670

Divas mikroservisa instances patērē 827 - 716 = 111 MB, vidēji 55 MB katra.

4) sistēmas raidījumi pēc 5 konteinera instanču palaišanas

	total	*used*	*free*
Mem	1497	1427	69
-/+ *buffers/cache*		984	512

Piecas mikroservisa instances patērē 984 - 716 = 268 MB, vidēji 55 MB katra.

Kā redzams, viens konteineris tērē 55 MB operatīvās atmiņas. 11 MB no tā aizņem mikroservisa pirmkods, tāpēc ir iespējams secināt, kā viena apskatīta tipa konteinera darbībai ir nepieciešams 44MB operatīvās atmiņas.

Veiktspējas testēšana

Testa gadījumā tiek ir pārbaudīts, kā vajadzīgie lauki ir pieejami, pieprasot šī rēķina datus no Invoice mikroservisa, un pieprasījums aizņem mazāk par 1 sekundi. Šī testa darbībai ir vajadzīgs palaist Invoice mikroservisu un atsevišķo mikroservisu ar līguma testa gadījumiem, kā arī testēšanas datu bāzi, kurā glabājas testēšanai izmantojama rēķina informācija. Testa gadījums tiek izpildīts Invoice mikroservisa izvietošanas laikā.

```
Feature: It is able to retrieve Invoice records stored in the database
  Scenario: Check attributes of a single Invoice record (request by ID)
    Given should wait at most 1 s until status code 200
    When the client performs GET request on /invoice/1860
    Then status code is 200
    And response contains property "id" with value "1860"
    And response contains property "name" with value "Rekins-5"
    And response contains property "createdDate" of type "long"
    And response does not contain property "updatedDate"
```

Servera kļūdu apstrāde saskarnē

Piemērā ir izmantota jQuery bibliotēka. Pirms pieprasījuma veikšanas komponente tiek bloķēta. Pēc kļūdas saņemšanas tiek veikti 10 pieprasījumi serverim ar 5 sekunžu intervālu. Pēc 10. pieprasījuma neveiksmīgas veikšanas komponentes vietā tiek paradīts ziņojums par kļūdu un pieprasījumi tiek veikti ar 1 minūtes intervālu.

```
var $invoiceForm = $('.invoiceForm');
var $pause = $invoiceForm.find('.pause');
var $serverError = $invoiceForm.find('.serverError');
function trySave(invoiceFormJson, count, timeout){
   if (count >= 10) { // ja sistēma jau tika pieprasīta 10 reizes
     if (!serverError.is('visible')) {
       serverError.show(); // paradīt kļūdas logu ar iespēju vēlreiz atsūtīt
     }
     trySave(invoiceFormJson, 0, 60000) // sūtīt katru minūti
     return; // beigt mēģinājumus
   }
   $.ajax({ // pieprasījuma veikšana
     type: 'PUT',
     url: '/invoice',
     data: invoiceFormJson,
     success: function (response) { // pieprasījums izpildīts
       pause.hide();
     },
     error: function (xhr, ajaxOptions, thrownError) { // ja sistēmai ir kļūda
       if (xhr.status == 503) { // 503: service unavailable
       // pēc 5 sekundēm atsūtīt pieprasījumu vēlreiz
         setTimeout(function() {
           trySave(invoiceFormJson, count + 1, timeout);
         }, timeout);
       }
     }
   });
}
pause.show(); // bloķē formu
var invoiceFormJson = invoiceForm.serializeObject(); // saņem formas datus JSON
trySave(invoiceFormJson, 0, 5000); // sūtīt katras 5 sekundes
```

DOKUMENTĀRĀ LAPA

Maģistra darbs: Mikroservisu arhitektūras realizācijas grūtību risināšana

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____
(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____
(Vadītāja paraksts)

Darbs iesniegts **maģistrantūras sekretariātā** _____.
(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: _____.
(Metodiķes paraksts)

Recenzents: _____
(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē
_____ prot. Nr. _____
(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____
(Sekretāra paraksts)