TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kaspar Kivistik 164347IABB

KLIENDI STAATUSE HALDUSSÜSTEEM MONESE NÄITEL

Bakalaureusetöö

Juhendaja: Viljam Puusep
MSc

Tallinn 2021

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Kaspar Kivistik

15.05.2021

Annotatsioon

Finantsasutuse infosüsteemis on olulisel kohal näidata kliendile vaid temale sobivat informatsiooni. Viis taolise tegevuse täitmiseks on määrata igale kliendile staatus.

Antud töö eesmärk on refaktoreerida finantsasutuses senini kasutusel olnud kliendi staatuse halduse süsteem ning viia see monoliitsest arhitektuurist mikroteenuste arhitektuurile. Eesmärk saavutatakse analüüsides olemasolevat süsteemi, selle tugevusi ja puudujääke, mille alusel disainitakse ja arendatakse uus infosüsteem, mis võimaldab hallata ja pärida klientide ligipääsuõiguseid.

Töö eesmärgi täitmiseks arendati rakendus, mis haldab kliendi staatusi rakendades *event-driven* arhitektuurimustreid. Analüüsiti *event-driven* arhitektuuri tugevusi ja puudujääke, millest lähtuvalt infosüsteem arendati.

Lõputöö tulemuseks on toodangkeskkonnas kasutusel infosüsteem, mis haldab keskmiselt 10 000 päringut minutis. Lõputöös tutvustatakse infosüsteemi arenduse käigus tehtud halbu otsuseid, võistujuhte ja neile loodud võimalikke lahendusi. Lisaks tutvustatakse infosüsteemi jälgimist kasutades kolmanda osapoole poolt pakutavat tarkvara. Lõputöö lõpus formuleeritakse töö kokkuvõte ja autor annab hinnangu süsteemi edasi arendamise võimalikkusele.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 25 leheküljel, 7 peatükki, 8 joonist.

// Abstract

Client access control status management system by example of Monese Ltd

In a financial institution, it is important to be able to allow access and restrict customers actions in its infosystem on demand. Way to achieve this is to assign an access control status for each customer.

The purpose of this thesis is to refactor an existing access control management system from a monolithic architecture to an architecture based on microservices. This is accomplished by analysing the existing solution used by Monese Ltd, analysing its architectural design, database and using that information to create a purpose-built information system that manages and provides client access control statuses.

Based on the introduced technologies and analysis of the existing system the thesis provides a realisation of a software application that is in used in a production environment. Main takeaways of the thesis are understanding event-driven architecture, database design and application monitoring using third party software.

The thesis covers some pitfalls and possible. solutions related to microservices based architecture, namely race conditions and mismatching data structures. At the end of the thesis an analysis of results is formulated and a possible future for the software project is presented.

The thesis is in Estonian and contains 25 pages of text, 7 chapters, 8 figures.

Lühendite ja mõistete sõnastik

AML	*Anti Money Laundering,* rahapesu tõkestamine
AWS	*Amazon Web Services,* Amazoni poolt arendatud pilveteenuste pakkuja
BAO	*Business as Usual,* tavapäraste funktsionaalsete toimingute tavapärane teostamine
FIFO	*First In First Out*, tarkvaraarenduses loodud süsteem, kus esimesena *queue'*sse saabunud sõnum väljub sealt esimesena
GDPR	*General Data Protection Regulation,* Euroopa Liidu isikuandmete kaitse üldmäärus
JSON	*Javascript Object Notation,* Javascripti programmeerimiskeel põhinev andmevahetusvorming
satelliitteenus	*Event-driven* arhitektuuris käivitatud alamteenus
SQS	*Simple Queue Service,* Amazoni poolt arendatud *queuing* tarkvara, mis võimaldab vahendada süsteemide vahel sõnumeid
queue	Eesti keeles järjekord, tuntud ka kui *message queue* on tarkvaraarenduse paradigma, kus rakenduste suhtlusel rakendatakse asünkroonset mustrit
race condition	Eesti keeles võistujuht. Olukord, kui rakendus saab sõnumi tegevustest üheaegselt erinevatel *thread*idel või sõnumid saabuvad teenusesse ebakorrapärases järjekorras
REST	*Representational State Transfer*, veebiteenuse arhitektuuri ülesehitamise viis
terviksüsteem	Infosüsteem kui tervik, mis kirjeldab kogu arendatava eesmärgi koodi

Sisukord

1 Sissejuhatus 10
1.1 Taust ja probleem 10
1.2 Ülesande püstitus ja oodatav tulemus 10
1.3 Metoodika 11
1.4 Ülevaade tööst 11
2 Teoreetiline taust 12
2.1 Ärilised nõuded 12
2.2 Funktsionaalsed nõuded 12
2.3 Pärandsüsteemid 13
2.4 Põimitus 13
3 Analüüs 15
3.1 Protsess 15
3.1.1 Staatuse muutmine automaatselt 15
3.1.2 Staatuse muutmine käsitsi 16
3.2 Olemasolev lahendus 17
3.3 Arhitektuur 19
3.3.1 Event-driven arhitektuur 19
4 Tehniline lahendus 21
4.1 Kasutatud teegid ja tehnoloogiad 21
4.1.1 Spring Boot 21
4.1.2 PostgreSQL 21
4.1.3 Amazon SQS 22
4.1.4 New Relic 22
4.2 Teenuse arhitektuur 22

4.2.1 Kliendi staatuse muutmine 22
4.2.2 Kliendi staatuse pärimine 24
4.3 Andmebaasi disaini kirjeldus 25
4.3.1 Lugemise tabel 25
4.3.2 Ajalooline tabel 26
4.3.3 Satelliitteenuste teavituste tabel 27
5 Valideerimine 28
5.1 Testimine 28
5.2 Käideldavus 28
5.2.1 Teenuse jälgimine 29
5.3 Tehnilised keerukused ja õpikohad 30
5.3.1 Võistujuhud (race conditions) 30
5.3.2 Sisend- ja väljundsõnumite valideerimine 31
6 Süsteemi edasiarendamine 33
6.1 Staatuse hulgi muutmine 33
6.2 Distributed tracing 33
6.3 Määratud staatusele kehtiva ligipääsu haldamine 33
6.4 Ajalooliste andmete rutiinne kustutamine 34
7 Kokkuvõte 35
Kasutatud kirjandus 36
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 38
Lisa 2 – Ühe miljoni kirjega AS-IS andmebaasitabeli analüüsiks loodud SQL laused ja analüüsiplaanid 39
Lisa 3 – Kümne miljoni kirjega AS-IS andmebaasitabeli analüüsiks loodud SQL laused ja analüüsiplaanid 41
Lisa 4 – Kliendi staatuse muutmine süsteemiarhitektuuri diagramm. Autori joonis. 43
Lisa 5 – Saja tuhande kirjega TO-BE andmebaasi lugemistabeli analüüsiks loodud SQL laused ja analüüsiplaanid 44

Lisa 6 – Kümne miljoni kirjega TO-BE andmebaasi lugemistabeli analüüsiks loodud SQL laused ja analüüsiplaanid 45

Lisa 7 – TO-BE andmebaasi ajaloolise tabeli loomiseks loodud SQL laused 46

Lisa 8 – TO-BE andmebaasi satelliitteenuste teavituse tabeli loomiseks loodud SQL laused 47

Jooniste loetelu

Joonis 1. Staatuse automaatne muutmine sulgemise näitel AS-IS 16

Joonis 2. Staatuse käsitsi muutmine AS-IS 17

Joonis 3. Kliendi staatuse andmebaasitabel AS-IS 18

Joonis 4. Kliendi staatuse pärimise kiirus suvalisel ajahetkel. Autori kuvatõmmis 24

Joonis 5. Kliendi staatuse lugemistabel TO-BE 26

Joonis 6. Partitsioneeritud andmebaasitabeli kustutamist kirjeldav pseudokood 26

Joonis 7. Kuvatõmmis New Relic keskkonnas näidatud käsitsi loodud meetrikatest. ... 29

Joonis 8. SELECT FOR UPDATE kasutamist kirjeldav pseudokood 31

1 Sissejuhatus

Tarkvaraarenduses esineb sagedasti olukordi kui tarkvara disainitakse ja arendatakse vajaduspõhiselt, tuvastatakse probleem ja leitakse sellele lahendus. Tarkvarasüsteemi kasvades lisandub olemasolevatele lahendustele funktsionaalsust aina juurde ja juurde. Finantsettevõtte Monese näitel on võimalik demonstreerida levinud probleemi tarkvaraarenduses: vajaduspõhiselt disainitud ja arendatud projekt võib lisanduva funktsionaalsuse tõttu hakata piirama senist kiiret süsteemi arengut [1]. Kui komponentide vahel esineb funktsionaalse vajaduseta sidusust, võib selline tihe põimitus luua olukordi, kus süsteemide haldamine ning uute arendajate ettevõttesse integreerimine muutub tarbetult keeruliseks.

1.1 Taust ja probleem

Kui alustava ettevõtte jaoks on monoliitse arhitektuurimudeli valimine mõistlik lahendus, mis võimaldab kiiret süsteemi arengut nii arenduse kui ka infrastruktuuri osas, siis laieneva ettevõtte jaoks see enam tingimata nii ei ole [2]. Süsteemi kasvades hakkab paratamatult selle haldust ja arendust modulaarsuse puudumine keerulisemaks muutma.

Monese otsustas selle keerukuse tõttu monoliitne arhitektuurimudel välja vahetada mikroteenustel põhineva vastu. Mikroteenused valiti, sest need võimaldavad paremat organiseerimist arendatava tarkvara osas ja vähendavad süsteemi põimumist [3]. Ettevõtte töötajana alates 2018. aastast on töö autoril olnud võimalus vana arhitektuuriga tutvuda.

1.2 Ülesande püstitus ja oodatav tulemus

Käesoleva bakalaureusetöö eesmärk on vastavalt töö autorile esitatud äri- ja süsteeminõuetele viia kliendi staatuse halduse süsteem monoliitsest süsteemimudelist mikroteenuse mudelile. Mikroteenustest koosneva süsteemi kui terviku tulemus on järgmine:

- Eraldada olemasolevast monoliitandmebaasist kliendi staatused eraldiseisvasse andmebaasi;
- käivitada kliendi staatuse muutmise protsess;
- salvestada kliendile määratud staatusi;
- edastada kliendi staatuse muutmine teistele mikroteenustele;
- omandada informatsiooni kliendi staatuse muutmisest;
- teha lisategevusi vastavalt satelliitteenuste tegevustele;
- kiirelt ja efektiivselt võimaldada kliendi staatuse pärimine.

1.3 Metoodika

Bakalaureusetöö valmib järgmistes etappides:

1. Tööle määratud nõuete sõnastamine vaadeldes nii äri- kui ka süsteeminõudeid.
2. Mikroteenuste arhitektuuri üldiste põhimõtete uurimine.
3. Mikroteenuste omavahelise suhtluse analüüs.
4. Andmebaasi mudeli analüüsimine ja teostamine vastavalt määratud nõuetele;
5. Rakenduse teostamine vastavalt määratud nõuetele.
6. Loodud rakenduse funktsionaalsuse testimine ning soovitud tulemuse saavutamise hindamine.
7. Lõpptulemuste formuleerimine.

1.4 Ülevaade tööst

Bakalaureusetöö raames valminud rakenduse taustast, nõuetest ja põhimõistetest annab ülevaate teine peatükk. Kolmas peatükk selgitab Monese infosüsteemis senini kasutatud lahendust ja analüüsib selle eeliseid ja puuduseid. Lisaks vaadeldakse senise lahenduse käigus täidetavat protsessi ja määratletakse uue teenuse läbiv arhitektuur.
Neljandas peatükis tutvustatakse loodud teenuse tehnilist külge, seal hulgas kasutatud tehnoloogiaid, tutvustatakse erinevaid rakenduse funktsionaalsusi, kirjeldatakse töö käigus loodav andmebaasi skeem ja selle efektiivsust. Teenuse toimimist toodangu keskkonnas, vaadeldes teenuse käideldavust ja testimist, hinnatakse viiendas peatükis. Lisaks tutvustatakse tarkvaraarenduse käigus esinenud tehnilisi probleeme. Kuuendas peatükis kirjeldatakse autori arvamust valminud teenuse võimalikest edasiarendustest.

2 Teoreetiline taust

Töö käigus loodav süsteem peab olema loodud vastavalt Monese poolt määratud nõuetele. Käesolev peatükk annab lühikese ülevaate bakalaureusetööle määratletud nõuetest ja käsitlevatest põhimõistetest.

2.1 Ärilised nõuded

Äriline nõue kohustab süsteemi täitma vajadust äril. Ärinõue ei kirjelda, mida süsteem peab tegema, vaid mis rolli süsteem peab täitma [4]. Sellest lähtudes on ärilised nõuded süsteemile järgmised:

- Iga kliendi staatust peab saama muuta.
- Kliendi ligipääs Monese infosüsteemis peab olema piiratud vastavalt tema staatusele.

2.2 Funktsionaalsed nõuded

Funktsionaalne nõue on kirjeldus, kuidas süsteem peaks käituma ja mis funktsioone ta peab täitma [5]. Käesoleva süsteemi funktsionaalsed nõuded on järgmised:

- Süsteem peab olema kirjutatud Java programmeerimiskeeles.
- Süsteem peab kasutama Spring Boot teeki.
- Süsteemil peab olema eraldiseisev andmebaas.
- Andmebaas peab kasutama PostgreSQL andmebaasisüsteemi.
- Mikroteenuste vaheline andmete edastamise metoodika peab olema lahendatud kasutades SQSi (*Simple Queue Service*).
- Mikroteenuste vaheline andmete pärimise metoodika peab olema REST (*Representational State Transfer*) protokoll.
- Kehtiva staatuse pärimine peab olema võimalikult efektiivne.

Kehtiva staatuse pärimise efektiivseks tegemiseks uuritakse bakalaureusetöö käigus seni kasutusel olnud andmebaasi ja selle tabeleid ning vastavalt loodud analüüsile luuakse uue andmebaasi skeem.

2.3 Pärandsüsteemid

Pärandsüsteem (inglise keeles *legacy system*) on süsteem, mis on infosüsteemi igapäeva tegevusele kriitilise tähtsusega, aga võib olla loodud jälgides aegunud mustreid ja tavasid. Pärandsüsteeide asendamine võib olla üks tarkvaraarenduse keerulisemaid probleeme, sest süsteemi asendamisel tuleb arvestada tõenäosusega, et teised süsteemid peavad olema sobivad nii uue kui ka vana lahendusega [6].

Süsteem muutub pärandiks, kui ta jääb ette infosüsteemi kui terviku arengule, seetõttu ei saa pärandsüsteemi tingimata nimetada halvaks süsteemiks [7]. See on loodud täpse eesmärgiga ja seda ta ka täidab. Põhjuseid pärandsüsteemi uuendada on mitu, nende seas:

- pärandsüsteemi ülalpidamine võib olla kallis [7];
- pärandsüsteemi muutmine ja edasiarendamine on pikk protsess ja võib endaga kaasa tuua terviksüsteemis ootamatuid muudatusi [7];
- pärandsüsteem on ehitatud viisil või keeles, millele on keeruline leida arendajaid seda edasi arendama ja haldama [7];
- pärandsüsteem ei vasta enam terviksüsteemi läbivale arhitektuurile [7].

2.4 Põimitus

Põimitus (inglise keeles *coupling*) iseloomustab tarkvarakomponentide vahelist tugevat seotust.

Tihe põimitus väljendub kahe või enama komponendi vahelisi sõltuvusi kas rakenduse tööfaaside, andmete liikumise või funktsionaalsuse näol. Tihe põimitus võib tekkida kiiruga arendatud tarkvarasüsteemides, kus pannakse vähe rõhku süsteemi hooldusele ja paindlikkusele. Sageli võib tihe põimitus olla põhjendatud, kui süsteem käitub kui must kast[1]. Mida väiksem süsteem on, seda kergem on tema tarkvarakomponente kokku siduda [8].

Õrn põimitus väljendub, kui terviksüsteemi tarkvarakomponendid töötavad autonoomselt. Õrnalt põimitud süsteemis tarkvarakomponendid töötavad kasutades ühist informatsiooni, aga sealjuures töötavad teadmata teise süsteemi olemasolust. Sellised

[1] Inglise keeles *black box*; Teenus, mille funktsionaalsus pole üldsusele tuntud [29]

tarkvarakomponendid demonstreerivad, kuidas tarkvarasüsteem eraldab funktsionaalsuse ja äriloogika komponentide vahele, millel ükski komponent ei sõltu teise komponendi tööst [9].

3 Analüüs

Käesolev peatükk kirjeldab olemasolevat lahendust ja sellest tulenevaid probleeme, millest lähtuvalt arendatakse bakalaureusetöö raames loodud rakendus. Protsessi kirjeldatakse lähtuvalt ärilistest nõuetest. Olemasolevat lahendust kirjeldatakse vaadeldes AS-IS tarkvarasüsteemi. AS-IS kirjeldus viitab millegi hetkeolukorrale, kui TO-BE viitab püstitatud eesmärgile [10], [11].

3.1 Protsess

Bakalaureuse töö käigus ei tohi muutuda kliendi staatuse muutmise protsess. Endiselt peab olema võimalik muuta kliendi staatust vajadusepõhiselt ning vastavalt staatusele, piirama kliendi ligipääsu Monese infosüsteemi.

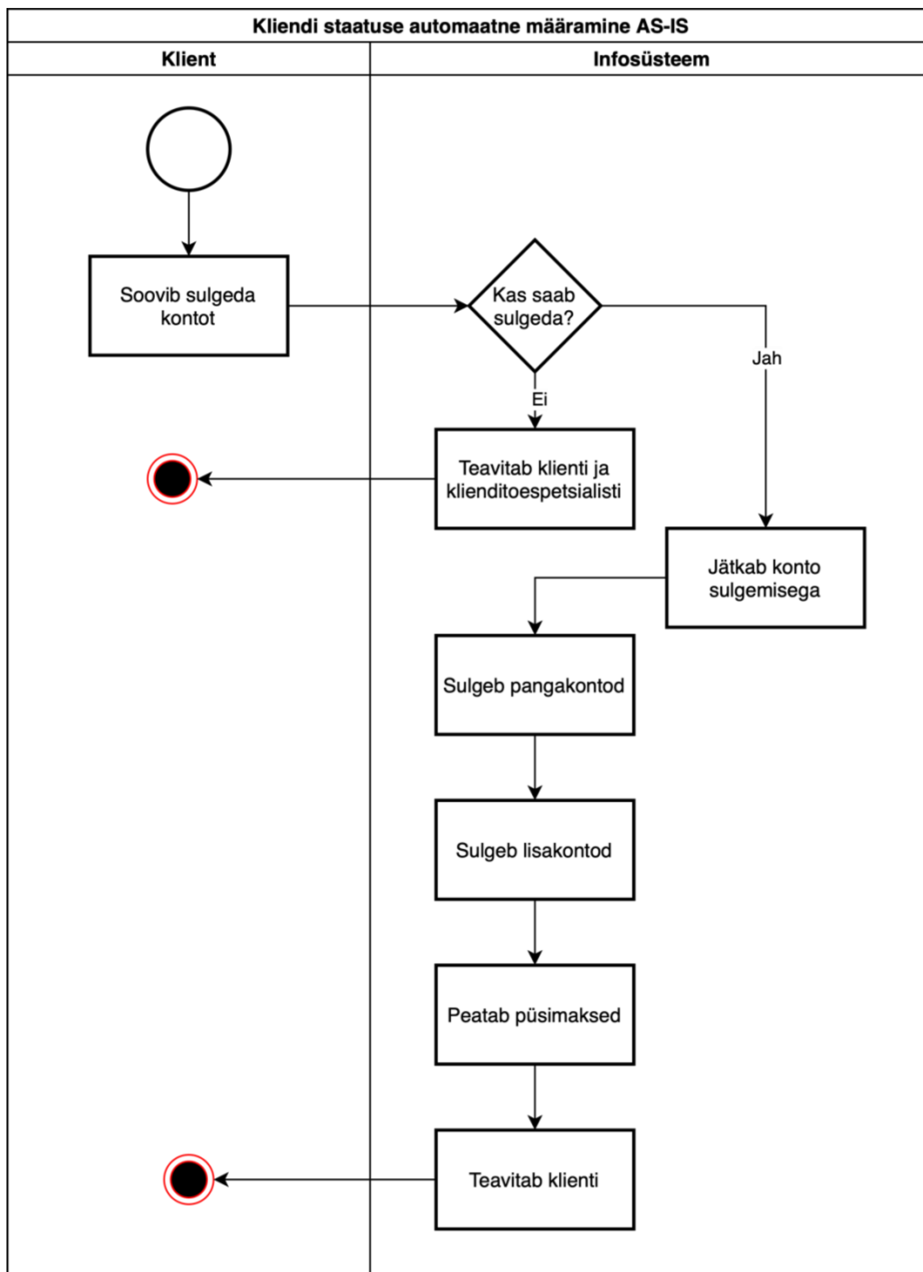
3.1.1 Staatuse muutmine automaatselt

Enamus kliendi staatuse muutmisi leiab aset automaatselt, see tähendab, et vastavalt kliendi tegevusele leiab Monese infosüsteem, et kliendi staatust on vaja muuta. Levinumad automaatselt staatuse muutmise põhjused on kliendi registreerimine või kliendi soovil konto sulgemine. Üldine äriline seisukoht on, et igasugune kliendi staatuse muutmine peab aset leidma automaatselt vähendamaks klienditoe tööd. Joonis 1 demonstreerib automaatset kliendi staatuse muutmist.

Üldiselt on staatuse muutmise käigus tarvis käivitada kõrvalisi protsesse. Näiteks kliendi kasutajakonto sulgemisel on lisaks infosüsteemi ligipääsu piiramisele vaja ka teha järgnevaid tegevusi:

- Pangakontode sulgemine
- Lisakontode sulgemine (nt kasvukontod)
- Püsimaksete peatamine

Protsessi lõppedes on veel tarvilik teavitada klienti tema staatuse muutmisest ja uutest reeglitest, mille järgi tema ligipääs infosüsteemi määratud on.



Joonis 1. Staatuse automaatne muutmine sulgemise näitel AS-IS

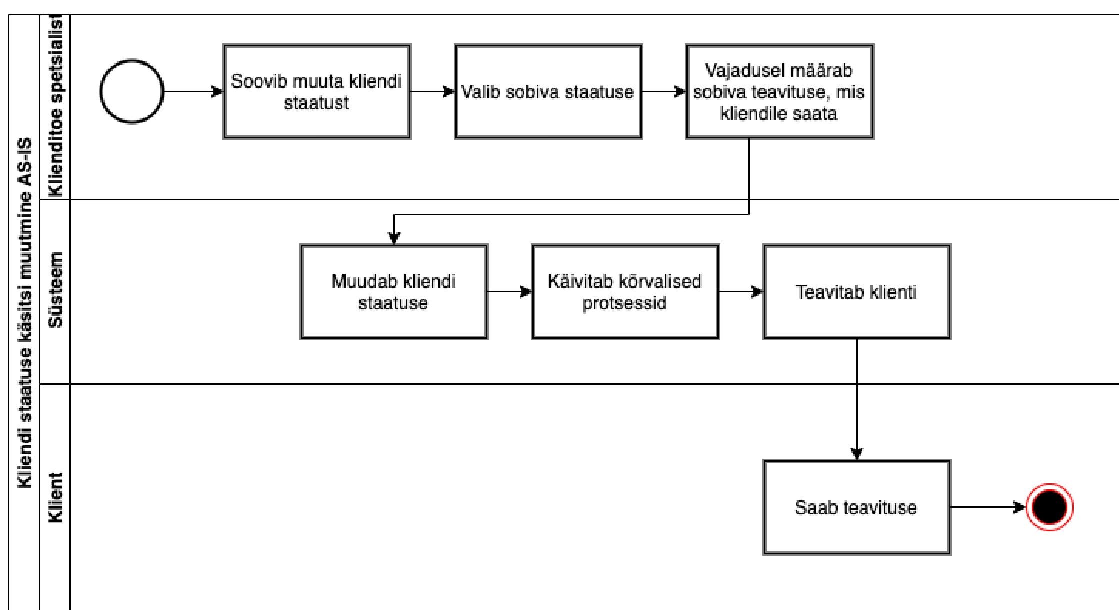
3.1.2 Staatuse muutmine käsitsi

Kliendi staatuse muutmine leiab aset Monese *backoffice*[1] tööriistas. Staatust muudetakse reeglina klienditoe spetsialisti poolt. Levinumad kasutusjuhud selleks on kliendi

[1] *Backoffice* süsteem haldab kõiki ettevõtte administratiivseid tegevusi sidudes kokku kõik ettevõtte infosüsteemid kasutajaliidesesse, kus on võimalik kliente ja nendega seonduvaid tegevusi hallata.

pääsukoodi taastamine või petturi ligipääsu blokeerimine. Joonis 2 demonstreerib kliendi staatuse käsitsi muutmist.

Käsitsi staatuse muutmisega kaasnevad samad reeglid ja kõrvalised protsessid, mis automaatselt käivitatava protsessiga. Ainsa erinevusena tuleb märkida, et käsitsi käivitatud protsessi käigus võib protsessi alustaja määrata, millise sisuga teavitus kliendile saata. Võrdluseks, automaatse protsessi käigus saadetakse alati kindlalt määratud teavitus.



Joonis 2. Staatuse käsitsi muutmine AS-IS

3.2 Olemasolev lahendus

Kliendi staatuse muutmise süsteemi olemasolev lahendus on põimitud andmebaasi, mida kasutab 2021. aasta aprilli seisuga 22 teenust, millest 13 omab õigust andmebaasi kirjutada. Süsteemide hulgas on nii monoliite kui ka neist edasi arendatud mikroteenuseid. Käesoleva lahenduse suurim mure peitub täpselt eelnevalt mainitud kättesaadavusest mitmete teenuste vahel. Jagades andmebaasi mitme teenuse vahel võib lihtsasti tekkida olukordi, kui andmebaasi skeemi (inglise keeles *schema*) muutmine toob endaga kaasa süsteemi kriitilisi muudatusi[1] [12].

[1] Muudatus süsteemis, mis võib tekitada teistes süsteemides veatingimusi

Olemasolev andmebaas on modelleeritud *append-only* [2] disaini kohaselt, mis tähendab, et ühtegi kirjet ei kustutata, selle asemel määratakse igale uuele kirjele viide, mille järgi on võimalik hinnata, kas see on ajalooline või kõige uuem sissekanne. Joonisel 3 demonstreeritakse olemasolevat andmebaasitabeli disaini.

kliendi_staatus_lugemine	
PK	**kliendi_id bigint**
	staatus varchar NOT NULL pohjus varchar NOT NULL loomise_aeg timestamptz DEFAULT NOW() NOT NULL muutmise_aeg timestamptz DEFAULT NOW() NOT NULL

Joonis 3. Kliendi staatuse andmebaasitabel AS-IS

Append-only mudeli eeliseks on andmete muutumatus [2]. Selline andmetabeli disain nõuab, et ühtegi rida ei tohi muuta ega kustutada, uue kirje lisandumisega vanad kirjed muutuvad ajalooliseks. Andmete kustutamine viiakse enamasti läbi määrates kirjele tulp kirjeldamaks, kas see on kustutatud või mitte. *Append-only* mudeli puuduseks on andmete jäädavalt kustutamine, näiteks GDPR (*General Data Protection Regulation*) määruse rakendumisel [2]. Andmed, mis on pealtnäha kustutatud, sest võimalikes kasutajaliidestes neid enam näha pole, ei ole tingimata andmebaasist eemaldatud, sest need on määratud kustutatuks. Olemasoleva lahenduse puuduseks on kehtiva staatuse pärimise kiirus.

Lähtudes bakalaureuse tööle määratud funktsionaalsest nõudest, on tarvis andmebaasitabelist võimalikult efektiivselt kätte saada kliendile viimati määratud staatus. Lisades 2 ja 3 on välja toodud PostgreSQL 11 andmebaasisüsteemis tehtud katsed, mille järgi leiti potentsiaalselt parim päringu ja indekseerimisstrateegia käesoleva andmebaasi tabelist. Tabelisse loodi 1 000 000 kirjet 100 000 kliendi kohta. Katsetuse tulemusel leiti, et kõige efektiivsem indekseerimisstrateegia on luua komposiitindeks kasutades kliendi identifikaatorit ja versiooni, sest andmebaasisüsteem tegi nii kõige vähem operatsioone sama tulemuse saavutamiseks.

3.3 Arhitektuur

3.3.1 *Event-driven* arhitektuur

Event-driven arhitektuur on tarkvaraarhitektuuri paradigma, mis kasutab rakenduse tööks sõnumeid (inglise keeles *event*). Sõnumiks peetakse midagi, mis infosüsteemi töö käigus süsteemide vahel edastatakse [13]. Käesoleva töö kontekstis olgu selleks soov kliendi staatust muuta või kliendi staatuse muutmine.

Event-driven arhitektuur võimaldab tarkvarasüsteeme funktsionaalselt lahutada, säilitades võimekuse reageerida võimalikele juhtumitele kogus infosüsteemis. Sellist arhitektuurimudelit kasutades on võimalik arendada tarkvarasüsteeme teineteisest sõltumatult, sest teenused ei pea teadma sissetuleva sõnumi allikaid ega ka, kuhu vastu saadetav sõnum jõuab. Teenused kohtlevad sõnumeid kui fakte millestki, mis on juhtunud [13].

Bakalaureusetöö käigus edastatakse infosüsteemis sõnumid teenusteni kasutades Amazoni SQS *queue* tehnoloogiat.

Sõnumite töötlemiseks on kolm üldist stiili:

- lihtne
- kompleksne
- voog (inglise keeles *stream*)

Lihtne sõnumi töötlus (inglise keeles *Simple Event Processing*) kirjeldab sõnumi protsesseerimist kui allvoo teenustes tegevuste käivitamist. Kompleksne sõnumi töötlus (inglise keeles *Complex Event Processing*) kirjeldab sõnumi protsesseerimist kui mitme sõnumi arvel konteksti loomisel ja nende terviku töötlemist. Sõnumid võivad teenuseni jõuda pika aja jooksul ja protsessi algus- ja lõppaeg pole otseselt määratletud. Voog sõnumi töötlus (inglise keeles *Stream Event Processing*) kirjeldab sõnumi saabumist süsteemi ja sellele lähtuvalt reaalajas otsuse tegemist [13].

Käesoleva bakalaureuse töö plaani kirjeldab kõige paremini lihtne sõnumite protsesseerimise stiil, sest sissetulevatest sõnumitest lähtuvalt on tarvis satelliitteenustes käivitada kõrvalisi tegevusi.

Event-driven arhitektuuri on võimalik rakendada nii monoliit- kui ka mikroteenuse arhitektuuri järgides, sõnumeid saab edastada ka ühe rakenduse siseselt, kuigi monoliitarhitektuuris läheb kaduma üks suurimaid *event-driven* arhitektuuri eeliseid, kui üks satelliitteenustest tol ajahetkel ei toimi, ei peatata kogu süsteemi toimimist, vaid jätkatakse lihtsalt ühe teenuse jagu vähema funktsionaalsusega. Kui teenus taas käivitub, on tal võimalik kõik sõnumid, mis tema *queue*'sse määratud on läbi töödelda.

Kui *event-driven* arhitektuuri võrrelda REST arhitektuuriga, kui kõik süsteemi komponendid suhtlevad teineteisega sünkroonselt, siis ükskõik millise süsteemi põrumisel on suur oht, et kogu infosüsteem võib põruda [14]. Arendatava bakalaureusetöö näitel pole kliendi staatuse muutmise käigus terviksüsteemile kriitiliselt oluline satelliitteenuste toimimine, mistõttu nende tegevuste võimalik mitte tegemine pole oluline ja seda saab ignoreerida.

4 Tehniline lahendus

4.1 Kasutatud teegid ja tehnoloogiad

4.1.1 Spring Boot

Spring Boot on avatud lähtekoodiga Java rakenduste teek. Spring Booti eesmärk on olla *out-of-the-box*[1] toode, mis toimib Spring teeki kasutades. Teegi üks suuremaid eeliseid on tema automaatne konfigureerimine, mis tähendab, et lisateeke on võimalik rakendusse integreerida vähese vaevaga [15]. Taoline *convention-over-configuration*[2] lähenemine hõlbustab rakenduse kiiret arendamist [16].

Käesolevas bakalaureusetöös kasutati Spring Booti versiooni 2.3.1. Käesolev versioon valiti, sest see oli töö alustamise hetkel kõige uuem Spring Booti versioon.

4.1.2 PostgreSQL

PostgreSQL on platvormisõltumatu ja avatud lähtekoodiga relatsiooniline andmebaasisüsteem. PostgreSQL arendamine algas 1986. aastal POSTGRES[3] projekti raames California ülikoolis Berkeleys.

Käesolevas bakalaureusetöös kasutati andmebaasisüsteemi versioon 11.5-alpine, sest see on maksimaalne versioon, mida Monese täna oma mikroteenustele pakub.

PostgreSQL 11. versiooni puuduseks on puudulik tugi tabelite partitsioneerimisele, puudub veel tugi partitsioneeritud tabelitele viitamine välisvõtmega ning loogilisele replikeerimisele, mõlemad on saanud toe vastavalt versioonis 12 ja 13 [17].

---

[1] Toode, mida saab hakata kasutama ilma lisakonfiguratsioonita

[2] Programmeerimise ideoloogia, kus üldiselt levinud situatsioonid lahendatakse kokkuleppeliselt automaatselt [28]

[3] https://www.postgresql.org/docs/current/history.html

4.1.3 Amazon SQS

Amazon SQS on AWS-i (Amazon Web Services) loodud *queuing* süsteem, mis on täielikult hallatud AWS-i poolt. See tähendab, et SQS on rakenduse jaoks kui *out-of-the-box* toode, mida saab ülesseadmisele järgnevalt kohe kasutama hakata.

SQS hõlbustab teenustevahelist sõnumite vahetust, võimaldades teenustel mitte olla teineteisest sõltuv ja põimitud. SQS võimaldab kaht tüüpi *queue*'sid: FIFO (inglise keeles *First In First Out*) ja tavaline. FIFO *queue* tagab täpse *queue*'sse sisenenud sõnumite järjekorra. Tavaline *queue* seda ei taga, aga tema kiirus sõnumite protsesseerimisel on suurem kui FIFO *queue*'l [18].

Käesolevas bakalaureusetöös kasutati Amazon SQS-i vastavalt Monese nõuetele ja sellele kuuluvale väikesele ülesseadmise vaevale. Valiti tavaline *queue* tüüp, sest sõnumite jõudmine teenusteni ebakorrapärases järjekorras on süsteemi poolt lahendatud rakendades kahesõnumilist arhitektuuri, mistõttu oli ebatarvilik valida aeglasem *queue*.

4.1.4 New Relic

New Relic on pilvepõhine platvorm, mis võimaldab arendajatel, inseneridel ja juhtidel reaalajas näha, mis süsteemides aset leiab. Platvorm võimaldab rakenduses esinevatele probleemidele kergemini jälile jõuda ja anda hinnangut, kas rakendus toimib vastavalt nõuetele [19].

4.2 Teenuse arhitektuur

Teenuse arhitektuuri saab loogiliselt jaotada kaheks: kliendi staatuse muutmine ja kliendi staatuse pärimine.

4.2.1 Kliendi staatuse muutmine

Kliendi staatuse muutmine jälgib kahesõnumilist disaini. Esimese sõnumi saab staatuse halduse teenus, misjärel teenus määrab kliendile uue staatuse ning edastab oma tegevusele vastavalt sõnumi kõikidele teistele teenustele, kes võivad olla huvitatud kliendi staatuse muutmise teavitusest.

Kliendi staatuse muutmist on võimalik algatada kahel moel: käivitades REST protokolliga teenuse *endpoint* või saates sõnumi teenuse SQS-i, misjärel sõnum teenuseni

jõuab. Saates sõnumi SQS-i, saab tegevust algatanud teenus oma tegevusi jätkata, hoolimata, kas kliendi staatus tegelikult muutus või mitte. Reeglina Monese teenustes käivitatakse REST *endpoint* olukorras, kui kliendi staatust muudetakse käsitsi, nii on võimalik klienditoe spetsialistil saada kohene tagasiside mingist tegevusest. SQS-i postitatakse sõnumeid enamasti automaatsel staatuse muutmisel, kui teenusel pole otseselt tarvis teada, kas tegevus õnnestus või mitte, ta usaldab staatust haldavat teenust, et tema viib oma tegevused täide.

Olles täitnud esialgse sõnumiga seonduva tegevuse, saadab staatuse haldusteenus teavituse kliendi staatuse muutumisest teistele teenustele. Teised teenused saavad seejärel otsustada, kas vastavalt määratud staatusele on tarvis neil mingisuguseid tegevusi täide viia. Kui on, siis nad teevad selle ja saadavad omakorda oma tegevusest teavituse tagasi staatuse haldusteenusele. Tähtis on märkida, et staatuse haldusteenusel on konfigureeritud iga kõrvalise teenuse tegevus staatuse järgi, mille kohaselt on võimalik otsustada, kas kõik kõrvalised teenused on oma tegevuse staatuse määratlemise järel lõpetanud. Staatuse haldusteenus salvestab märke teise teenuse tegevusest andmebaasi ja kui kõik vajalikud tegevused on lõpetatud, saab haldusteenus vajadusel edastada kliendile teavituse tema staatuse muutumisest.

Staatuse muutmine toimub järgnevates sammudes, mida illustreerib ka lisa 4:

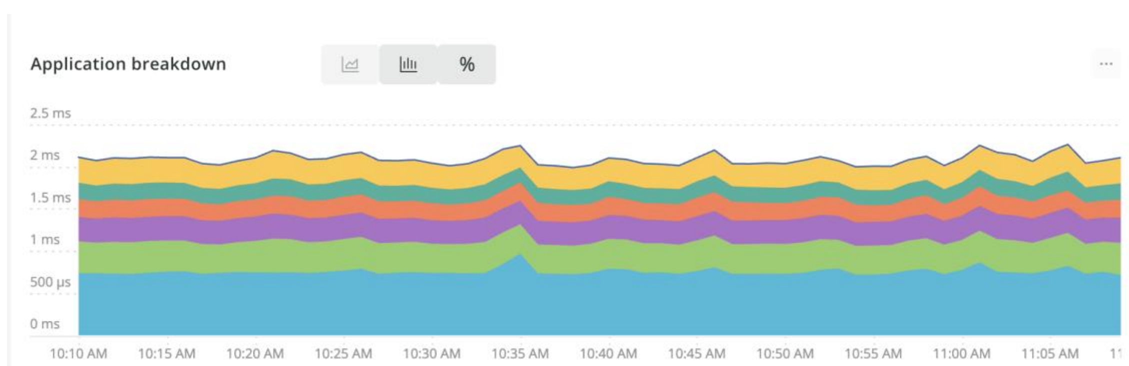
- Staatuse haldusteenus saab käskluse kliendi staatust muuta.
- Staatuse haldusteenus lisab uue staatuse andmebaasi.
- Staatuse haldusteenus väljastab sõnumi teistele teenustele kliendi staatuse muutumisest.
- Teised teenused saavad teavituse kliendi saatuse muutumisest.
- Teised teenused teevad vastavalt staatusele ettenähtud tegevuse.
- Teised teenused saadavad tagasi staatuse haldusteenusele teavituse oma tegevuste lõpetamisest läbi SQS-i.
- Staatuse haldusteenus saab sõnumi teise teenuse tegevusest.
- Staatuse haldusteenus salvestab märke teise teenuse tegevusest andmebaasi.
- Kõikide märgete saabumisel teavitatakse klienti.
- Kliendi staatuse muutmise protsess on lõppenud.

4.2.2 Kliendi staatuse pärimine

Kliendi staatuse pärimine leiab aset vaid läbi REST *endpoint*i. *Endpoint*i eesmärk on igale soovijale edastada kliendi staatus võimalikult efektiivselt. Seetõttu jätab *endpoint* igasuguse valideerimise tegemata, näiteks ei kontrollita päritava kliendi olemasolu, luuakse eeldus, et kui kliendi staatus on tabelis, siis ta on ka olemas. Kui aktiivne staatus on puudu, siis on süsteemis tekkinud veaolukord ning väljastatakse erind (inglise keeles *exception*). Käesoleva loogika kiirus on oluline, sest toodangu keskkonnas päritakse kliendi staatust minutis umbes 6000 korda.

Vaadeldes kliendi staatuse pärimist tunni aja vahemikus näeme, et keskmine päringu kiirus jääb 2 ja 2.5 millisekundi vahele (vt Joonis 4).

- Sinine värv näitab andmebaasi pärimiskiirust.
- Roheline värv näitab *endpoint*i käivitanud teenuse või töötaja tuvastamisele kulunud aega.
- Lilla värv näitab aega, mis kulub päringule *tracer*i külge panemiseks. Igale teenuses käivitatud tegevusele määratakse külge ühine identifikaator, mille abil on võimalik veaolukordi teenuse logifailist kergemini tuvastada [20].
- Punane värv näitab üldist Java koodi töö aega.
- Tumeroheline värv näitab Spring Booti poolt *endpoint*i käivitamise aega.
- Kollane värv näitab tuvastamata muude tegevustele kuluvat aega.
- Sinine joon kõige tipus näitab kogu *endpoint*i käivitamise ja vastuse saamisele kuluvat aega.



Joonis 4. Kliendi staatuse pärimise kiirus suvalisel ajahetkel. Autori kuvatõmmis

4.3 Andmebaasi disaini kirjeldus

Töö käigus asendati olemasolevas lahenduses kasutusel olnud üks andmebaasitabel kahega, loodi tabel, mis hoiab vaid kliendi viimast staatust ja tabel, mis hoiab kliendi staatuste ajaloolisi andmeid. Vastav lähenemine valiti, sest enamus päringud andmebaasi tehakse küsimaks kliendi viimati määratud staatust. Lisaks loodi tabel salvestamaks satelliitteenuste poolt saadetud teavitusi tehtud tegevustest.

4.3.1 Lugemise tabel

Lugemise tabel loodi eesmärgiga pärida andmebaasist andmeid kasutades SQL (*Structured Query Language*) päringus vaid ühte filtreerimistingimust: kliendi identifikaatorit. Ühe filtreerimistingimuse kasutamine kaotab vajaduse otsida tabelist tingimata viimast kirjet: kirje juba on seda. Lugemistabeli eeliseks on väikesem indeksi suurus, sest pärandsüsteemi andmebaasitabelisse loodud komposiitindeksi asemel on kasutusel primaarvõtmega automaatselt loodud indeks. Lisaks kaotati automaatselt genereeritud primaarvõti, see asendati kliendi identifikaatoriga. Määrates primaarvõtmeks kliendi identifikaator, pole tarvis spetsiaalselt sellele indeksit luua, PostgreSQL loob selle primaarvõtmele automaatselt [21]. Tänu sellele on tabelisse andmete lisandumisel või muutmisel tarvis uuendada vähem indekseid, mis kiirendab andmete muteerumise protseduuri [22]. Joonis 5 kirjeldab lugemistabeli pealiskaudset ülesehitust.

Lisa 1 demonstreerib lugemistabeli loomist 100 000 kliendile ja selle pärimise päringuplaani. Päringuplaani järgi on näha tunduvat pärimiskiiruse kasvu ja efektiivsust, mida iseloomustavad võtmesõnad *planning time*, *execution time* ja *cost*.

	kliendi_staatus_lugemine
PK	kliendi_id bigint
	staatus varchar NOT NULL pohjus varchar NOT NULL loomise_aeg timestamptz DEFAULT NOW() NOT NULL muutmise_aeg timestamptz DEFAULT NOW() NOT NULL

Joonis 5. Kliendi staatuse lugemistabel TO-BE

4.3.2 Ajalooline tabel

Ajalooline tabel loodi eesmärgiga salvestada andmebaasi kõik klientidele määratud staatused. Kuna kõik seni kliendile määratud staatused omavad vaid ajaloolist ja auditeerivat tähtsust, st neid ei pärita teenuste poolt, pole selle tabeli põhieesmärk andmete pärimine ja efektiivsus. Kuna ajalooline tabel kasvab aja jooksul kiiremini kui lugemistabel, siis võib juhtuda, et see tabel võib põhjustada ketta täitumist. Seetõttu on ajalooline tabel partitsioneeritud.

Ajalooline tabel partitsioneeriti ajavahemike järgi. Nii on võimalik andmeid tabelist kustutada, sooritades vaid tabeli partitsiooni kustutamine, mida demonstreerib joonis 6.

```
DROP TABLE kliendi_staatus_ajalooline_partition_2021;
```

Joonis 6. Partitsioneeritud andmebaasitabeli kustutamist kirjeldav pseudokood

Partitsiooni kustutamise eelis päringuga valimi otsimisel on, et erinevalt PostgreSQL UPDATE ja DELETE käsklustest, DROP käsklus ei salvesta töödeldavaid andmeid vahemällu, mis võib ketta täitumisel olla andmebaasi operatsioonidele laastav [23].

Lisa 7 demonstreerib ajaloolise tabeli ja sellele partitsioonide loomist.

4.3.3 Satelliitteenuste teavituste tabel

Käesoleva tabeli eesmärk on salvestada satelliitteenuste poolt tehtud tegevused, mille tulemusena on võimalik kliendi staatuse muutmise protsess kehtestada lõpetanuks. Nende andmete mitte salvestamine tähendab, et klienti ei ole võimalik teavitada staatuse muutumisest. Tabelisse salvestatavaid andmeid päritakse vaid staatuse uuendamise protsessi vältel. Kui protsess on lõppenud, muutuvad need andmed ebaoluliseks ning võib andmebaasist kustutada. Satelliitteenuste teavituste tabeli loomist kirjeldab lisa 8.

5 Valideerimine

Käesolevas peatükis kirjeldatakse rakenduse jälgimist ja veendumist kõikide komponentide töös toodangu keskkonnas.

5.1 Testimine

Kuna arendatud rakenduses on väga õhuke äriloogika kiht, pandi rakenduse testimisel suurem rõhk integratsioonitestidele.

Integratsioonitestimine on tarkvara testimise vorm, kus testitakse rakenduse eraldiseisvate osade koos töötamist eesmärgiga veendumaks, et integreeritud üksuste suhtlus toimib riketeta [24].

Testid kirjutati kasutades JUnit 5 ja Spring Boot integratsioonitestide teeke. Mõlemad teegid on Spring Boot rakendusse sisse ehitatud ja puudus vajadus nende eraldi konfigureerimiseks.

Erinevate sõnumite loogika testimiseks edastati testikeskkonnale näidissõnum samal kujul, kuidas see ka teenusesse toodangus jõuab. Nii saavutati kindlus satelliitteenuste integratsioonide toimimiseks.

5.2 Käideldavus

Teenuse käideldavus osutab, kas infosüsteem töötab ja kas selle andmed on kättesaadaval. Kõige lihtsakoelisemalt arvutatakse teenuse käideldavust valemiga (1).

$$Availability = \frac{AST - DT}{AST} \times 100\% \quad (1)$$

Valemis (1) kasutatud lühendid kirjeldatakse valemis (2).

$$\begin{aligned} &Availability - teenuse\ \text{ülaloleku}\ protsent \\ &AST - Agreed\ Service\ Time, aeg\ tundides, mil\ teenus\ peab\ olema\ kättesaadav \\ &DT - Downtime, aeg\ tundides, mis\ on\ lubatud\ teenusel\ olla\ mitteaktiivne \end{aligned} \quad (2)$$

[25]

Käesolevale teenusele on vastavalt Monese nõuetele arvutatakse käideldavus valemiga (3).

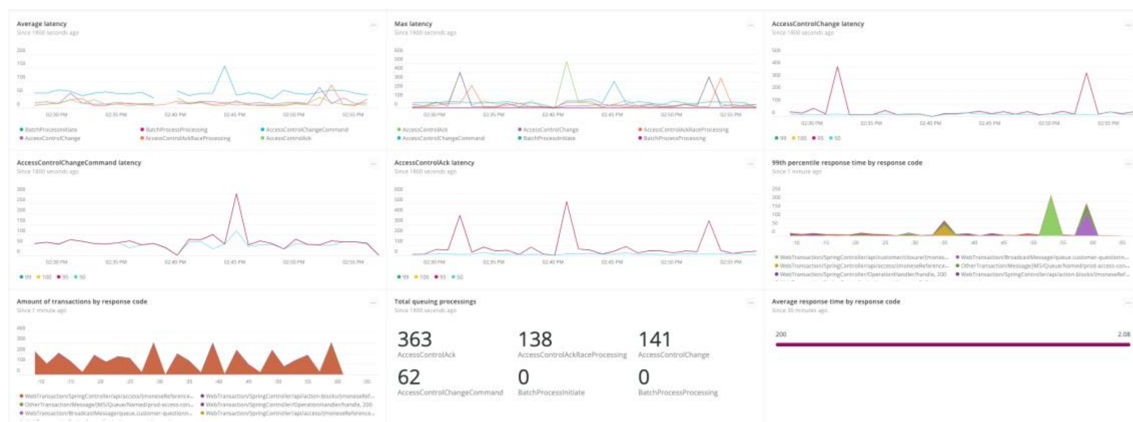
$$\frac{1000 - 1}{1000} \times 100\% = 99.9\% \quad (3)$$

Bakalaureusetöö käigus valminud teenus on olnud Monese infosüsteemides kasutusel alates 2020 aprillikuust. Selle aja jooksul pole teenusel olnud ühtegi planeeritud ega planeerimata mitteaktiivset hetke. Kõik muudatused, mis teenusega tehtud on, on olnud võimalik teostada teenust desaktiveerimata.

5.2.1 Teenuse jälgimine

Jälgimaks teenuse toimimist toodangukeskkonnas kasutati New Relic monitoorimisteenust, rakenduse töö hindamiseks ühendati rakenduse külge Micrometer[1] New Relic lisand[2], mis võimaldab saata New Relic keskkonda arendaja poolt määratud meetrikat.

Monitoorimiskeskkonda saadetakse iga REST *endpoint*i käivitamised ja nende tulemusel tekkinud vastuskoodid ning iga SQS-ist tulnud sõnum ja selle töötlemise aeg. Tänu nendele andmetele on võimalik koguda andmeid ja luua automaatseid teavitusi arendajatele kõikide käivitatud protsesside kohta süsteemis. Joonisel 6 demonstreeritakse 2021. aasta aprillis kasutusel olnud meetrikad.



Joonis 7. Kuvatõmmis New Relic keskkonnas näidatud käsitsi loodud meetrikatest.

[1] https://micrometer.io/

[2] https://micrometer.io/docs/registry/new-relic

5.3 Tehnilised keerukused ja õpikohad

Rakenduse arendamisel tuli mitmel juhul tegeleda tehniliselt keerukate olukordadega, mis nõudsid töö autorilt lisateadmiste omandamist ja oskust lahendada probleeme jäädavalt.

5.3.1 Võistujuhud (*race conditions*)

Tõenäoliselt suurim väljakutse rakenduse arendamise käigus oli võistujuhtude (inglise keeles *race condition*) lahendamine. Võistujuhtude lahendamise tulemusel tuli teha ka süsteemi arhitektuuri mõjutavaid muudatusi.

Süsteemi esialgne disain ei näinud ette kahesõnumilist arhitektuuri. Esialgse disaini kohaselt pidi staatuse haldussüsteem saama sama sõnumi, mille saavad satelliitteenused. Selline disain tekitas mitmeid probleeme.

Tekkis olukord, kui kõrvalised teenused saatsid teavituse oma tegevusest enne algse sõnumi jõudmist haldussüsteemi.

Vältimaks, et käesolev võistujuht üldse tekkida saaks, muutus kliendi staatuse muutmise disain ühelt sõnumilt kahele sõnumile. Nii on kliendi uus staatus alati andmebaasi salvestatud enne kui satelliitteenused oma sõnumi edastavad.

Lisaks tekkis olukord, kui mitu satelliitteenust saatis teavituse oma tegevusest haldussüsteemi samaaegselt, tekitades probleemi, kus kliendi staatus võis jääda mitte protsesserituks ja klienti staatuse muutmisest ei teavitatud. Probleem lahendati kahel viisil.

Veendumaks, et kliendi staatus protsesseeritakse isegi võistujuhu esinemisel, saadab haldusteenus endale meeldetuletava sõnumi paari minuti pärast peale staatuse muutmise algatust, kontrollimaks kas kõik satelliitteenused on oma tegevusest teavituse saatnud. Reeglina on kõik tegevused paari minutiga olnud lõppenud ning see sõnum ei teinud midagi. Kui esineb võistujuht, aga kõik satelliitteenused on oma tegevuse lõpetanud, saab pidada staatuse muutmist lõplikuks ning kliendile saadeti teavitus. Kui mõni teenus ei olnud oma teavitust saatnud, võib eeldada, et tol teenusel on midagi viga ning tema teavitus tuleb hiljem.

Kahe sõnumiga arhitektuur andis kindluse, tänu millele saab satelliitteenuse tegevuse salvestamise jooksul pärida kliendile määratletud staatust (kogu protsessi käivitanud

sõnumi), seades tollele andmebaasireale andmebaasilukk kogu transaktsiooni vältel. Selleks kasutati PostgreSQL SELECT FOR UPDATE käsklust. Käesolev käsklus piirab teistel andmebaasi kasutavatel teenustel ja *thread*idel valitud andmebaasi rida või ridu lugeda, muuta või kustutada enne kui lukustatud andmebaasi transaktsioon lõppenud on [26].

SELECT FOR UPDATE kasutamist kirjeldab joonis 8.

```
SELECT * FROM kliendi_staatus_ajalooline WHERE sonumi_id = :sonumi_id FOR
UPDATE
```

Joonis 8. SELECT FOR UPDATE kasutamist kirjeldav pseudokood

Andmebaasiluku määramise järel sai eemaldada süsteemi poolt automaatselt saadetud meeldetuletus, sest enam ei ole võimalik sellisel võistujuhul tekkida.

5.3.2 Sisend- ja väljundsõnumite valideerimine

Veendumaks, et teenuse sisendsõnumid oleks kõikidele Monese terviksüsteemis olevatele teenustele kättesaadavad, loodi sõnumite ja nende serialiseerimiseks teek, mis on kõigile Monese teenustele kättesaadavad.

Java programmeerimiskeele 15. versiooniga tutvustati Java ühe levinuima ajaseisu määratleva abstraktsioonile, Instant[1] klassile, nanosekundi täpsus.

Probleem esineb selles, et PostgreSQL 11. versioon veel ei toeta nanosekundi täpsust, mistõttu, kui kasutada PostgreSQLis otsitava võtmena aega, siis võib tänu Java koodi ja PostgreSQLi erinevustele esineda veaolukordi.

Käesolev rakendus kasutab aega otsimaks kirjet andmebaasitabelist. Satelliitteenuste vastuste protsesseerimise käigus otsitakse esialgset staatuse muutmise sõnumit nii sõnumi identifikaatori kui ka selle sõnumi loomise aja järgi, sest nii valitakse kõige sobivam andmebaasitabeli partitsioon.

Käesolev veaolukord lahendati luues Java keeles andmestruktuur, mida on võimalik tuletada Java Instant abstraktsioonist, eemaldades sellelt nanosekundi täpsuse. Loodud

[1] https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html

andmestruktuur määrati kõikidele sõnumitele, mis on sisendiks või väljundiks staatuse haldusteenusel.

Kuna kõik sõnumid väljuvad ja saabuvad teenusesse JSON (*Javascript Object Notation*) kujul, pidi uue andmestruktuuri tegema kergesti loetavasse JSONi kujule, eelistatult täpselt samasuguseks nagu loetakse Java Instant abstraktsiooni. Selleks kasutati Javas tuntud JSONi lugemise ja JSONisse kirjutamise teeki Jackson, millele sai kerge vaevaga anda juhised, kuidas andmestruktuuri JSON kujule kirjutada ja seda ka JSON kujust lugeda.

Nii tehti kindlaks, et sõltumata infosüsteemi komponentide versioonist või keelest, on alati rakenduse sisendväärtust võimalik luua vaid ühel moel.

6 Süsteemi edasiarendamine

Käesolevas peatükis tuuakse välja autori poolt planeeritavad edasiarendused.

6.1 Staatuse hulgi muutmine

Kõiksugu pettuse tõkestamise metoodikaga käib kaasas ka hulga klientide staatuse muutmine, reeglina suletuks. Seetõttu on üks AML (*Anti Money Laundering*) funktsioonidest pettuse tõkestamiseks kinnitatud petturitest klientide sulgemine. Luues käesoleva tegevuse nimel kasutajaliidese ja funktsionaalsuse kliente hulgi sulgeda, võib see suuresti säästa AML töötajate aega.

Lisaks vajab staatuse hulgi muutmine kasutajaliidest, kus on võimalik määrata kliendid, keda sulgeda, määrata vajalik teavitus ja kuvada kogu hulgiprotsessi staatus reaalajas.

6.2 Distributed tracing

Bakalaureusetöös loodud monitoorimise loogika kehtib vaid töö käigus loodud teenusele, jäetakse välja süsteem kui tervik. Võimaldamaks tervikliku süsteemi monitoorimist võib olla kasulik lisada süsteemile võimekus edastada kõikide mikroteenuste tasemel andmeid, millega on võimalik jälgida ühe sõnumi liikumist sõnumi tekkest kuni viimase tegevuseni.

Distributed tracing on viis, kuidas jälgida infosüsteemi tööd läbi sõnumite ja informatsiooni liiklemise mikroteenustes. [27] *Distributed tracing* võimaldab viia rakenduse monitoorimist ja vigade tuvastamist hetkelahendusest kõrgemale tasemele, sest *distributed tracing*u näol ei jälgita vaid ühte teenust, vaid süsteemi kui kogumit.

6.3 Määratud staatusele kehtiva ligipääsu haldamine

Kuna igale staatusele kehtivad ligipääsureeglid on veel pärandsüsteemi osa, oleks edasiarenduse ja süsteemi terviklikkuse nimel tarvis tuua ka igale staatusele kehtivate reeglite hoiustamine arendatud teenusesse. Siinkohal ei tasu segamini ajada

satelliitteenuste poolt tehtavaid tegevusi kliendi staatuse muutmisel, vaid teenus peab otsustama, mis telefonirakenduse vaatele kliendil on õigus siseneda või mitte.

6.4 Ajalooliste andmete rutiinne kustutamine

Töö käigus loodud andmebaasi skeem sisaldab mitmeid näiteid andmetest, mida reaalajas päritakse vähe. Näiteks hoiustatakse klientide ajaloolisi staatuseid ja satelliitteenuste teavitusi. Mõlemad tabelid sisaldavad andmeid, mida kliendi staatuse protsessi käigus päritakse, aga protsessi lõppedes muutuvad BAU (inglise keeles *business as usual*) protsesside jaoks ebavajalikuks. Andmete kogunemine põhjustada andmebaasi kettamahu enneaegset täitumist.

Seetõttu võib olla kasulik andmebaasi ajaloolisi tabeleid rutiinselt tühjendada. Tabelite kustutamine on tehtud lihtsamaks kasutades tabelite partitsioneerimist; neid saab kergesti kustutada. Kui andmebaasi ketas on jõudnud täituvuseni, on võimalik andmebaasi mahtu suurendada.

7 Kokkuvõte

Bakalaureusetöö eesmärgiks oli luua finantsasutusele Monese süsteem, mis hoiab, haldab ja pakub klientidele määratud staatusi. Selle jaoks loodi rakendus pidades silmas *event-driven* arhitektuurimustreid ja loodi uus andmebaas. Süsteem ehitati nii, et satelliitteenuste lisamine süsteemi kui tervikusse oleks võimalikult lihtne ja ei nõuaks erilisi arenduspingutusi staatuse haldussüsteemi.

Töö käigus loodud katsete järgi on näha uue süsteemi eeliseid vanaga võrreldes andmebaasi efektiivsuse ja päringute kiiruse näitel.

Töö autor õppis bakalaureusetöö kirjutamise käigus ehitama õrnalt põimituid infosüsteeme, efektiivseid andmebaasisüsteeme ja mikroteenuste arhitektuuri põhimõtteid. Lisaks õppis töö autor rakenduse tööd jälgima ja tegema otsuseid monitooringul esinenud andmete põhjal.

Loodud lahendust on võimalik edasi arendada, lisades juurde staatuste hulgi muutmise funktsionaalsust või täiendades juba käibele võetud rakenduse jälgimise strateegiaid.

Töö tulemusena jõudis arendatud süsteem toodangu keskkonda ja on igapäevases kasutuses.

Kasutatud kirjandus

[1] J. Hilton, „Why coupling will destroy your application and how to avoid it," 9 March 2016. [Võrgumaterjal]. Available: https://jonhilton.net/2016/03/09/why-coupling-will-destroy-your-application-and-how-to-avoid-it/. [Kasutatud 13 April 2021].

[2] P. Howard, „Append-only databases and the GDPR conundrum," 16 February 2018. [Võrgumaterjal]. Available: https://www.bloorresearch.com/2018/02/append-databases-gdpr-conundrum/. [Kasutatud 14 April 2021].

[3] J. Lumetta, „MONOLITH VS MICROSERVICES: WHICH IS THE BEST OPTION FOR YOU?," 10 May 2018. [Võrgumaterjal]. Available: https://www.webdesignerdepot.com/2018/05/monolith-vs-microservices-which-is-the-best-option-for-you/. [Kasutatud 13 March 2021].

[4] N. Tran, „Business Requirements VS Stakeholder Requirements," 16 December 2019. [Võrgumaterjal]. Available: https://medium.com/@nhan.tran/business-requirements-vs-stakeholder-requirements-8a5127c4fb12. [Kasutatud 13 April 2021].

[5] Altexsoft, „Functional and Nonfunctional Requirements: Specification and Types," 29 May 2018. [Võrgumaterjal]. Available: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/. [Kasutatud 13 April 2021].

[6] Gartner, „Legacy Application or System," [Võrgumaterjal]. Available: https://www.gartner.com/en/information-technology/glossary/legacy-application-or-system. [Kasutatud 20 April 2021].

[7] Talend, „What is a Legacy System?," [Võrgumaterjal]. Available: https://www.talend.com/resources/what-is-legacy-system/. [Kasutatud 20 April 2021].

[8] S. Mankovski, „Tight Coupling," %1 *Encyclopedia of Database Systems*, New York, Springer, 2018.

[9] S. Mankovski, „Loose Coupling," %1 *Encyclopedia of Database Systems*, New York, Springer, 2018.

[10] Cambridge Dictionary, „Cambridge Dictionary," Cambridge University Press 2021, [Võrgumaterjal]. Available: https://dictionary.cambridge.org/dictionary/english/as-is. [Kasutatud 12 May 2021].

[11] Cambridge Dictionary, „Cambridge Dictionary," Cambridge University Press 2021, [Võrgumaterjal]. Available: https://dictionary.cambridge.org/dictionary/english/to-be. [Kasutatud 12 May 2021].

[12] A. D. Natale, „Why a shared database is considered an anti-pattern in the microservice architecture," 28 September 2020. [Võrgumaterjal]. Available: https://thegreenerman.medium.com/why-having-a-shared-database-is-considered-an-anti-pattern-in-the-microservice-architecture-392aee75ff7d. [Kasutatud 14 April 2021].

[13] B. M. Michelson, „Event-driven architecture overview," Patricia Seybold Group 2, no. 12, 2006.

[14] G. Jansen ja J. Saladas, „Advantages of event-driven architecture," 18 June 2020. [Võrgumaterjal]. Available: https://developer.ibm.com/articles/advantages-of-an-event-driven-architecture/. [Kasutatud 19 April 2021].

[15] Spring, „Why Spring?,“ [Võrgumaterjal]. Available: https://spring.io/why-spring. [Kasutatud 29 April 2021].

[16] Spring, „Convention over configuration,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch16s10.html. [Kasutatud 29 April 2021].

[17] PostgreSQL, „About,“ [Võrgumaterjal]. Available: https://www.postgresql.org/about/. [Kasutatud 20 April 2021].

[18] Amazon Web Services, „Amazon SQS,“ [Võrgumaterjal]. Available: https://aws.amazon.com/sqs/. [Kasutatud 20 April 2021].

[19] New Relic, „New Relic,“ [Võrgumaterjal]. Available: https://newrelic.com/about. [Kasutatud 20 April 2020].

[20] C. Kidd, „Tracing vs Logging vs Monitoring: What's the Difference?,“ 8 March 2019. [Võrgumaterjal]. Available: https://www.bmc.com/blogs/monitoring-logging-tracing/. [Kasutatud 29 April 2021].

[21] PostgreSQL, „DDL Constraints,“ [Võrgumaterjal]. Available: https://www.postgresql.org/docs/11/ddl-constraints.html. [Kasutatud 15 April 2021].

[22] PostgreSQL, „Indexes,“ [Võrgumaterjal]. Available: https://www.postgresql.org/docs/11/indexes-intro.html. [Kasutatud 27 April 2021].

[23] PostgreSQL, „Routine Vacuuming,“ [Võrgumaterjal]. Available: https://www.postgresql.org/docs/11/routine-vacuuming.html#VACUUM-FOR-SPACE-RECOVERY. [Kasutatud 15 April 2021].

[24] „Integration Testing,“ [Võrgumaterjal]. Available: https://softwaretestingfundamentals.com/integration-testing/. [Kasutatud 20 April 2021].

[25] S. Rance, „How to Define, Measure and Report IT Service Availability,“ 2018 June 22. [Võrgumaterjal]. Available: https://itsm.tools/how-to-define-measure-and-report-service-availability/. [Kasutatud 2021 April 19].

[26] PostgreSQL, „Explicit Locking,“ [Võrgumaterjal]. Available: https://www.postgresql.org/docs/11/explicit-locking.html#LOCKING-ROWS. [Kasutatud 16 April 2021].

[27] Open Tracing, „opentracing.io,“ [Võrgumaterjal]. Available: https://opentracing.io/docs/overview/what-is-tracing/. [Kasutatud 20 April 2021].

[28] F. Sáez, „Convention Over Configuration,“ [Võrgumaterjal]. Available: https://facilethings.com/blog/en/convention-over-configuration. [Kasutatud 20 April 2021].

[29] ReqTest, „Black Box Testing - Understanding the Basics,“ 8 August 2019. [Võrgumaterjal]. Available: https://reqtest.com/testing-blog/black-box-testing/. [Kasutatud 27 April 2021].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Kaspar Kivistik

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Kliendi staatus haldussüsteem Monese näitel", mille juhendaja on Viljam Puusep
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

15.05.2021

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Ühe miljoni kirjega AS-IS andmebaasitabeli analüüsiks loodud SQL laused ja analüüsiplaanid

```
CREATE TABLE kliendi_staatus (
  id             bigserial PRIMARY KEY,
  kliendi_id     bigint                   NOT NULL,
  kliendi_staatus varchar                 NOT NULL,
  pohjus         varchar                  NOT NULL,
  loomise_aeg    timestamptz DEFAULT NOW() NOT NULL,
  versioon       int                      NOT NULL
);

INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 1
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 2
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 3
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 4
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 5
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 6
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 7
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 8
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 9
FROM GENERATE_SERIES(1, 100000) AS g (id);
INSERT INTO kliendi_staatus (kliendi_id, kliendi_staatus, pohjus, versioon)
SELECT g.id, 'OK', 'OK', 10
FROM GENERATE_SERIES(1, 100000) AS g (id);

CREATE INDEX idx_kliendi_staatus_kliendi_id_versioon ON kliendi_staatus
(kliendi_id, versioon);
EXPLAIN ANALYSE
SELECT *
FROM kliendi_staatus
WHERE kliendi_id = 32425
ORDER BY versioon DESC
LIMIT 1;
```

```sql
DROP INDEX idx_kliendi_staatus_kliendi_id_versioon;

CREATE INDEX idx_kliendi_staatus_kliendi_id ON kliendi_staatus (kliendi_id);
EXPLAIN ANALYSE
SELECT *
FROM kliendi_staatus
WHERE kliendi_id = 32425
ORDER BY versioon DESC
LIMIT 1;
```

```
QUERY PLAN
1 Limit  (cost=0.42..3.54 rows=1 width=92) (actual time=0.061..0.062 rows=1 loops=1)
2   ->  Index Scan Backward using idx_kliendi_staatus_kliendi_id_versioon on kliendi_staatus  (cost=0.42..15555.92 rows=5000 width=92) (actual time=0.060..0.060 rows=1 loops=1)
3         Index Cond: (kliendi_id = 32425)
4 Planning Time: 0.424 ms
5 Execution Time: 0.077 ms
```

```
QUERY PLAN
1  Limit  (cost=43.64..43.64 rows=1 width=34) (actual time=0.111..0.112 rows=1 loops=1)
2    ->  Sort  (cost=43.64..43.66 rows=10 width=34) (actual time=0.110..0.110 rows=1 loops=1)
3          Sort Key: versioon DESC
4          Sort Method: top-N heapsort  Memory: 25kB
5          ->  Bitmap Heap Scan on kliendi_staatus  (cost=4.50..43.59 rows=10 width=34) (actual time=0.085..0.100 rows=10 loops=1)
6                Recheck Cond: (kliendi_id = 32425)
7                Heap Blocks: exact=10
8                ->  Bitmap Index Scan on idx_kliendi_staatus_kliendi_id  (cost=0.00..4.50 rows=10 width=0) (actual time=0.078..0.078 rows=10 loops=1)
9                      Index Cond: (kliendi_id = 32425)
10 Planning Time: 0.470 ms
11 Execution Time: 0.133 ms
```

Lisa 3 – Kümne miljoni kirjega AS-IS andmebaasitabeli analüüsiks loodud SQL laused ja analüüsiplaanid

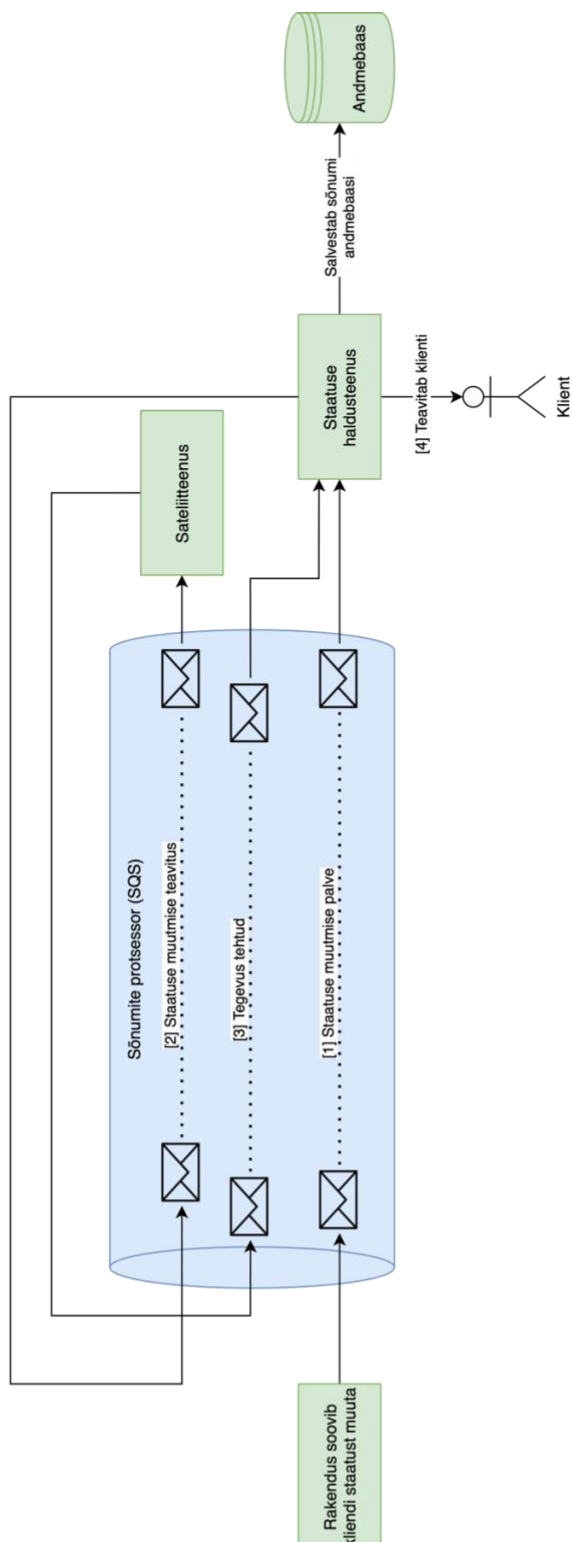
```
CREATE TABLE kliendi_staatus_10_mil (
  id             bigserial PRIMARY KEY,
  kliendi_id     bigint                     NOT NULL,
  kliendi_staatus varchar                   NOT NULL,
  pohjus         varchar                    NOT NULL,
  loomise_aeg    timestamptz DEFAULT NOW()  NOT NULL,
  version        int                        NOT NULL
);

INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 1
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 2
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 3
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 4
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 5
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 6
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 7
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 8
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 9
FROM GENERATE_SERIES(1, 1000000) AS g (id);
INSERT INTO kliendi_staatus_10_mil (kliendi_id, kliendi_staatus, pohjus,
version)
SELECT g.id, 'OK', 'OK', 10
FROM GENERATE_SERIES(1, 1000000) AS g (id);
```

```
CREATE INDEX idx_kliendi_staatus_10_mil_kliendi_id_versioon ON
kliendi_staatus_10_mil (kliendi_id, version);
EXPLAIN ANALYSE
SELECT *
FROM kliendi_staatus_10_mil
WHERE kliendi_id = 32425
ORDER BY version DESC
LIMIT 1

DROP INDEX idx_kliendi_staatus_10_mil_kliendi_id_versioon;

CREATE INDEX idx_kliendi_staatus_10_mil_kliendi_id ON kliendi_staatus_10_mil
(kliendi_id);

EXPLAIN ANALYSE
SELECT *
FROM kliendi_staatus_10_mil
WHERE kliendi_id = 32425
ORDER BY version DESC
LIMIT 1
```

```
   QUERY PLAN
1  Limit  (cost=0.43..3.54 rows=1 width=92) (actual time=0.041..0.042 rows=1 loops=1)
2    ->  Index Scan Backward using idx_kliendi_staatus_10_mil_kliendi_id_versioon on kliendi_staatus_10_mil  (cost=0.43..155495.43 rows=50000 width=92) (actual time=0.039..0.040 rows=1 loops=1)
3          Index Cond: (kliendi_id = 32425)
4  Planning Time: 0.342 ms
5  Execution Time: 0.078 ms
```

```
    QUERY PLAN
1   Limit  (cost=77273.43..77273.43 rows=1 width=92) (actual time=0.387..0.389 rows=1 loops=1)
2     ->  Sort  (cost=77273.43..77398.43 rows=50000 width=92) (actual time=0.386..0.386 rows=1 loops=1)
3           Sort Key: version DESC
4           Sort Method: top-N heapsort  Memory: 25kB
5           ->  Bitmap Heap Scan on kliendi_staatus_10_mil  (cost=939.93..77023.43 rows=50000 width=92) (actual time=0.297..0.376 rows=10 loops=1)
6                 Recheck Cond: (kliendi_id = 32425)
7                 Heap Blocks: exact=10
8                 ->  Bitmap Index Scan on idx_kliendi_staatus_10_mil_kliendi_id  (cost=0.00..927.43 rows=50000 width=0) (actual time=0.033..0.033 rows=10 loops=1)
9                       Index Cond: (kliendi_id = 32425)
10  Planning Time: 0.309 ms
11  Execution Time: 0.415 ms
```

Lisa 4 – Kliendi staatuse muutmine süsteemiarhitektuuri diagramm. Autori joonis.



Lisa 5 – Saja tuhande kirjega TO-BE andmebaasi lugemistabeli analüüsiks loodud SQL laused ja analüüsiplaanid

```
CREATE TABLE kliendi_staatus_lugemine (
  id          bigserial     PRIMARY KEY,
  kliendi_id  bigint UNIQUE             NOT NULL,
  staatus     varchar                   NOT NULL,
  pohjus      varchar                   NOT NULL,
  loomise_aeg timestamptz DEFAULT NOW() NOT NULL,
  muutmise_aeg timestamptz DEFAULT NOW() NOT NULL
);

INSERT INTO kliendi_staatus_lugemine (kliendi_id, staatus, pohjus)
SELECT g.id, 'OK', 'OK'
FROM GENERATE_SERIES(1, 100000) AS g(id);

CREATE INDEX idx_kliendi_staatus_lugemine_kliendi_id ON
kliendi_staatus_lugemine (kliendi_id);
```

```
  QUERY PLAN
1 Index Scan using idx_kliendi_staatus_lugemine_kliendi_id on kliendi_staatus_lugemine  (cost=0.29..8.31 rows=1 width=96) (actual time=0.028..0.029 rows=1 loops=1)
2   Index Cond: (kliendi_id = 32425)
3 Planning Time: 0.070 ms
4 Execution Time: 0.045 ms
```

Lisa 6 – Kümne miljoni kirjega TO-BE andmebaasi lugemistabeli analüüsiks loodud SQL laused ja analüüsiplaanid

```
CREATE TABLE kliendi_staatus_lugemine_10_mil (
  kliendi_id   bigint PRIMARY KEY,
  staatus      varchar                    NOT NULL,
  pohjus       varchar                    NOT NULL,
  loomise_aeg  timestamptz DEFAULT NOW() NOT NULL,
  muutmise_aeg timestamptz DEFAULT NOW() NOT NULL
);

INSERT INTO kliendi_staatus_lugemine_10_mil (kliendi_id, staatus, pohjus)
SELECT g.id, 'OK', 'OK'
FROM GENERATE_SERIES(1, 10000000) AS g(id);

CREATE INDEX idx_kliendi_staatus_lugemine_10_mil_kliendi_id ON
kliendi_staatus_lugemine_10_mil (kliendi_id);
```

```
  QUERY PLAN
1 Index Scan using idx_kliendi_staatus_lugemine_10_mil_kliendi_id on kliendi_staatus_lugemine_10_mil  (cost=0.43..8.45 rows=1 width=88) (actual time=0.179..0.180 rows=1 loops=1)
2   Index Cond: (kliendi_id = 32425)
3 Planning Time: 0.435 ms
4 Execution Time: 0.203 ms
```

Lisa 7 – TO-BE andmebaasi ajaloolise tabeli loomiseks loodud SQL laused

```
CREATE TABLE kliendi_staatus_ajalooline (
  sonumi_id          uuid,
  kliendi_id         bigint          NOT NULL,
  staatus            varchar         NOT NULL,
  pohjus             varchar         NOT NULL,
  aeg_loodud         timestamptz     NOT NULL,
  aeg_protsesseritud timestamptz,
  looja_nimi         bigint          NOT NULL,
  PRIMARY KEY (sonumi_id, aeg_loodud)
) PARTITION BY RANGE (aeg_loodud);

CREATE INDEX idx_kliendi_staatus_ajalooline_kliendi_id_aeg_loodud ON
kliendi_staatus_ajalooline (kliendi_id, aeg_loodud);

CREATE TABLE kliendi_staatus_ajalooline_partition_default PARTITION OF
kliendi_staatus_ajalooline DEFAULT;
CREATE TABLE kliendi_staatus_ajalooline_partition_2015 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2015-01-01') TO ('2016-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2016 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2016-01-01') TO ('2017-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2017 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2017-01-01') TO ('2018-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2018 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2018-01-01') TO ('2019-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2019 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2019-01-01') TO ('2020-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2020 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2021 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');
CREATE TABLE kliendi_staatus_ajalooline_partition_2022 PARTITION OF
kliendi_staatus_ajalooline FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');
```

Lisa 8 – TO-BE andmebaasi satelliitteenuste teavituse tabeli loomiseks loodud SQL laused

```
CREATE TABLE satelliiteenuse_teavitus (
  sonumi_id      uuid,
  teenuse_nimi   varchar,
  aeg_loodud     timestamptz DEFAULT NOW() NOT NULL,
  looja_nimi     bigint                    NOT NULL,
  PRIMARY KEY (sonumi_id, teenuse_nimi)
) PARTITION BY RANGE (aeg_loodud);

CREATE INDEX idx_satelliitteenuse_teavitus_sonumi_id ON
satelliiteenuse_teavitus (sonumi_id);
```