TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Reiko-Rainer Reinup 179430IADB

Andmetervikluse ja -kvaliteedi täiustamine andmemustrite abil

bakalaureusetöö

Juhendajad: Jaanus Põial

PhD

Argo Kivirüüt

BSc

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: Reiko-Rainer Reinup

Annotatsioon

Tarkvaraprojekti elutsükli jooksul tekib rakenduses suuremal või vähemal määral andmevigu. Need vead võivad oma olemuselt olla märkamatud, aga vahepeal ka äriliselt kriitilised, põhjustades halvemal juhul otsest rahalist kahju.

Töö on kirjutatud tellija olemasoleva süsteemi vaatest. Süsteemi kasutavad tellija töötajad, kes viivad süsteemis läbi oma tööülesandeid. Ülesannete täitmise viimane samm lõpeb tihti tellija raha välja saatmisega füüsilistele isikutele.

Antud töö raames otsitakse lahendusi tellija süsteemi andmetervikluse ja andmekvaliteedi täiustamiseks. Töö jooksul uuritakse võimalikke lahendusi, analüüsides nende sobivust tellija süsteemi ja soovidega.

Töö käigus realiseeritakse ja testitakse lahendused, mis vastavad tellija soovidele ja võtavad arvesse tellija süsteemi olemasolevat arhitektuuri, äriprotsesse ja tarkvaraarenduse häid tavasid. Loodud lahendused vähendavad tellija süsteemis andmevigade tekkimist ja võimaldavad andmevigu kiiremini leida.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 6 peatükki, 33 joonist.

Abstract

Data Integrity and Quality Improvement Using Data Patterns

Faulty data is bound to be found in all software projects to a greater or lesser extent. These faults are often unnoticable, but can be critical and cause direct monetary damage.

The thesis is written from the perspective of a client's existing system. The users of the system are employees of the client, who use the system to perform their tasks. The tasks often end with sending money from the client to other physical persons.

In the analysis phase solutions to improve the data integrity and data quality of the client's system are researched. Possible solutions are analysed and their compatibility with the client's system and wishes are evaluated.

The analysed solutions are implemented and tested taking into consideration the client's system's existing architecture, business processes and good practices of software development. Implemented solutions reduce the emergence and allow quicker detection of faulty data

The thesis is in Estonian and contains — 40 pages of text, 6 chapters, 33 figures.

Lühendite ja mõistete sõnastik

API	*Application programming interface,* rakendusliides
AngularJS	JavaScripti raamistik kasutajaliideste arendamiseks
annotatsioon	*Annotation*, töös mõeldud Java keeles kasutusel metaandmete märkimisvormi
cron	Käesolevas töös mõeldud cron lauseid, mis määravad protsessi käivitamise ajavahemiku.
DML	Data Manipulation Language
endpoint	Ühenduspunkt, kust pakutakse mingit teenust
Hibernate	Tööriist relatsioonilise andmemudeli kaardistamiseks objektorienteeritud andmemudeliks
Hoidla	*Repository*, töös mitte koodihoidla, vaid andmebaasiliides
HTTP	*Hypertext Transfer Protocol,* andmevahetusprotokoll veebis
Java	Töös kasutatav tagarakenduse programmeerimiskeel
Moodul	Tellija süsteemi komponent, millest võib töö raames mõelda kui mikroteenusest.
Oracle	Andmebaasimootor
PostgreSQL	Andmebaasimootor
Spring Boot	Töö tagarakenduses kasutusel Java raamistik

Sisukord

1.Sissejuhatus 10

1.1 Projekti taust 10

1.2 Ülesande püstitus 10

1.3 Eesmärk 11

1.4 Metoodika 11

2.Ärianalüüs 13

2.1. Reeglimootor 13

2.1.1. Reeglid 13

2.1.1.1. Maksete reeglid 14

2.1.2. Vigaste andmete tuvastamise aeg 14

2.1.3. Vigaste andmete ülevaade 15

2.2. Automaatprotsessid 16

2.2.1. Avalduse lukud 17

3.Tehniline analüüs 20

3.1. Reeglimootor 20

3.1.1. Struktuur 21

3.1.2. Hoiustamine 23

3.1.3. Protsessi käivitamine 24

3.1.4. Administraatori ülevaade 25

3.2. Automaatprotsessid 26

3.2.1. Luku struktuur 28

4.Realiseerimine 29

4.1.Tagarakendus 29

4.1.1.Reeglimootor 29

4.1.2.Automaatprotsessid 32

4.2.Kasutajaliides 35

4.2.1.Reeglimootor 35

4.2.2.Automaatprotsessid 38

4.3.Testimine 39

4.3.1. Automaattestid....39
4.3.2. Manuaalne testimine....42
5.Hinnang....47
5.1.Tulemused....47
5.2.Edasiarendused....47
6.Kokkuvõte....49
Kasutatud kirjandus....50
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks....52
Lisa 2 – Algskript andmebaasi eksempari initsialiseerimiseks....53

Jooniste loetelu

Joonis 1. Tellija süsteemi lihtsustatud äriprotsess. 18
Joonis 2. Kuvatõmmis oletatavast andmemudeli struktuurist. 23
Joonis 3. Visand planeeritavast administraatoripaneelist. 25
Joonis 4. Administraatorile saadetav e-kirja kuju. 25
Joonis 5. Koodilõik andmekontrolli käivitamise meetodist. 30
Joonis 6. Koodilõik DML lause käivitamise ja tulemuste tagastamise meetodist. 30
Joonis 7. Koodilõik e-kirja saatmise meetodist parameetri väärtuse põhjal. 31
Joonis 8. Koodilõik e-kirja sisu koostamise meetodist tulemuste põhjal. 32
Joonis 9. Koodilõik avalduse domeeniobjekti klassi meetoditest. 33
Joonis 10. Koodilõik avalduse teenusklassi masslukustamismeetodist. 33
Joonis 11. Koodilõik luku lisamisest maksude arvutamise protsessile. 34
Joonis 12. Koodilõik luku lisamisest makseridade eksportimise protsessile. 34
Joonis 13. Koodilõik luku lisamisest digiallkirjastamise protsessile. 34
Joonis 14. Näide AngularJS andmeallikast. 35
Joonis 15. Reeglimootori failide puu. 36
Joonis 16. ruleEngine.js faili sisu. 36
Joonis 17. Koodilõik päringu tulemuse salvestamisest $scope muutujasse. 37
Joonis 18. Koodilõik tabeli genereerimisest. 37
Joonis 19. Koodilõik andmekontrolli käivitamisest. 38
Joonis 20. Koodilõik avalduse lukustatuse teate kuvamiseks. 38
Joonis 21. Kuvatõmmis lukustatud avalduse teatest ja mitteaktiivsetest nuppudest. 39
Joonis 22. Meetod avalduse muutmise lubamise kontrollimiseks. 39
Joonis 23. Testklassi deklaratsioon koos annotatsioonidega. 40
Joonis 24. Koodilõik PostgreSQL konteineri loomisest ja testi initsialiseerijast. 41
Joonis 25. Koodilõik reeglimootori automaattestist. 41
Joonis 26. Lõik logidest pärast kasutajaliidese nupuvajutust. 42
Joonis 27. Kuvatõmmis edukast andmekontrollist kasutajaliideses. 43
Joonis 28. Kuvatõmmis mustri muutmisest kasutajaliideses. 43
Joonis 29. Kuvatõmmis kasutajaliideses pärast andmekontrolli inaktiivse mustriga. 44

Joonis 30. Lõik logidest inaktiivse mustriga. 44
Joonis 31. Kuvatõmmis saadetud e-kirjast andmekontrolli tulemustega. 45
Joonis 32. Lõik rakenduse logist maksude arvutamise protsessi ajal. 46
Joonis 33. Kuvatõmmis automaatprotsessi poolt lukustatud avaldusest. 46

1. Sissejuhatus

Tarkvaraprojekti elutsükli jooksul tekib rakenduses suuremal või vähemal määral vigu. Need vead võivad oma olemuselt olla märkamatud, aga vahepeal ka äriliselt kriitilised, põhjustades halvemal juhul otsest rahalist kahju.

1.1 Projekti taust

Tellija süsteem on alates 2015. aastast aktiivses arenduses ja kasutuses olnud. Rakendust kasutavad tellija töötajad oma tööülesannete täitmisel. Ülesannete täitmise viimane samm lõpeb tihti tellija raha välja saatmisega füüsilistele isikutele.

Antud töös käsitletakse üht konkreetset tellija süsteemi, millega autor seotud on, kuid see on vaid osa tellija rakenduste võrgustikust. Kõik rakendused suhtlevad omavahel ja vahetavad andmeid. Kuna andmete vahetust on palju ja neid sisestavad tihti inimesed käsitsi, siis võib tekkida olukord, kus tellija süsteemi andmebaasi jõuavad vigased andmed.

Käesoleva pandeemia tõttu on andmemahud süsteemis kasvanud ning sellega koos ka vigaste andmete raporteerimine. Andmete aegsasti parandamata jätmisest tingituna on esinenud intsidente, mille tulemusena sai tellija rahalist kahju väljasaadetava raha või ajakulu näol.

1.2 Ülesande püstitus

Vigastest andmetest võivad tekkida vead süsteemi käitumises, mis mõjutab otseselt süsteemi kasutajaid. Vigaseid andmeid leiab süsteemist vaid kasutajate abil, kes süsteemis igapäevaseid toiminguid teevad ja vigu märkavad.

Tellija süsteemis töötab iga päev mitmeid automaatseid protsesse. Mõned näited nendest automaatprotsessidest on massiline maksude arvutamine, rahasummade väljamaksmine ja rahaga seonduvate otsuste tegemine. Automaatprotsessid vastutavad ettekirjutatud reeglite alusel kogu protsessi kulgemise eest ise. Enamasti on kogu protsess süsteemi kasutaja eest peidetud.

Igal automaatprotsessil on põhjalikud ettekirjutatud reeglid, kuid tellija süsteemi nüansirikkas valdkonnas on alati haruldasi erandeid ja piirjuhte, mida ei ole reeglitesse kirja pandud, sest see ei ole olnud sel hetkel äriliselt mõistlik. See tähendab, et aeg-ajalt vigaste andmete tekkimine on automaatprotsessi puhul sisse arvestatud.

Automaatprotsessid on tellija süsteemis vajalikud, sest tellija tööülesanded on seadusandlusest tulenevalt kindla tähtajaga ja manuaalselt ei jõuaks tellija neid läbi töötleda.

Viimase aasta jooksul on andmemahud tellija süsteemis mitmekordistunud, mis tähendab, et tellija süsteemis on tööülesannete hulk märkimisväärselt kasvanud ja nii automaatprotsessidest kui ka käsitsi sisestatavatest andmetest tulenevaid vigu tekib aina rohkem.

Tellija süsteemis esinevad andmevead tekitavad tellijale rahalist kahju nii parandamisele kuluva aja näol kui ka otseselt vigaste andmete tõttu ebakorrektsete maksete välja saatmise näol.

1.3 Eesmärk

Andmevigade kasvav hulk süsteemis on üleliigne ja ebavajalik koormus nii süsteemi kasutajatele kui ka süsteemi arendajatele, millest tellija saab kahju.

Töö eesmärk on täiustada tellija projektis andmevigade leidmist ja vigade vältimist. Töö eesmärk on tuvastada andmevead enne, kui nad jõuavad tekitada tellijale ajalist ja rahalist kahju.

Selle jaoks analüüsitakse võimalikke lahendusi ja realiseeritakse süsteemile arendused vigaste andmete kiiremaks tuvastamiseks ja andmevigade vältimiseks.

1.4 Metoodika

Esimese sammuna tuleb analüüsida probleemi ning ärilisi ja tehnilisi piiranguid. Autor alustab olemasoleva lahenduse analüüsimist, otsides tellija süsteemis kriitilisi kohti, mis on altid andmevigade tekkimisele. Seejärel tuleb analüüsida, kuidas on mõistlik

andmekontrolli teostada. Teostatud analüüsi põhjal tuleb realiseeritavat lahendust planeerida.

Autoril tuleb analüüsida autoril püstitatud eesmärkide täitmiseks võimalikke tarkvaralisi lahendusi, pidades silmas töö mahtu ning kliendipoolseid ärinõudeid ja -piiranguid. Arvestada tuleb süsteemi jätkusuutlikkusega.

Pärast planeerimist on vaja analüüsi osas välja töötatud lahendus realiseerida ja testida.

Viimase sammuna tuleb analüüsida tulemusi ning võimalikke murekohti. Tuleb välja selgitada, kas süsteem vastab algselt püstitatud eesmärkidele. Välja tuua võimalikud süsteemi edasiarendused.

2. Ärianalüüs

Käesolevas peatükis antakse ülevaade loodava süsteemi äriprotsessist ning planeeritakse lahendus.

Aruteludes tellija ja süsteemi analüütikutega tuli ilmsiks, et tellija soovib oma süsteemis läheneda andmeterviklusele ja -kvaliteedi täiustamisele kahest suunast: ristuvate automaatprotsesside leevendamine ja andmete kuju jälgimine andmebaasis andmemustrite abil.

2.1. Reeglimootor

Reeglimootor on süsteem, mille eesmärk on ärireeglite rakendamine ja muutmine rakenduse töötamise ajal [1]. Antud töö kontekstis tähendab see tellija süsteemi andmete õigsuse kontrollimist, kasutades ettekirjutatud reegleid.

Reeglimootori abil saab kirjeldada andmete kuju ja vastavat kuju kontrollida kirjeldatud andmemustriga. Kuna üks töö eesmärkidest on vigaste andmete kiirem tuvastamine, siis on reeglimootori kasutuselevõtt seeläbi sobilik lahendus.

Vigaste andmete otsimise protsessi ümbritsevad detailid vajavad põhjalikku analüüsi. Süsteemi administraator peab vigadest teadlik olema, et vajadusel neid parandada. Lisaks on vaja ka välja selgitada, millise ajalise värskusega veaandmeid on vaja, et tellija süsteem loodavast süsteemist kasu saab.

2.1.1. Reeglid

Reeglite loomiseks on vaja välja selgitada nõuded, mis teatud andmetele kehtima peavad. Praegu on tellija huvitatud eelkõige makseridadele rakenduvate reeglite täideviimisest. Reeglimootori arendust ja analüüsi see ei mõjuta, sest reeglimootorisse on võimalik kirjutada tulevikus mistahes andmetele rakenduvaid reegleid.

2.1.1.1. Maksete reeglid

Maksed on tellija süsteemis tähtsal kohal. Maksete näol liigub tellija süsteemist igakuiselt välja tuhandeid makseridu suurtes summades. Seega on ilmne, miks tellija leiab, et makseridade õigsus on äriliselt kriitiline.

Analüüsides süsteemi dokumentatsiooni, äriloogikat ja varasemaid süsteemi andmevigu, on võimalik välja lugeda reegleid, millele maksed vastama peavad. Reeglid, mida tuleb andmemustritega kirjeldada, on järgnevad:

1) Isiku ja teatud liiki avalduse piires ei tohi eksisteerida kahte sama perioodiga makserida, mis on mõlemad kinnitatud staatuses.

2) Andmebaasiväli *TAX_FREE_INCOME_SUM* ei tohi olla negatiivne.

3) Andmebaasiväli *NET_SUM* ei tohi olla tühi, kui makserida on kinnitatud staatuses ja väljamaksmise kuupäev on täna või minevikus.

2.1.2. Vigaste andmete tuvastamise aeg

Töö teostamiseks on vaja välja selgitada, kui tihti peab süsteem vigaseid andmeid kontrollima. Laias laastus on variante kolm: igal sisestusel kontrolli teostamine, kontrollimine teatud perioodi tagant ja kontrollimine teatud hetkel äriprotsessi ajal.

Igal andmesisestusel kontrolli teostamine annaks kõige värskema seisu andmevigadest, kuid paneks rakendusele lisakoormuse. Olenevalt tabelitest, mida vastav andmemuster puudutab, aeglustab igal sisestusel andmete kontrollimine süsteemi kasutajatel igapäevaste toimingute läbiviimist.

Äriliselt ei ole kõige värskemad andmed kriitilised, sest varasemalt kahju tekitanud vead ei ole tekkinud vahetult pärast andmete sisestamist, vaid on ilmnenud hilisemas äriprotsessis. Ühtlasi, kuna tehniliselt peab igal sisestusel kontrolli teostama läbi mingisuguse kuulaja (kas andmebaasi päästik või Hibernate'i sündmusekuulaja), siis on vaja iga muudatusega reeglitele rakendust uuesti ehitada ja tarnida. Pikas perspektiivis kulub selle lahenduse teostamisele ja ülalhoiule rohkem aega, mis tähendab tellijale suuremat rahakulu ja seetõttu ei ole igal andmesisestusel kontrolli teostamine mõistlik.

Äriliselt on teatud perioodi tagant andmete kontrollimine piisava andmevärskusega. Võttes arvesse, et süsteemis toimuvad paljud automaatprotsessid just öisel ajal, et hommikuks oleks süsteemi kasutajate jaoks kõik mahukad toimingud juba tehtud, siis on loogiline lisada öiste automaatprotsesside hulka ka perioodiline andmete kontroll. Süsteemi igaöine maksude arvutamine lõpeb tavaliselt enne kella 5t. Esimesed tellija süsteemi kasutajad alustavad süsteemis tööd kell 6. Seetõttu lepiti süsteemi analüütikute ja tellijaga kokku sooritada igaöine andmete kontroll kell 5.15.

Teatud perioodi tagant kontrollimine võimaldab kasutada olemasolevat rakenduse infrastruktuuri, kuhu on juba hetkel realiseeritud raamistik ajakava järgi sündmuste teostamiseks. See vähendab töö teostamiseks kuluvat aega, mis omakorda säästab tellijale raha.

Andmete kontrollimine vahetult enne kriitiliste äriprotsesside teostamist tõstab kindlustunnet ja tagab suurema andmete tervikluse. Näiteks, makseridade süsteemist välja saatmist viib läbi tellija raamatupidaja. Kui andmekontrolli teostada vahetult enne maksete välja saatmist, võib olla kindel, et raamatupidaja välja saadetavate maksete hulgas ei ole ebakorrektseid makseridu.

Juhtumite jaoks, kus vigaste andmete tuvastamise protsess tuleb käivitada ebakorrapärasel ajal, tuleb süsteemi administraatorile luua administraatoripaneeli nupp protsessi manuaalseks käivitamiseks.

2.1.3. Vigaste andmete ülevaade

Kui realiseeritav süsteem leiab vigaseid andmeid, siis peab süsteemi administraator neist teada saama, vead üle vaatama ja vajadusel midagi ette võtma. Potentsiaalseid lahendusi on süsteemi administraatori teavitamiseks kaks: luua kasutajaliidesesse administraatori paneeli andmete jälgimiseks vaade või saata vigaste andmete raport süsteemi administraatorile e-kirja kujul.

E-kirja kujul raporti saatmine on lihtsakoeline lahendus, mida on võrdlemisi kerge ka tehniliselt realiseerida. E-kirja eeliseks saab ühtlasi tuua märguande aspekti, st süsteemi

administraatorile tuleb kirja kujul eraldi teade, et süsteemis on tähelepanu vajavaid andmeid. Selline info edastamise meetod on praegu tellija süsteemis ka kasutusel.

E-kirja puudujäägiks võib tuua raskesti painutatava vormi. Kuna e-kiri on oma olemuselt lihtsalt tekst, siis ei ole võimalik loodavasse raportisse lisafunktsionaalsust sisse ehitada, kui selleks vajadus peaks tekkima. Näiteks, kui peaks tekkima topeltmakserida, siis oleks süsteemiadministraatoril mugav otse administraatoripaneelis nendele makseridadele klõpsata ja jõuda asjakohase infoni.

Vigaste andmete ülevaadet on ka kasutajakogemust arvesse võttes loogiline luua administraatoripaneeli, kuna peatükis 2.1.2. analüüsi käigus tekkis vajadus administraatoripaneelis vigaste andmete sektsiooni järele.

Parima tulemuse saamiseks, st süsteemi administraatorile märguande saatmiseks ja mugavaks vigade ülevaateks, on mõistlik kombineerida e-kirja saatmine administraatoripaneeli lahendusega. Vigaste andmete leidmise puhul tuleb realiseeritaval süsteemil saata e-kiri süsteemi administraatorile lakoonilise raportiga, misjärel saab administraator suunduda rakendusse ja vaadata andmed üle administraatoripaneelilt.

Üleliigsete e-kirjade vältimiseks peab olema süsteemis võimekus soovi korral vigaste andmete e-kirjade saatmine välja lülitada.

2.2. Automaatprotsessid

Varasemalt on tellija süsteemis tekkinud ärikriitilisi andmevigu seoses automaatprotsessidega. Seda on juhtunud eelkõige kahel kujul.

Üks korduvalt esinenud juhtum on automaatprotsesside suure seadusandlusest tuleneva keerukuse ja reeglite rohkuse tõttu teatud piirjuhtude automaatprotsessis korrektselt kirjeldamata jätmine. See on süsteemi ehitamisel tellija poolt sisse arvestatud kompromiss, et hoida süsteemi paindliku ja konfigureeritavana. Tänaseks on aga süsteemis andmemahud kasvanud, mis tähendab, et piirjuhte tekib aina sagedamini, mistõttu tellija peab manuaalselt sekkuma ja tuvastama võmalikud andmevead.

Andmevigade kiiremaks tuvastamiseks sobib eelmises peatükis kirjeldatud reeglimootor.

Teiseks, tellija süsteemis olemasolevate rohkete automaatprotsesside tõttu on esinenud automaatprotsesside ristumist. Näiteks, kui üks automaatprotsess kestab mingil põhjusel tavapärasest kauem ja sel ajal hakkab tööle teine samu andmeid kasutav automaatprotsess, mis eeldab, et esimene on lõpetanud, siis võivad tekkida tellija süsteemi vigased andmed, sest üks automaatprotsess kirjutab teise tulemused üle.

Kuna automaatprotsesse on tellija süsteemis palju ja nad töötavad enamasti süsteemis erinevates osades, erinevatel aegadel ja erinevate kestustega, siis ei ole mõistlik neid rangelt järjestada. Lisaks, uute automaatprotsesside lisamisel peaks alati olemasolevat järjestust muutma ja ümber tegema, mis nõuaks iga kord mahukat analüüsi ja arendust.

Seepärast soovib tellija, et süsteem kontrollib andmete õigsust, mitte niivõrd automaatprotsesside õigsust.

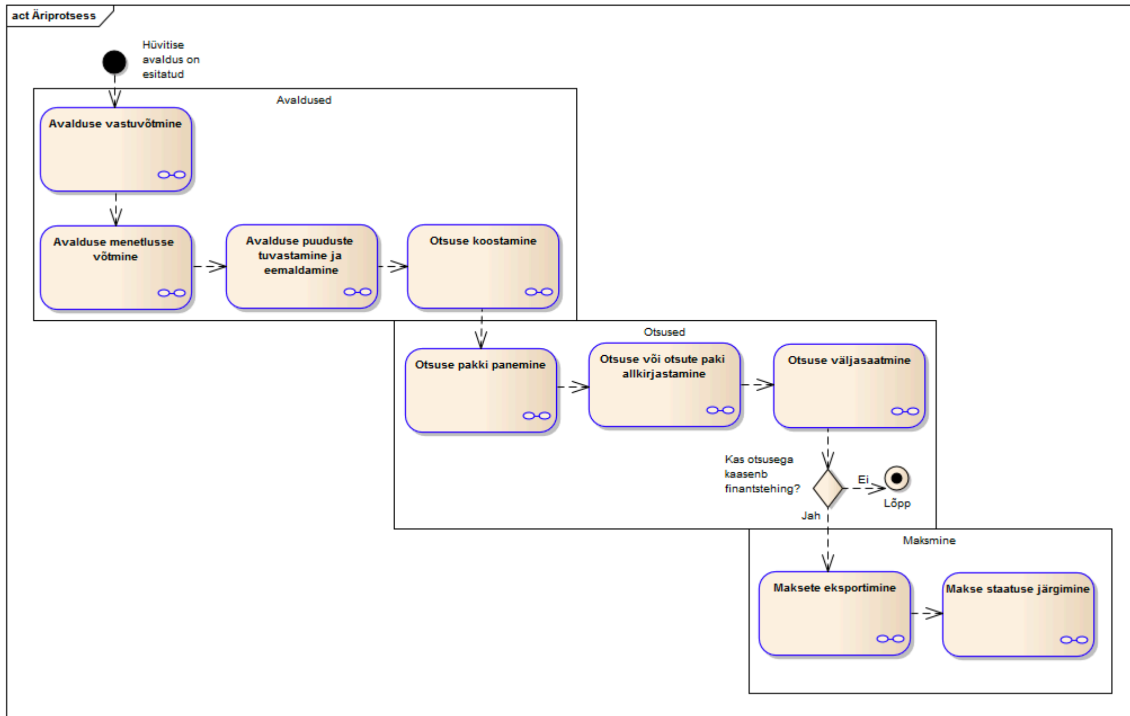
2.2.1. Avalduse lukud

Vigaste andmete tekkimise vältimiseks ristuvate automaatprotsesside tõttu soovib tellija, et kriitilisi andmeid saab “lukku panna”. Lukustamine tähendab tellija süsteemi kontekstis äriliste andmeobjektide andmemuudatuste keelamist.

Tellija süsteemi lihtustatud äriprotsessi joonisel (vt. joonis 1) on välja toodud süsteemi peamine töövoog, mille kõrgeima taseme põhiobjektiks võib pidada avaldust. Äriloogilises mõttes luuakse otsused avalduste külge ja maksed otsuste külge.

Tellija süsteemis töötavad automaatprotsessid on just avaldustega suuresti seotud. Kui avalduse infot loevad ja muudavad automaatprotsessid ja menetlejad samal ajal, siis võib süsteemi tekkida vigaseid andmeid protsesside ristumise tagajärjel.

Avalduse andmete ülekirjutamise vältimiseks peab saama tellija süsteemis avaldust lukustada, kui mõni automaatprotsess sellega parajasti toiminguid teostab. Kui avaldus on lukustatud, siis menetleja saab avalduse avamisel asjakohase teate ja ta ei saa kasutajaliideses avaldust muuta.



Joonis 1. Tellija süsteemi lihtsustatud äriprotsess.

Lisaks avaldustele peab tellija süsteemis olema võimalik lukustada ka teist ärikriitilist andmeobjekti - makseid. Maksetega seonduvalt töötab tellija süsteemis mitmeid automaatprotsesse, mistõttu on need haavataval positsioonil. Näiteks kulus teatud korral arendajapoolse vea tõttu igaöisele maksude arvutamise protsessile, mis tavaliselt kestab umbes neli tundi, terve ööpäev. See tähendab, et kõik makseridu puudutavad toimingud, mis tööpäeval tehti, kirjutati maksude arvutamise protsessi poolt üle ja selle tulemusena tekkis rohkelt vigaseid makseridu, mis põhjustas tellijale märkimisväärset rahalist kahju.

Selliste olukordade vältimiseks tuleb automaatprotsesside toimingute ajaks makserida lukustada nii, et teised protsessid või süsteemi kasutajad ei saaks makserida muuta. Kuna äriprotsessiliselt (vt. joonis 1) on maksed seotud otsusega, mis on seotud avaldusega, siis makserea lukustamiseks piisab avalduse kui kõrgeima taseme ärilise objekti lukustamisest.

Tellija soovib avalduse lukke maksude arvutamise protsessile, makseridade eksportimise protsessile ja automaatsele digiallkirjastamisprotsessile.

Lisaks soovib tellija, et kui andmeobjekt on lukustatud, siis oleks võimalik välja selgitada, milline protsess lukustamise toimingu läbi viis.

3. Tehniline analüüs

Töö eesmärkide täitmisel tuleb arvesse võtta olemasolevat tehnilist pinu ja sellele vastavalt valida tööriistad ning tehnoloogiad. See aitab kaasa süsteemi jätkusuutlikkusele.

Tellija süsteemis on kasutusel tagarakenduses Spring Boot raamistik ja programmeerimiskeeleks Java 8. Java uuendamine versioonile 11 on hetkel viimastes etappides, seega tuleb töö realiseerimisel arvesse võtta sobituvust ka uuema Java versiooniga.

Andmebaasiga suhtlemiseks on kasutusel Hibernate raamistik.

Andmebaasi mootoriks on tellija süsteemis kasutusel Oracle 12g andmebaas, mille uuendamine versioonile 19c on hetkel käimas. Pikemas perspektiivis on tellija süsteemis plaanis kasutusele võtta PostgreSQL andmebaas. Töö realiseerimisel on seega tarvis arvestada kõikide andmebaasi mootoritega.

Kasutajaliideses on tellija süsteemis kasutusel AngularJS raamistik.

3.1. Reeglimootor

Reeglimootor peab sisaldama reegleid ja äriloogikat, mille abil kirjeldatud reegleid täita. Reeglimootori realiseerimiseks on võimalik lisada tellija süsteemi mõni olemasolev teek või realiseerida tellija süsteemi vajaliku funktsionaalsusega reeglimootor ise.

Tuntumatest reeglimootori teekidest tulevad tellija süsteemi olemust silmas pidades kõne alla Drools ja OpenL Tablets. Äriloogika abstraheerimist võimaldavaid reeglimootoreid on teisigi, kuid teised reeglimootorid töötavad valdavalt ärireeglite kirjeldamisega koodi tasandil. See tähendab, et ärireeglite haldamiseks on vaja teha muudatusi koodis, mis toob endaga kaasa rakenduse uuesti ehitamise ja tarnimise, minnes vastuollu peatükis 2.1.2. analüüsitud nõuetega.

Drools on avatud lähtekoodiga ärireeglite haldamislahendus, mis võimaldab ärireeglite haldamist läbi Drools Workbench veebikeskkonna. Droolsi eeliseks võib pidada ärireeglite kirjeldamist ka koodivõõrale inimesele ja aktiivset kasutajatuge. [2]

Droolsi miinusteks on tellija süsteemi vaatest mahukad sõltuvused ja ajaliselt kulukas juurutusprotsess. Samuti on Drools tellija süsteemi ärinõuete täitmiseks liiga põhjalik, sest tellijal on vaja vaid väikest osa Droolsi pakutavast funktsionaalsusest.

OpenL Tablets on samuti avatud lähtekoodiga süsteem, mis võimaldab ärireeglite haldamist nii läbi veebi kui ka Microsoft Excel failide. See tähendab, et ka koodivõõras inimene saab kerge vaevaga ärireegleid hallata. [3]

OpenL Tabletsi negatiivsed küljed on sarnased Droolsile: süsteemisõltuvus on mahukas, süsteemi juurutamine on vaevaline ja kaasneb rohkelt kasutamata jäävat lisafunktsionaalsust.

Tähelepanu pälvib asjaolu, et nii Drools kui ka OpenL Tablets suurendaksid oluliselt tellija süsteemi jalajälge. Tellija süsteemis on vaja vaid kergendatud funktsionaalsust sellest, mida pakuvad Drools ja OpenL Tablets.

Samuti on uute lahenduste kasutuselevõtt süsteemi kasutajate, antud juhul eelkõige süsteemi administraatori, jaoks ajaliselt kulukas õppimisprotsess.

Seega on tõhusam ja mõistlikum realiseerida lihtsakoeline reeglimootori lahendus süsteemi ise. Sellega ei kaasne süsteemi lisasõltuvusi ja väiksema keerukusega süsteem on kasutaja jaoks sõbralikum. Nullist ehitatud reeglimootor võimaldab süsteemi lisada täpselt nii palju lisafunktsionaalsust, nagu tellija soovib ja ei koorma seeläbi tellija süsteemi ebavajaliku kasutamata koodiga.

3.1.1. Struktuur

Kuna reeglimootorisse peab olema võimalik kirjeldada komplekseid seoseid, siis ei ole mõistlikkuse piirides võimalik mahutada kogu infot ühte objekti, vaid tuleb kasutada objektidevahelisi seoseid.

Seega koosneb reeglimootori reegel kui objektide kogum reegliinfost, tingimustest ja tulemustest. Tingimuse objekt kirjeldab väljale kehtivaid nõudeid, reegliinfo objekt on kogum reeglitest ja tulemuse objekt kirjeldab ühe reegliinfo vigaseid andmeid.

Tingimuse objektis on kirjeldatud järgnevad väljad:

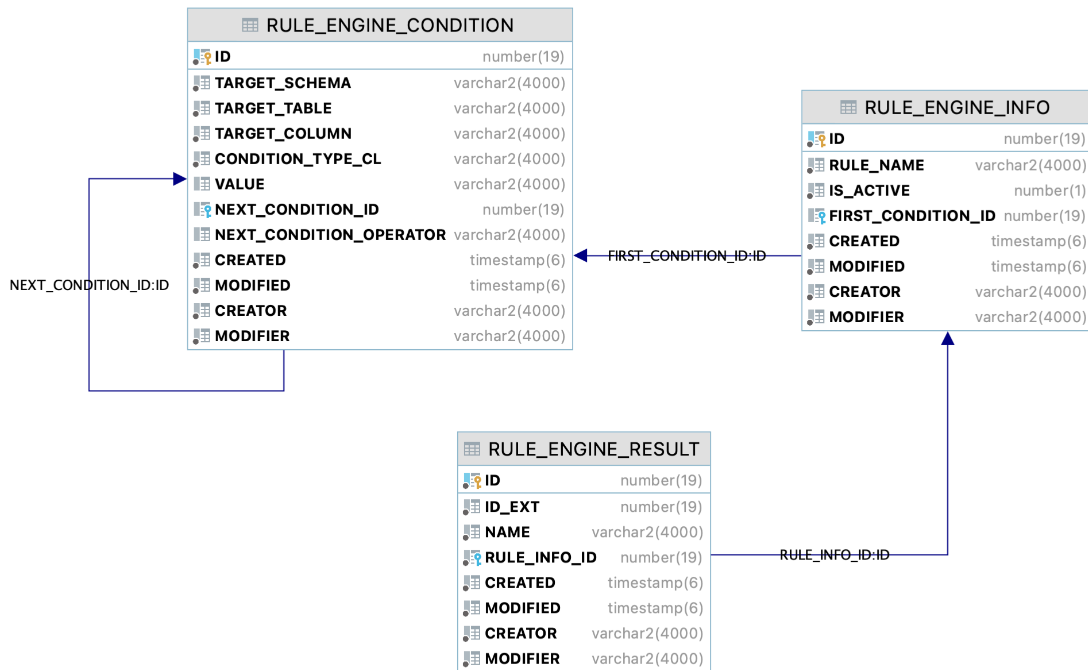
- skeemi nimi, kus asjakohane tabel asub
- tabeli nimi, kus asjakohane väli asub
- välja nimi, millele tingimus kehtib
- tingimuse tüüp, mida tuleb väljale rakendada
- tingimuse väärtus, millega tuleb välja võrrelda
- viide järgmisele tingimusele
- järgmise tingimuse loogika (kas loogiline liitmine või korrutamine)

Reegliinfo objektis on kirjeldatud järgnevad väljad:

- reeglile määratud nimi
- viide tingimusahela esimesele tingimusele
- reegli aktiivsusstaatus tõeväärtusena

Tulemuse objektis on kirjeldatud järgnevad väljad:

- vigase andmerea identifikaator
- vigase andmerea tabeli nimi
- viide reegliinfo objektile, mille raames tulemus leiti



Joonis 2. Kuvatõmmis oletatavast andmemudeli struktuurist.

3.1.2. Hoiustamine

Vigaste andmete mustreid kirjeldavaid reegleid on vaja hoiustada. Võttes arvesse tellija süsteemi struktuuri, on selleks kaks varianti: reeglite hoiustamine koodis ja reeglite hoiustamine andmebaasis.

Koodis ehk rakenduse mälus reeglite hoiustamine vähendab töö keerukust ja parandab protsessi jõudlust, kuid toob endaga kaasa märkimisväärseid miinuseid:

- Reeglite muutmiseks on vaja rakendus uuesti ehitada ja tarnida.

- Tellija süsteemi administraator ei saa reegleid ise muuta.

Reeglite hoiustamine andmebaasis nõuab sobiliku andmebaasiskeemi loomist. See lisab küll tööle keerukust, kuid on pikemas perspektiivis jätkusuutlikum ja kasulikum:

- Reeglite muutmiseks ei pea rakendust seiskama.

- Süsteemi administraator saab andmebaasist reegleid näha ja vajadusel ka lisada.

- Reeglite kuvamine süsteemi kasutajale on hõlpsam.

Kuna uue rakenduse versiooni tarnimine on tellija infrastruktuuri silmas pidades aeganõudev protsess ja süsteemi administraator soovib süsteemis toimuvaga kursis olla, siis on mõistlik töö käigus realiseeritava süsteemi andmemustreid hoiustada andmebaasi tasandil.

Ühtlasi on vaja hoiustada ka andmekontrolli tulemusi. Kui vigaste andmete üle vaatamiseks piisaks vaid e-kirja lahendusest, siis ei oleks otseselt vaja tulemusi püsivalt hoiustada, vaid võiks tulemused otse mälust e-kirjana teele panna ja äriprotsess lõpetada. Kuna aga süsteemi administraatoril peab olema tulemustest ülevaade läbi kasutajaliidese, siis tuleb andmekontrolli tulemused samuti andmebaasi salvestada.

3.1.3. Protsessi käivitamine

Ärianalüüsi kohaselt tuleb andmete kontroll käivitada kolmel viisil: vahetult enne kriitilist äriprotsessi, periooditi kord ööpäevas ja administraatoripaneelist.

Kuna hetkel on tellija poolt prioriteetne äriprotsess, enne mida on vaja andmeid kontrollida, maksete väljasaatmine, siis selle teostamine ei oma suurt keerukust. Kui vastav äriloogika on teenusena realiseeritud, siis piisab vaid koodis teenuse välja kutsumisest enne maksete väljasaatmist ja vigade esinemise korral vastavaid makseid mitte välja saata. Näiteks, kui raamatupidaja proovib välja saata makseridu, mille hulgas on makserida identifikaatoriga 1 ja vahetult enne makseridade välja saatmist käivitatud andmete kontrollimine tuvastas, et makserida identifikaatoriga 1 on vigane, siis seda makserida välja ei saadeta.

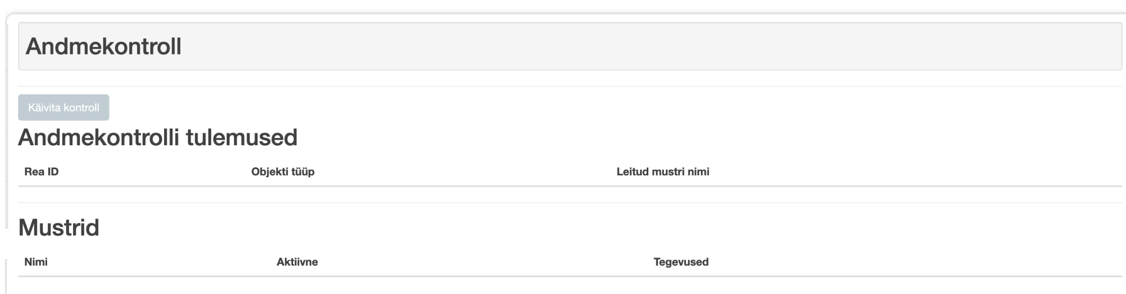
Perioodit kontrollimist on võimalik teostada läbi plaaniliste protsesside teenusklassi, mis on praegu juba süsteemis olemas. Teenusklassi tuleb lisada uus funktsioon, mis kutsub välja andmete kontrollimise teenust. Kuna perioodit kontrollimine peab toimuma kord ööpäevas kell 5.15 hommikul, siis ajalist määrust kirjeldav cron lause peab olema 0 15 5 * * *.

Administraatoripaneelist välja kutsumine nõuab kasutajaliidese arendust. Täpsemalt on vaja luua administraatoripaneeli andmekontrollinimeline alampaneel ja nupp, mis kutsub välja andmekontrolli teenust.

Kui protsess jõuab lõpuni ja leiab oma töö tulemusena vigaseid andmeid, siis tuleb saata administraatorile e-mail lühiülevaatega vigastest andmetest.

3.1.4. Administraatori ülevaade

Eelmises peatükis mainitud administraatoripaneeli tuleb luua ka administraatorile ülevaade süsteemis olevatest reeglitest ja vastavate reeglite poolt leitud vigastest andmetest.



Joonis 3. Visand planeeritavast administraatoripaneelist.

Visandatud administraatoripaneelist (vt. joonis 3) on näha, et sellise kasutajaliidese disaini puhul on administraatoril ülevaade vigastest andmetest, mis erinevad mustrid leidnud on. Lisaks tuleb luua Rea ID tulbale klõpsatav link, mis võimaldab administraatoril liikuda andmeobjekti vaate lehele ja probleemi lähemalt uurida.

Reiko-Rainer Reinup
Andmekontrolli tulemused 15.04.2021
To: Reiko-Rainer Reinup

Andmekontrollis kasutatud mustrid:

- Mustri nimi 1.

Vigaseid andmeridu 5

- Mustri nimi 2.

Vigaseid andmeridu 0

- Mustri nimi 3.

Vigaseid andmeridu 1

Täpsem info: www.example.com/admin/andmevead

Joonis 4. Administraatorile saadetav e-kirja kuju.

Tellija süsteemi tuleb lisada parameeter, mille järgi otsustatakse, kas vigaste andmete e-kiri saadetakse süsteemi administraatorile või mitte. Parameetrite tabel ja üldine loogika on hetkel tellija süsteemis olemas.

Adminstraatorile saadetav e-kiri andmekontrolli tulemustest ei pea olema ülemäära põhjalik, kuna vajalikke tegevusi teostab administraator kasutajaliideses. Seega võib kirja sisu koosneda vaid põhiinfost: andmekontrollis kasutatud mustrite nimed, mitu vigast kirjet vastava mustriga leiti ja lingist administraatoripaneeli (vt. joonis 4).

3.2. Automaatprotsessid

Automaatprotsesside ristumise leevendamiseks on vaja luua tellija süsteemi avalduste ja makseridade lukustamise võimekus. Selleks on mitmeid tehnilisi lahendusi, kuid analüüsi ja märkimist väärivad eelkõige kaks: Hibernate raamistiku andmelukud ja andmemudelil põhinevad loogilised lukud.

Hibernate raamistiku andmelukkude kasutamine pakub mitmeid eeliseid. Hibernate luku kasutuselevõtt on tehniliselt lihtne. Vaja oleks vaid lisada lukustatava objekti viimase uuendamise ajatempli andmebaasiväljale Hibernate annotatsioon @Version ning äriliselt kriitilisse kohta täpsustada, kuidas andmeid lukustada. [4]

Lisaks võimaldab Hibernate täpsustada andmete lukustamistüüpi: optimistlik ja pessimistlik. Kasutades optimistlikku andmelukku, on teistel protsessidel endiselt võimalik lukustatud andmeid lugeda, kuid andmeid muuta ei ole võimalik enne, kui algne protsess on andmed vabastanud. Kui teine protsess proovib andmeid muuta, siis viskab Hibernate OptimisticLockException erindi. Pessimistliku andmeluku puhul ei ole teistel protsessidel võimalik lisaks andmete muutmisele ka lukustatud andmeid lugeda. [4] [5]

Kolmas suur eelis, mida Hibernate lukkude kasutamine võimaldab, on lukkude loomine mistahes andmereale. See tähendab, et kui tulevikus peaks tekkima vajadus panna lukku midagi muud peale avalduste ja makseridade, siis ei vaja see lisaarendust. [4]

Küll aga on Hibernate luku kasutamine tellija süsteemis vastunäidustatud süsteemist tulenevate iseärasuste tõttu. Täpsemalt, kuna tellija süsteem koosneb mitmest moodulist, mis omavahel suhtlevad, siis võib ühes moodulis lukku pandud andmetega teises moodulis ärilisi toiminguid teha. Näiteks, kui otsuste moodul küsib maksete mooduli käest makseridade kohta infot, aga makseread on lukus, siis maksete moodul tagastaks Hibernate luku korral veateate. Sellel hetkel on aga otsus süsteemi juba loodud ja selle protsessi tagasi pööramine on tellija süsteemis keerukas, sest otsus kui äriline objekt koosneb paljudest andmeobjektidest ja allkirjastatud dokumentidest. Kuna otsus on keeruline äriobjekt, siis selle tagasi võtmine ei ole äriliselt mõistlik tegevus.

Selle loogika muutmine nõuab väga mahukat analüüsi ja arendust ning tellija sellest hetkel huvitatud ei ole.

Andmemudelil põhinevad loogilised lukud tähendaksid andmeobjektile lukustatud staatuse välja lisamist. Kui andmeobjektil oleks väli, mis viitab rea andmeobjekti lukustatusele, siis saab äriliselt kriitilistes kohtades kontrollida välja väärtust ning vastavalt sellele mingi äriprotsessiga edasi minna või mitte. Andmeobjektile luku väljade lisamine ei ole tehniliselt kuigi keerukas, kuid sellega kaasnevad teised negatiivsed aspektid.

Erinevalt Hibernate'i lukusüsteemist peab igale andmeobjektile lisama lukustaatuse välja ja lukustamismeetodid eraldi. Kuna hetkel on tellija huvides avalduse kui kõrgeima taseme ärilise objekti lukustamine, siis ei ole see suur lisatöö, kuid tuleviku lukustamisvajaduste ja üldise jätkusuutlikkuse koha pealt pälvib siiski tähelepanu.

Lisaks ei paku andmeobjekti väljapõhine lukk erinevaid lukustamistüüpe (optimistlik ja pessimistlik). Väiksem paindlikkus ja piirangud on oma olemuses negatiivsed, kuid tellija ärinõuete täitmise jaoks ei ole erinevad lukustamistüübid hädavajalikud.

Kuigi andmemudelil põhinev loogiline lukk on võimekuselt nõrgem, kui Hibernate'i poolt pakutavad võimalused, siis süsteemi eripärasid, tellija soove ja ärinõudeid silmas pidades on siiski õigem kasutada andmemudelil põhinevat loogilist lukku.

3.2.1. Luku struktuur

Luku struktuuri välja selgitamisel on vaja arvesse võtta, et lukk peab sisaldama endas infot nii lukustatuse staatuse kohta kui ka luku lisanud protsessi kohta.

Andmemudelil põhineva luku jaoks on vaja lisada lukustamist vajavale andmeobjektile uus väli - *locked*. Selle välja andmetüüp võib olla ärilist eesmärki täites erinev. Täpsemalt, välja tüübina võib kaaluda tõeväärtust kui ka ajatemplit. Ajatempel on antud kontekstis eelistatum variant, sest selle abil on võimalik välja selgitada, kui kaua on andmeobjekt lukus olnud. Kui väljal puudub väärtus, siis on andmeobjekt lukustamata ja väärtuse olemasolul on andmeobjekt lukustatud.

Lukustaja info lisamine käib samuti andmevälja kaudu. See tähendab, et lukustatavale andmeobjektile on vaja lisada tekstiväli *lockType* ja koodis andmeobjekti lukustades luku tüüp parameetrina kaasa anda.

4. Realiseerimine

Töö realiseerimisel on vaja arvesse võtta süsteemi jätkusuutlikkust ja pidada kinni puhta koodi põhimõtetest.

Peamise tööriistana süsteemi realiseerimisel on kasutusel IntelliJ IDEA 2020.3.2.

4.1. Tagarakendus

Tellija süsteemis on tagarakenduses kasutusel kontroller-teenus-hoidla muster, mis tähendab üldistatult, et igal äriobjektil on süsteemis temaga seonduvate toimingute teenus. Teenus kasutab andmebaasile ligipääsuks hoidlat ning teenusele ligipääsuks kasutatakse vastava äriobjekti kontrollerit [6]. Projekti koodi loetavuse huvides on mõistlik seda mustrit uute arendustega jätkata.

Moodulitevahelisel suhtlusel on hetkel tellija süsteemis kasutusel Retrofit2. Retrofit2 võimaldab teostada tüübikindlaid HTTP päringuid läbi Java keele [7]. Kui realiseerimise käigus on vaja teostada HTTP päringuid ühelt moodulilt teisele, siis on selle jaoks vaja tellija süsteemis lisada pöörduspunkt vastava mooduli Retrofit2 liidesesse.

4.1.1. Reeglimootor

Reeglimootori realiseerimist alustatakse andmebaasitabelite loomisest vastavalt analüüsitud andmemudelile. Tingimuse, reegliinfo ja tulemuste tabeli nimedeks määratakse vastavalt *rule_engine_condition*, *rule_engine_info* ja *rule_engine_result*. Loodud andmemudel täidetakse algandmetega vastavalt 2.1.1.1. peatükis kirjeldatud maksete reeglitele.

Andmebaasiobjektide Java koodis kasutamiseks luuakse vastavad domeenimudelid ja hoidlad. Süsteemi loodi klassid *RuleEngineCondition*, *RuleEngineInfo, RuleEngineResult* ja neile vastavad Sprint Booti poolt pakutavad *CrudRepository* liidese teostusklassid. Kõik loodud domeenimudeli klassid laiendavad tellija süsteemis juba olemasolevat klassi *DomainBase,* milles on kõikide olemite ühised väljad.

Reeglimootori äriloogika kirjeldamiseks luuakse teenusklass nimega *RuleEngineService*. Teenusklassi realiseeritakse äriloogika, mis loeb andmebaasist sisse aktiivses staatuses reeglid ja loob reegli kontrollimiseks vastava DML lause. DML lause käivitatakse andmebaasis kasutades *JdbcTemplate* objekti (vt. joonis 5 ja 6).

```
public void startDataCheckProcess () {
    LOG.info("Data checking process started");
    Iterable<RuleEngineInfo> allRules =
                          ruleEngineInfoRepository.findAll();
    List<RuleEngineResult> newRuleEngineResults = new ArrayList<>();
    allRules.forEach(rule -> {
        if (rule.getIsActive()) {
            List<RuleEngineResult> ruleEngineResults =
                                          checkRule(rule);
            ruleEngineResults.forEach(ruleEngineResult -> {
                if (!ruleEngineResultRepository.exists(
                        byIdExtNameRuleId(ruleEngineResult))) {
                    newRuleEngineResults.add(ruleEngineResult);
                }});
        } else {
            LOG.info("Ignoring inactive rule: ID({}) {}",
                        rule.getId(), rule.getRuleName());
        }
    });
    emailService.sendRuleEngineResult(newRuleEngineResults);
    ruleEngineResultRepository.saveAll(newRuleEngineResults);
    LOG.info("Data checking process completed");
}
```

Joonis 5. Koodilõik andmekontrolli käivitamise meetodist.

```
private List<RuleEngineResult> checkRule(RuleEngineInfo rule) {
    LOG.info("Checking rule - ID {}: {}", rule.getId(),
                                    rule.getRuleName());
    RuleEngineCondition firstCondition = rule.getFirstCondition();
    String sql = buildSql(firstCondition);
    List<ResultId> queryResult = jdbcTemplate.query(sql,
                  new BeanPropertyRowMapper<>(ResultId.class));
    LOG.info("Found {} conflicting rows", queryResult.size());
    return queryResult.stream()
            .map(resultId -> extractRuleEngineResult(resultId,
                        firstCondition.getTargetTable(), rule))
            .collect(Collectors.toList());
}
```

Joonis 6. Koodilõik DML lause käivitamise ja tulemuste tagastamise meetodist.

Kui lause tagastab andmeridu, siis need read salvestatakse *rule_engine_result* tabelisse, pidades silmas, et kui tabelis on juba olemas sama *id_ext*, *name* ja *rule_info_id* väärtustega rida, siis rida ei salvestata, sest see rida on juba sama reegli poolt antud tabelist leitud (vt. joonis 5).

Tellija süsteemi parameetrite hulka lisatakse uus parameeter, mille abil on võimalik e-kirja saatmist sisse ja välja lülitada. Parameetri väärtuseks on vaikimisi süsteemi administraatori e-posti aadress. Juhul, kui parameeter on väärtustatud, siis koostatakse tagastatud andmeridade põhjal ka e-kiri, mis saadetakse parameetri väärtuses olevale aadressile (vt. joonis 7).

```
public void sendRuleEngineResult(
            List<RuleEngineResult> ruleEngineResults) {
    ParameterInfo emailParam = null;
    try {
        emailParam = adminModuleClient
            .getParameterByCode(SEND_RULE_ENGINE_RESULTS.name());
    } catch (Exception e){
        LOG.error("Unable to send email! Parameter '{}' is not found",
            SEND_RULE_ENGINE_RESULTS.name());
    }
    if (emailParam != null) {
        String defaultRecipient = emailParam.getValue();

        EmailMessage emailMessage = createAndSetCommonEmailMesssage(
                  defaultRecipient, defaultRecipient,
                  String.format("Andmekontrolli tulemused %s (%s)",
                  LocalDate.now(), activeProfile),
                  createRuleEngineResultsBody(ruleEngineResults));

        emailSender.sendEmail(emailMessage);
    }
}
```

Joonis 7. Koodilõik e-kirja saatmise meetodist parameetri väärtuse põhjal.

Selle jaoks luuakse tellija süsteemis juba olemasolevasse teenusklassi *EmailService* uus meetod nimega *sendRuleEngineResults*, mis võtab parameetrina vastu *RuleEngineResult* objektid ja saadab e-kirjana kokkuvõtte vigastest andmetest. E-kirja koostamisel lähtutakse peatükis 3.1.4. analüüsitud e-kirja kujust (vt. joonis 7 ja 8).

```
private String createRuleEngineResultsBody(
            List<RuleEngineResult> ruleEngineResults) {
    StringBuilder sb = new StringBuilder();
    Map<RuleEngineInfo, List<RuleEngineResult>> resultsByRuleInfo =
      ruleEngineResults.stream()
       .collect(Collectors.groupingBy(RuleEngineResult::getRuleInfo));
    sb.append("Andmekontrollis kasutatud mustrid:").append(NEW_LINE);
    resultsByRuleInfo.forEach((k, v) -> {
        sb.append("- “).append(k.getRuleName())
                        .append(NEW_LINE);
        sb.append(String.format("Vigaseid andmeridu %s", v.size()))
                        .append(NEW_LINE);
    });
    sb.append(NEW_LINE);
    sb.append(String.format("Täpsem ülevaade: %s/admin/request/
                ruleEngine/rule_engine/list", BASE_URL));
    return sb.toString();
}
```

Joonis 8. Koodilõik e-kirja sisu koostamise meetodist tulemuste põhjal.

Äriloogika käivitamiseks luuakse kontrollerklass *RuleEngineController*. Kontroller võtab vastu HTTP päringuid. Vastavate pöörduspunktide poole pöördudes on võimalik andmekontrolli protsessi käivitada, näha varasemaid andmekontrolli tulemusi ja muuta olemasolevaid reegliinfo objekte. Kontrollerklass on vajalik, et teenusklassiga oleks võimalik suhelda väljastpoolt rakendust ennast, sest teenusele on vaja ligi pääseda veebist ja teistest moodulitest.

Rakendusesiseselt on võimalik teenusklass sõltuvusena sisse tuua mistahes teise klassi, kasutades Springi poolt pakutavat *@Autowired* annotatsiooni [8].

Andmekontrolli teenusklass tuuakse sõltuvusena sisse tellija süsteemis olemasolevasse *CronService* ja *PaymentExportService* teenusklassi. *CronService* klassis kutsutakse analüüsi kohaselt välja andmekontrolli käivitamise meetodit perioodilt. *PaymentExportService* klassis kutsutakse andmekontrolli käivitamise meetodit välja enne maksete süsteemist välja saatmist.

4.1.2. Automaatprotsessid

Ristuvate automaatprotsesside leevendamiseks välja töötatud lukkude realiseerimisega alustatakse luku struktuuri loomisega avalduse objektile. Avalduste tabelisse luuakse

andmebaasi uued väljad *locked* ajatempli andmetüübiga ja *lock_type_cl* teksti andmetüübiga.

Avalduse lukustamise protsessi võimalikult mugavaks tegemiseks luuakse lukustamis- ja lahtilukustamismeetodid otse domeenimudelit kirjeldavasse klassi (vt. joonis 9).

```
public void lock(String lockType) {
    locked = LocalDateTime.now();
    lockTypeCl = lockType;
}

public void unlock() {
    locked = null;
    lockTypeCl = null;
}
```

Joonis 9. Koodilõik avalduse domeeniobjekti klassi meetoditest.

Nende meetodite lisamisega on tellija süsteemil funktsionaalsus avaldusi lukustada ja lahti lukustada.

Kuna lukustamist vajavaid avaldusi tekib tellija süsteemi enamasti suurtes kogustes korraga, siis luuakse avalduste teenusklassi *ApplicationService* meetodid massiliseks lukustamiseks ja lahti lukustamiseks (vt. joonis 10).

```
public void setLock(boolean lock, String lockType,
List<Long> applicationIdList) {

      Iterable<Application>applications=
            getApplicationsByIds(applicationIdList);
      if (lock) {
            applications.forEach(application ->
                  application.lock(lockType));
      } else {
            applications.forEach(Application::unlock);
      }
      applicationRepository.saveAll(applications);
}
```

Joonis 10. Koodilõik avalduse teenusklassi masslukustamismeetodist.

Teenusele ligipääsu tagamiseks teistele moodulitele luuakse kontrollerklassi *ApplicationController* uus meetod, millega on võimalik läbi HTTP päringu avalduse lukuga toiminguid teostada. Ühtlasi lisatakse pöörduspunkt Retrofit2 liidesesse.

Avalduse lukustamise loogika lisatakse enne ärikriitilisi protsesse ja lahti lukustamise loogika siis, kui äriprotsessid on lõpetanud. Nagu välja toodud peatükis 2.2.1., sisestatakse lukustamiseks ja lahtilukustamiseks vajalik loogika maksude arvutamise protsessile (vt. joonis 11), makseridade eksportimise protsessile (vt. joonis 12) ja digiallkirjastamise protsessile (vt. joonis 13) . Lisaks, enne töö alustamist kontrollitakse protsessis, kas mingitel avaldusel on juba lukk peal. Kui on, siis vastavat avaldust või avaldusega seotud muud äriobjekti protsessi ei kaasata.

```
List<Long> applicationIds = Streams.stream(tkhPayments)
            .map(Payment::getApplicationIdExt)
            .distinct()
            .collect(Collectors.toList());

LOG.info("Locking applications {}", applicationIds);
mainModuleClient.setLock(true,
"APPLICATION_LOCK_TAX_CALCULATION", applicationIds);
```

Joonis 11. Koodilõik luku lisamisest maksude arvutamise protsessile.

```
if (!payments.isEmpty()) {
      List<Long> applicationsToLock = Streams.stream(
      info.getpaymentsPreview().getApplicationIdList())
      .distinct()
      .collect(Collectors.toList());

      LOG.info("Locking applications {}", applicationsToLock);
      mainModuleClient.setLock(true,
            "APPLICATION_LOCK_NAV_EXPORT", applicationsToLock);
}
```

Joonis 12. Koodilõik luku lisamisest makseridade eksportimise protsessile.

```
List<Long> applicationExtList = applicationDecisionList.stream()
.map(ApplicationDecision::getApplicationExt)
.collect(Collectors.toList());

LOG.info("Locking applications {}", applicationExtList);
mainModuleClient.setLock(true,
"APPLICATION_LOCK_DIGI_STAMP", applicationExtList);
```

Joonis 13. Koodilõik luku lisamisest digiallkirjastamise protsessile.

Kui protsess on oma töö lõpetanud, siis võtab protsess avaldustelt lukud maha kasutades sama meetodit, mida lukustamiselgi, kuid esimene argument *setLock* meetodil on *false*.

4.2. Kasutajaliides

Tellija süsteemi kasutajaliideses on kasutusel AngularJS versioon 1.6.3.

4.2.1. Reeglimootor

Reeglimootori funktsionaalsuse raames luuakse kasutajaliidesesse uus vaade administraatoripaneeli, kus on ülevaade leitud vigastest andmetest, võimalus andmekontrolli manuaalselt käivitada ja olemasolevaid reegleid muuta.

Uue vaate realiseerimist alustatakse AngularJS andmeallikate loomisest, mis võimaldavad suhelda tagarakendusse loodud *endpoint*'idega kasutades AngularJS *$resource* liidest. Liides võimaldab mugavalt teostada ja käsitleda HTTP päringuid. [9]

Tellija süsteemi lisatakse kõikide andmemustrite leidmise, individuaalse andmemustri leidmise ja muutmise, andmekontrolli käivitamise ning andmekontrolli tulemuste leidmise andmeallikad (vt. joonis 14).

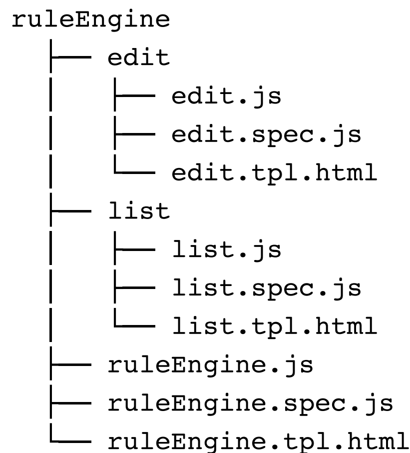
```
angular.module('shared.factory.main.ruleEnginePattern', [
      'ngResource'
])

.factory('RuleEnginePattern', function($resource) {
{     return $resource('/main/rule_engine/patterns/:id', {id: '@id'},
            get: {method: 'GET', isArray: false},
            save: {method: 'POST', isArray: false}
      });
});
```

Joonis 14. Näide AngularJS andmeallikast.

Seejärel luuakse vaade visandi põhjal (vt. joonis 3). Tellija süsteemi kasutajaliidese koodis juba väljakujunenud struktuuri järgides luuakse kataloog *ruleEngine* ja kõik sellega seonduvad JavaScripti kontrollerid ning neile vastavad HTML mallid. Kontrollerid ja mallid luuakse nii andmete vaatamiseks kui ka andmete muutmiseks (vt. joonis 15). Failid *js* laiendiga on JavaScripti kontrollerid, kus on reeglina kogu vastav

äriloogika. Failid *tpl.html* laiendiga on HTML mallid, millest luuakse lõppkasutajale nähtav dokument. Failid *spec.js* laiendiga on kasutajaliidesepoolsed testid.



```
ruleEngine
├── edit
│   ├── edit.js
│   ├── edit.spec.js
│   └── edit.tpl.html
├── list
│   ├── list.js
│   ├── list.spec.js
│   └── list.tpl.html
├── ruleEngine.js
├── ruleEngine.spec.js
└── ruleEngine.tpl.html
```

Joonis 15. Reeglimootori failide puu.

List ja *edit* nimelistes kataloogides on vastavalt andmete kuvamise ja muutmisega seonduvad kontrollerid ja mallid. Ülemkataloogis paiknev *ruleEngine.tpl.html* mall ja *ruleEngine.js* kontroller ei sisalda endas äriloogikat, vaid tegelevad ainult navigeerimisega (vt. joonis 16).

```
angular.module('admin.request.ruleEngine', [
    'admin.request.ruleEngine.list'
])
.config(function($stateProvider) {
    $stateProvider.state('admin.request.ruleEngine', {
        url: '/ruleEngine',
        controller: 'RuleEngineCtrl',
        templateUrl:
'admin/request/ruleEngine/ruleEngine.tpl.html',
        redirectTo: 'admin.request.ruleEngine.list'
    });
})
.controller('RuleEngineCtrl', function RuleEngineController() {
});
```

Joonis 16. ruleEngine.js faili sisu.

Navigeerimise eest vastutab tellija süsteemi kasutajaliideses UI-Router pakk, mis on AngularJS raamistikus mitteametlik standard navigeerimiseks ja ümber suunamiseks projektisiseste vaadete vahel [10]. Tänu UI-Router pakile koosneb *ruleEngine.tpl.html* ainult ühest tühjast atribuutideta HTML elemendist: *ui-view*.

Andmekontrolli vaatekontrolleris (*list.js*) käivitatakse päringud andmekontrolli tulemuste ja olemasolevate mustrite leidmiseks, et neid kasutajale tabelina kuvada. Päringute vastused salvestatakse kasutades AngularJS *$scope* muutujat, mis käitub ühenduslülina kontrolleri ja malli vahel [11]. Seeläbi on võimalik päringu vastusest tekitatud tabelikonfiguratsiooni mallis kasutada (vt. joonis 17 ja 18).

```
$scope.patternsTableConfig = {
      list: function() {
            return RuleEngineResults.get();
      }
};
```

Joonis 17. Koodilõik päringu tulemuse salvestamisest $scope muutujasse.

Olemasolevate andmemustrite muutmiseks luuakse vastava tabeli viimasesse tulpa nupp andmete muutmiskuvale navigeerimiseks. See nupp kasutab eelnevalt mainitud UI-Routeri funktsionaalsust, navigeerides muutmiskuvale muutmiskontrolleris kirjeldatud suunaviite abil. Korrektse andmemustri kuvamise jaoks antakse argumendina kaasa ka muutmist vajava andmemustri identifikaator (vt. joonis 18).

```
<paging-table config="patternsTableConfig">
...
      <td data-title="'admin.request.ruleEngine.actions' | translate">
            <action-select actions="actions" callback-item=“row”>
            </action-select>
      </td>
...
</paging-table>
```

Joonis 18. Koodilõik tabeli genereerimisest.

Muutmiskontrolleris päritakse tagarakendusest viite põhjal andmemustri objekt. Vaatele luuakse vorm, mille läbi on võimalik muuta andmemustri nime ja aktiivsusstaatust. Vormi sisestamise korral salvestatakse muudetud andmemuster andmebaasi ja kasutaja navigeeritakse tagasi andmekontrolli administraatoripaneeli üldvaatele.

Andmekontrolli adminipaneeli üldvaatele lisatakse nupp, mis käivitab andmekontrolli. Andmekontrolli lõpuni jõudmisel värskendatakse lehte, et tabelites kajastuksid viimase andmekontrolli tulemused (vt. joonis 19).

```
$scope.start = function () {
      RuleEngineProcess.start(
            self.ruleEngineProcessSuccess,
            self.ruleEngineProcessFail);
};

self.ruleEngineProcessSuccess = function () {
      $rootScope.showSuccessAlert(
            'admin.request.ruleEngine.success');
      $state.reload();
};

self.ruleEngineProcessFail = function () {
      $rootScope.showFailAlert(
            'admin.request.ruleEngine.error');
};
```

Joonis 19. Koodilõik andmekontrolli käivitamisest.

Adminipaneeli lisatakse andmekontrolli sakk, millele klõpsates navigeeritakse kasutaja andmekontrolli vaatele.

4.2.2. Automaatprotsessid

Avalduse detailvaatele navigeerides teeb süsteem päringu tagarakendusse vastava avalduse kohta. Peatükis 4.1.2. lisatud luku info väljad jõuavad selle päringuga kasutajaliidesse, seega piisab lukustatuse staatuse kontrollimiseks vaid *locked* välja väärtusest.

Kasutajaliideses kuvatakse süsteemi kasutajale avalduse detailvaates teade, et avaldus on lukustatud ja hetkel rohkem toiminguid teha ei saa. Selle jaoks lisatakse avalduse detailvaate HTML faili veateade, mida kuvatakse, kui avaldus on lukustatud (vt. joonis 20 ja 21).

```
<div uib-alert ng-show="application.isLocked"
ng-class="'alert-danger'">
      {{'Avaldus on lukus'}}
</div>
```

Joonis 20. Koodilõik avalduse lukustatuse teate kuvamiseks.

Joonis 21. Kuvatõmmis lukustatud avalduse teatest ja mitteaktiivsetest nuppudest.

Tellija süsteemis on avalduse detailvaadet juhtiva JavaScript kontrolleri koodis meetod nimega *editApplicationIsAllowed*, mis teeb kindlaks, kas süsteemi kasutajal on lubatud avaldusel muudatusi teha. Kui meetod tagastab negatiivse tõeväärtuse, siis teeb see avaldusel samu äriloogilisi samme, mida on vaja ka teha avalduse lukus oleku puhul (vt. joonis 21). Seega lisatakse meetodisse avalduse lukustatuse staatuse kontrollimine (vt. joonis 22).

```
$scope.editApplicationIsAllowed = function() {
      return allowedStatuses.indexOf($scope.application.statusCode)
            >= 0 &&
            !$scope.application.locked &&
            !$scope.application.hasAutomaticDecisionProcess;
};
```

Joonis 22. Meetod avalduse muutmise lubamise kontrollimiseks.

4.3. Testimine

Kõiki tarkvaraarendusi on vaja testida, et veenduda nende õigsuses. Töös teostatakse verifitseerimist läbi automaattestide kui ka käsitsi läbi kasutajaliidese.

4.3.1. Automaattestid

Automaattestide realiseerimisel on mõistlik kasutada läbivtestimise (*end-to-end testing*) meetodit, kuna seeläbi testitakse kogu äriprotsess algusest lõpuni [12]. Antud kontekstis tähendab see tagarakenduse äriloogika testimist alates päringu saatmisest kontrollerisse kuni andmebaasikihi ja teiste teenuste poole pöördumiseni.

Kuna testimist alustatakse kontrollerisse päringu saatmisest, siis tuleb automaattestis kasutusele võtta ka REST Assured teek, millega on võimalik saata Java testides mugavalt HTTP päringuid ja verifitseerida vastuseid [13]. Sellist funktsionaalsust pakuvad ka teised teegid, kuid kuna REST Assured on tellija süsteemis praegu juba kasutusel, siis ei ole mõistlik uusi sõltuvusi lisada, vaid kasutada olemasolevaid.

Kuna nii reeglimootori kui ka lukkude äriloogika hõlmab endas rohket andmebaasi kasutust, siis on mõistlik kasutada automaattestimiseks Testcontainers teeki.

Testcontainers on väikse jalajäljega Java teek, mis võimaldab luua ühekordselt kasutatavaid andmebaasi eksemplare. Lisaks sobitub Testcontainers tellija süsteemis juba kasutusel oleva testimisraamistikuga (JUnit) hästi kokku. [14]

Testcontainers teegi kasutuselevõtuks lisatakse vajalikud sõltuvused projekti sõltuvuste nimekirja. Seejärel luuakse testklass *RuleEngineControllerTest*, mis annoteeritakse testi käivitamiseks vajalike annotatsioonidega (vt. joonis 23).

```
@Testcontainers
@ActiveProfiles("unittest")
@ContextConfiguration(initializers =
RuleEngineControllerTest.Initializer.class)
@SpringBootTest(classes = {MainApplication.class},
webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@MockBeans(value = {@MockBean(DeletedEntitySettings.class)})
public class RuleEngineControllerTest extends RestAssuredUtil {
```

Joonis 23. Testklassi deklaratsioon koos annotatsioonidega.

Testcontainers ühekordse andmebaasiobjekti loomiseks luuakse testklassi staatiline muutuja *postgres* annoteerides selle Testcontainers annotatsiooniga *@Container*. Selle muutuja initsialiseerimisel luuakse mällu PostgreSQL andmebaasi eksemplar koos baasskriptis (vt Lisa 2) toodud algseisu ja -andmetega.

Seejärel luuakse testklassi initsialiseerija, mis määrab käivitatava testi andmeallika äsja loodud PostgreSQL eksemplari suunas (vt. joonis 24).

```
@Container
public static PostgreSQLContainer pg = new PostgreSQLContainer<>
      (TestContainerConfiguration.POSTGRESQL_VERSION)
     .withDatabaseName(TestContainerConf.POSTGRESQL_DATABASE_NAME)
     .withUsername(TestContainerConfiguration.POSTGRESQL_USER_NAME)
     .withPassword(TestContainerConfiguration.POSTGRESQL_PASSWORD)
     .withInitScript("tmp/test_init.sql")
     .withCommand("postgres -c max_connections=50");

static class Initializer implements
ApplicationContextInitializer<ConfigurableApplicationContext> {

    @Override
    public void initialize(ConfigurableApplicationContext
configurableApplicationContext) {
    TestPropertyValues.of(
     "spring.datasource.url=" + pg.getJdbcUrl(),
     "spring.datasource.username=" + pg.getUsername(),
     "spring.datasource.password=" + pg.getPassword(),
     "spring.datasource.driverClassName=" + pg.getDriverClassName(),
     "spring.datasource.driver-class-name" + pg.getDriverClassName())
            .applyTo(configurableApplicationContext.getEnvironment());
    }
}
```

Joonis 24. Koodilõik PostgreSQL konteineri loomisest ja testi initsialiseerijast.

Testolukordade käivitamisel tehakse REST Assured teegi abil päring kontrollerisse (vt. joonis 25) ja sealne äriloogika käivitatakse nagu rakenduse tavapärase töö ajal. Seejärel veendutakse reeglimootori testi puhul, et e-kiri on saadetud, vigased andmed on andmebaasi tekkinud ja verifitseeritakse nende andmete sisu (vt. joonis 25).

```
given().post("rule_engine/start").then()
     .assertThat().statusCode(200);
verify(emailService).sendRuleEngineResult(anyList());

String savedResultsSql =
"SELECT * FROM RULE_ENGINE_RESULT r
            WHERE r.id_ext in (333, 444)";

List<RuleEngineResult> ruleEngineResults =
jdbcTemplate.query(savedResultsSql,
new BeanPropertyRowMapper<>(RuleEngineResult.class));

assertNotNull(ruleEngineResults);
assertEquals(2, ruleEngineResults.size());
```

Joonis 25. Koodilõik reeglimootori automaattestist.

Andmete lukustamist testitakse analoogselt: luuakse Testcontainers andmebaas, REST Assured abiga saadetakse päringuid avalduse lukustamise *endpoint'*i ja kontrollitakse avalduse lukustatust.

Lisaks luuakse testolukorrad ka automaatprotsesside juba olemasolevatesse testklassidesse, kus mängitakse läbi olukord, kus automaatprotsessi kaasatavad äriobjektid on lukus.

4.3.2. Manuaalne testimine

Kuna automaattestid katavad loodud funktsionaalsuse äärejuhud ja keerulisemad olukorrad, siis võib töö kontekstis manuaalset testimist käsitleda kui suitsutestimise meetodi variatsiooni.

Suitsutestimine on tarkvara testimise vorm, mis viiakse läbi pärast tarkvara loomist, et kontrollida, kas tarkvara kriitilised funktsioonid töötavad korrektselt. Suitsutestimise eesmärk ei ole teha ammendavaid teste. [15]

Manuaalset testimist teostatakse läbi kasutajaliidese ning selle eesmärk on välja selgitada, kas kõik põhifunktsionaalsused töötavad nii, nagu töö analüüsi osas nõutud.

Kõigepealt testitakse, kas andmekontrolli käivitamine läbi kasutajaliidese töötab ja kuidas rakendus käitub. Kasutajaliideses nupule vajutades saadetakse päring, mida on tagarakenduse logist võimalik jälgida. Logi kinnitab, et andmekontrolli protsess on tõepoolest käivitunud (vt. joonis 26).

```
2021-05-12 ...RuleEngineService: Data checking process started
2021-05-12 ...RuleEngineService: Checking rule - ID 1:
Netosumma puudub mineviku kinnitatud maksel
2021-05-12 ...RuleEngineService: Found 3 conflicting rows
2021-05-12 ...RuleEngineService: Checking rule - ID 2:
Makse tulumaksuvaba summa negatiivne
2021-05-12 ...RuleEngineService: Found 2 conflicting rows
2021-05-12 ...RuleEngineService: Data checking process completed
```

Joonis 26. Lõik logidest pärast kasutajaliidese nupuvajutust.

Pärast protsessi edukat läbimist värskendatakse kasutajaliidese lehte, mille käigus laetakse uued andmed tabelitesse ning kuvatakse eduka päringu teade (vt. joonis 27).

Andmekontroll õnnestus

Reiko-Rainer Reinup | ROLE_ADMIN | Avalanchelabs | V

Vaheta roll

Avaldused | Otsused | Maksed | Aruanded | EL tagasinõuded | Admin

Staaži baasandmed | Klassifikaatorid | Vormid | Kasutajaõigused | Päringud | Seaded

Andmekontroll

Käivita andmekontroll

Andmekontrolli tulemused

Leiti 6 tulemust, lehel 1 - 1, kuva lehel 10

Rea ID	Objekti tüüp	Leitud mustri nimi
38120	PAYMENT	Netosumma puudub mineviku kinnitatud maksel
3187860	PAYMENT	Netosumma puudub mineviku kinnitatud maksel
3332249	PAYMENT	Netosumma puudub mineviku kinnitatud maksel
3332251	PAYMENT	Netosumma puudub mineviku kinnitatud maksel
179501	PAYMENT	Makse tulumaksuvaba summa negatiivne
3864777	PAYMENT	Makse tulumaksuvaba summa negatiivne

Mustrid

Leiti 2 tulemust, lehel 1 - 1, kuva lehel 10

Nimi	Aktiivne	Tegevused
Netosumma puudub mineviku kinnitatud maksel	Jah	Vali tegevus
Makse tulumaksuvaba summa negatiivne	Jah	Vali tegevus

Joonis 27. Kuvatõmmis edukast andmekontrollist kasutajaliideses.

Mustri muutmine õnnestus

Reiko-Rainer Reinup | ROLE_ADMIN | A

Vaheta roll

d | Avaldused | Otsused | Maksed | Aruanded | EL tagasinõuded | Admin

Staaži baasandmed | Klassifikaatorid | Vormid | Kasutajaõigused | Päringud | Seaded

Andmekontroll

Käivita ar

Andmekontrolli tulemused

Leiti 6 tulemust, lehel 1 - 1, ku

Rea ID	Objekti tüüp	Leitud mustri nimi
38120	PAYMENT	MUUDETUD
3187860	PAYMENT	MUUDETUD
3332249	PAYMENT	MUUDETUD
3332251	PAYMENT	MUUDETUD
179501	PAYMENT	Makse tulumaksuvaba summa negatiivne
3864777	PAYMENT	Makse tulumaksuvaba summa negatiivne

Mustrid

Leiti 2 tulemust, lehel 1 - 1, ku

Nimi	Aktiivne	Tegevused
MUUDETUD	Ei	Vali tegevus
Makse tulumaksuvaba summa negatiivne	Jah	Vali tegevus

Joonis 28. Kuvatõmmis mustri muutmisest kasutajaliideses.

Järgmisena testitakse olemasoleva mustri muutmist ja muudatuse kajastust äriprotsessis. Muudetakse puuduliku netosumma mustrit. Kasutajaliideses valitakse mustrite tabelis viimases tulbas tegevuste alt muutmise tegevus, mille peale avaneb kasutajaliideses mustrimuutmise kuva. Kuval muudetakse mustri nime ja aktiivsusstaatust (vt. joonis 28).

Olemasolevate andmekontrolli tulemuste andmebaasitabel tühjendatakse ja andmekontrolli protsess käivitatakse muudetud mustriga uuesti. Eeldatav tulemus siinkohal on ainult kahe tulemuse leidmine ja logi peaks kajastama muudatusi andmemustrites. Andmekontrolli käivitamise järel on kasutajaliidesest ja logist näha, et tulemused vastasid ootustele (vt. joonis 29 ja 30).

Joonis 29. Kuvatõmmis kasutajaliideses pärast andmekontrolli inaktiivse mustriga.

```
2021-05-13 ...RuleEngineService: Data checking process started
2021-05-13 ...RuleEngineService: Ignoring inactive rule:
ID(1) MUUDETUD
2021-05-13 ...RuleEngineService: Checking rule - ID 2:
Makse tulumaksuvaba summa negatiivne
2021-05-13 ...RuleEngineService: Found 2 conflicting rows
2021-05-13 ...RuleEngineService: Data checking process completed
```

Joonis 30. Lõik logidest inaktiivse mustriga.

Manuaalse testimise käigus selgitatakse välja ka e-kirja kuju korrektsus ja saatmise funktsionaalsus. Selleks kirjutati ajutiselt koodis üle e-kirja saaja ja saatja e-posti aadressid, et testimise käigus läheksid kõik e-kirjad testijale ning rakendus taaskäivitati. Olemasolevate andmekontrolli tulemuste andmebaas tühjendati uuesti, sest äriloogika kohaselt ei saadeta e-kirja välja juba leitud andmete kohta. Lisaks aktiveeriti eelnevalt deaktiveeritud reegel, et e-kirja vormistus paremini välja tuleks.

Andmekontrolli protsess käivitati uuesti ning peatselt pärast protsessi õnnestumist saabus e-kiri, kus oli kogu asjakohane info välja toodud (vt. joonis 31). Täpsustuseks, e-kirja pealkirjas sulgudes toodud tekst peegeldab süsteemis saatmise hetkel aktiivset kasutaja profiili. Näiteks, toodangukeskkonnas oleks sulgudes kirjas *live* ja testkeskkonnas *test*.

Reiko-Rainer Reinup
Andmekontrolli tulemused 2021-05-13 (reiko)
To: Reiko-Rainer Reinup

Andmekontrollis kasutatud mustrid:
- Netosumma puudub mineviku kinnitatud maksel
Vigaseid andmeridu 3
- Makse tulumaksuvaba summa negatiivne
Vigaseid andmeridu 2

Täpsem ülevaade: [redacted]/admin/request/ruleEngine/rule_engine/list

Joonis 31. Kuvatõmmis saadetud e-kirjast andmekontrolli tulemustega.

Automaatprotsesside andmelukkude manuaalseks testimiseks tuleb käivitada andmeid lukustav automaatprotsess ja seejärel navigeerida mõnele avaldusele, mis käivitatud protsessi raames lukustati.

Digiallkirjastamise protsess on võrdlemisi kiire ja võib töö lõpetada enne, kui oleks võimalik kasutajaliideses huvipakkuva avalduse detailvaatele navigeerida, mistõttu ei sobi see testimiseks. Kuna maksude arvutamise protsess kestab paar tundi, siis on see andmelukkude testimiseks sobilik.

Testimiseks käivitatakse maksude arvutamise protsess käsitsi. Rakenduse logist on näha avalduste identifikaatorid, mille protsess lukustas (vt. joonis 32). Lukustamises veendumiseks valitakse pisteliselt mõned avaldused, navigeeritakse kasutajaliideses nende kuvale ning jälgitakse tulemust. Avalduse detailkuval oli kõikidel kontrollitud avaldustel teade avalduse lukustatuse kohta (vt. joonis 33) ning kõik toimingud avaldusega keelatud.

Maksude arvutamise protsessi lõppedes on rakenduse logist võimalik välja lugeda, et protsess võttis avaldustelt lukud maha. Kasutajaliideses avalduse detailkuvale navigeerides ei ole enam teadet avalduse lukustatuse kohta ja nupud on taas aktiivsed.

```
2021-05-13 ...TaxCalculationService: Locking applications [485988,
490218, 483355, 484452, 482406, 484238, 488632, 490220, 488966...]
...
2021-05-13 ...TaxCalculationService: Unlocking applications [485988,
490218, 483355, 484452, 482406, 484238, 488632, 490220, 488966...]
2021-05-13 ...TaxCalculationService: Calculating taxes process is done
```

Joonis 32. Lõik rakenduse logist maksude arvutamise protsessi ajal.

-19024164

Avaldus on lukus

Isikukood

Isiku nimi TEST_Jelena TEST_Luus

Avalduse esitamise kuupäev 03.09.2019

Avalduse saabumise kuupäev 03.09.2019

Joonis 33. Kuvatõmmis automaatprotsessi poolt lukustatud avaldusest.

5. Hinnang

Antud peatükis tuuakse välja tehtud töö tulemused ja mõju tellija süsteemile. Lisaks pakutakse välja ideid, kuidas loodud süsteemi on võimalik edasi arendada.

5.1. Tulemused

Töö raames realiseeriti tellija projekti arendused, mis aitavad leida andmevigu kiiremini ja vähendavad ristuvatest protsessidest tekkinud andmevigu. Arendused testiti nii automaattestidega kui ka käsitsi läbi kasutajaliidese.

Tellija süsteemi arenduskoormuse tõttu ei ole realiseeritud arendused tänaseks tellija toodangukeskkonda jõudnud. Seetõttu ei ole võimalik hetkel veel arvuliselt välja tuua, kui palju loodud lahendused vigaseid andmeridu leidsid või protsesside ristumisi ära hoidsid.

Küll aga testiti arendusi kohalikus testkeskkonnas. Testimise tulemusena võib pidada tehtud tööd õnnestunuks.

Realiseeritud reeglimootori abiga on võimalik vigaseid andmeid regulaarselt leida ning vigaste andmete leiust teavitatakse süsteemi administraatorit.

Automaatprotsessid lukustavad oma toimingute ajaks andmed nii, et teised automaatprotsessid lukustatud äriobjekte ei kasuta. Lisaks on luku ajal ka süsteemi kasutajal keelatud lukustatud äriobjektiga tööülesandeid läbi viia.

Analüüsitud lahendused ja nende põhjal realiseeritud arendused on funktsionaalsed ja täidavad oma eesmärki.

5.2. Edasiarendused

Kuigi tellija süsteemi loodud arendused suurendavad andmete terviklust ja täidavad oma eesmärki, siis alati on võimalik leida viise, kuidas süsteemi veelgi täiustada.

Tänaseks loodud reeglimootoriga on võimalik süsteemi administraatoril jälgida andmemustrite poolt vigaste andmete tekkimist ja käivitada andmete kontrolli protsess, kuid administraatoril ei ole võimalik ise andmemustreid kasutajaliideses luua. Võimalik süsteemi edasiarendus on võimaldada läbi administraatoripaneeli süsteemi administraatoril andmemudeli reeglite loomine ja lisamine.

Loodud arendustega on praegu võimalik kirjeldada andmemustreid ja arendatud reeglimootor tõlgib mustrid ümber andmebaasipäringuteks. Kuigi see funktsionaalsus on tellija süsteemi vaatest väga kasulik, siis võib selle puudujäägiks pidada andmemustrite vähest keerukust. Võimalik süsteemi edasiarendus on reeglimootori täiendamine nii, et reeglimootor suudaks andmebaasipäringuteks tõlkida andmemustreid, mis kirjeldavad andmete kuju üle mitmete andmebaasi skeemide ja tabelite.

Tellija süsteem võimaldab tänaseks avalduse kui äriobjekti lukustamist. Küll aga ei saa seda üksinda pidada süsteemi täiustavaks, kui seda võimalust ei kasutata. Töös toodi välja mõned kohad, kus avalduse lukustamise loogika toob tellijale kasu, kuid tellija süsteemi mahtu silmas pidades on neid kohti ilmselt veelgi. Nende kohtade leidmine nõuab rohket analüüsi, mis jäi käesoleva töö mahust välja. Tänu olemasolevale lukkude funktsionaalsusele on sellele analüüsile vastavate arenduste teostamine süsteemis märksa lihtsam.

Edasiarendusena võib kaaluda ka reeglimootori poolt leitud vigaste andmetega seotud avalduste lukustamist. Kuna vigased andmed vajavad süsteemi administraatori tähelepanu, et selgitada välja vigade kriitilisus, siis on mõistlik vältida süsteemis nende vigaste andmete kasutamist. Funktsionaalsuse realiseerimist raskendab asjaolu, et kõik andmed ei pruugi olla nii otseselt avalduste või mistahes teiste äriobjektidega seotud, kui seda on käesoleva töö raames fookuses olnud makseread. Sellise funktsionaalsuse arendamine nõuab mahukat analüüsi ja võib vajada ka olemasoleva reeglimootori mingil määral ümber kirjutamist, kuid võib olla süsteemi andmekvaliteedi vaatest väga kasulik.

6. Kokkuvõte

Bakalaureusetöö eesmärgiks oli täiustada tellija projektis andmevigade leidmist ja vältimist ning tuvastada andmevead enne, kui nad jõuavad tekitada tellijale ajalist ja rahalist kahju.

Selle jaoks analüüsiti mitmeid võimalikke lahendusi, pidades silmas tellija soove, süsteemi olemasolevat arhitektuuri ja äriprotsesse. Analüüsi tulemusena jõuti järeldusele ja valiti välja, millist lahendust on tellija süsteemi kontekstis kõige sobilikum realiseerida. Lahendusteks osutusid algeline reegelimootor, mis võimaldab teostada andmebaasi tasandil andmete kuju kontrolli andmemustrite abil, ja tellija süsteemi kõrgetasemelise äriobjekti lukustamine nii, et seda ei saaks muuta.

Töös kirjeldati väljavalitud lahenduse realiseerimise protsessi. Realiseerimisel võeti arvesse tarkvaraarenduse häid tavasid ja süsteemi jätkusuutlikkust. Lahenduse õigsust ja korrektsust verifitseeriti nii automaattestidega kui ka manuaalselt läbi kasutajaliidese.

Tänu töö tulemusena valminud arendustele on tellija süsteem andmevigade tekkele vastupidavam ja võimaldab andmevigu kiiremini leida. Töös toodi välja ka ideed edasiarenduseks, mille abil oleks võimalik veelgi andmekvaliteeti ja -terviklust tõsta, kuid mis jäid käesoleva töö mahust välja.

Kasutatud kirjandus

[1] “Rule engine object - Documentazione IBM,” [Võrgumaterjal]. Saadaval: https://www.ibm.com/docs/it/odm/8.8.0?topic=engine-rule-object [Kasutatud 16. mai 2021]

[2] “Drools - Business Rules Management,” [Võrgumaterjal]. Saadaval: https://www.drools.org/ [Kasutatud 12. aprill 2021].

[3] “What Is OpenL Tablets?,” [Võrgumaterjal]. Saadaval: http://openl-tablets.org/what-is-openl-tablets [Kasutatud 12. aprill 2021].

[4] “Hibernate Developer Guide. Chapter 5. Locking,” [Võrgumaterjal]. Saadaval: https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html [Kasutatud 14. aprill 2021]

[5] “OptimisticLockException (Java(TM) EE 7 Specification),” [Võrgumaterjal]. Saadaval: https://docs.oracle.com/javaee/7/api/javax/persistence/OptimisticLockException.html [Kasutatud 14. aprill 2021]

[6] “Spring Roo - Reference Documentation. Chapter 10. Application Layering,” [Võrgumaterjal]. Saadaval: https://docs.spring.io/spring-roo/reference/html/base-layers.html [Kasutatud 20. aprill 2021]

[7] “Retrofit,” [Võrgumaterjal]. Saadaval: https://square.github.io/retrofit/ [Kasutatud 24. aprill 2021]

[8] “Autowired (Spring Framework 5.3.7 API),” [Võrgumaterjal]. Saadaval: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html [Kasutatud 11. mai 2021]

[9] “AngularJS: API: $resource,” [Võrgumaterjal]. Saadaval: https://docs.angularjs.org/api/ngResource/service/$resource [Kasutatud 8. mai 2021]

[10] “UI-Router for AngularJS (1.x) - UI-Router,” [Võrgumaterjal]. Saadaval: https://ui-router.github.io/ng1/ [Kasutatud 8. mai 2021]

[11] “AngularJS: Developer Guide: Scopes,” [Võrgumaterjal]. Saadaval: https://docs.angularjs.org/guide/scope [Kasutatud 8. mai 2021]

[12] “END-To-END Testing Tutorial: What is E2E Testing with Example,” [Võrgumaterjal]. Saadval: https://www.guru99.com/end-to-end-testing.html [Kasutatud 9. mai 2021]

[13] “REST Assured,” [Võrgumaterjal]. Saadaval: https://rest-assured.io/ [Kasutatud 10. mai 2021]

[14] “Testcontainers,” [Võrgumaterjal]. Saadaval: https://www.testcontainers.org/ [Kasutatud 28. aprill 2021]

[15] “Smoke Testing - SOFTWARE TESTING Fundamentals,” [Võrgumaterjal]. Saadaval: https://softwaretestingfundamentals.com/smoke-testing/ [Kasutatud 10. mai 2021]

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

Mina, Reiko-Rainer Reinup

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Andmetervikluse ja -kvaliteedi täiendamine andmemustrite abil", mille juhendajad on Argo Kivirüüt ja Jaanus Pöial

   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.

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 – Algskript andmebaasi eksempari initsialiseerimiseks

```
CREATE SEQUENCE rule_engine_condition_id_seq;
CREATE SEQUENCE rule_engine_info_id_seq;
CREATE SEQUENCE rule_engine_result_id_seq;

CREATE TABLE rule_engine_condition (
   id NUMBER(19) NOT NULL,
   target_schema VARCHAR2(4000) NOT NULL,
   target_table VARCHAR2(4000) NOT NULL,
   target_column VARCHAR2(4000) NOT NULL,
   condition_type_cl VARCHAR2(4000) NOT NULL,
   value VARCHAR2(4000),
   next_condition_id NUMBER(19),
   next_condition_operator VARCHAR2(4000) NOT NULL,
   created TIMESTAMP NOT NULL,
   modified TIMESTAMP NOT NULL,
   creator VARCHAR2(4000) NOT NULL,
   modifier VARCHAR2(4000) NOT NULL,
   CONSTRAINT rule_engine_condition_id_pk PRIMARY KEY (id),
   CONSTRAINT next_condition_id_fk FOREIGN KEY (next_condition_id)
      REFERENCES rule_engine_condition (id)
);

CREATE TABLE rule_engine_info (
      id NUMBER(19) NOT NULL,
      name VARCHAR2(4000) NOT NULL,
      first_condition_id NUMBER(19),
      created TIMESTAMP NOT NULL,
      modified TIMESTAMP NOT NULL,
      creator VARCHAR2(4000) NOT NULL,
      modifier VARCHAR2(4000) NOT NULL,
      CONSTRAINT rule_engine_info_id_pk PRIMARY KEY (id),
      CONSTRAINT first_condition_id_fk FOREIGN KEY
      (first_condition_id) REFERENCES rule_engine_condition (id)
);


CREATE TABLE rule_engine_result (
    id NUMBER(19) NOT NULL,
    id_ext NUMBER(19) NOT NULL,
    name VARCHAR2(4000) NOT NULL,
    rule_info_id NUMBER(19) NOT NULL,
    created TIMESTAMP NOT NULL,
    modified TIMESTAMP NOT NULL,
    creator VARCHAR2(4000) NOT NULL,
    modifier VARCHAR2(4000) NOT NULL,
```

```
    CONSTRAINT rule_engine_result_id_pk PRIMARY KEY (id),
    CONSTRAINT rule_eng_res_rule_id_fk FOREIGN KEY (rule_info_id)
            REFERENCES rule_engine_info (id)
);


CREATE TABLE PAYMENT
(
    ID NUMBER(19, 0),
    PERSON_DECISION_EXT NUMBER(19, 0),
    STATUS_CL VARCHAR2(30 CHAR),
    COMPENSATION NUMBER(16, 2),
    PAYMENT_DATE DATE,
    CREATED TIMESTAMP(6),
    MODIFIED TIMESTAMP(6),
    EXPORT_ID NUMBER(19, 0),
    COMPENSATION_TYPE_CL VARCHAR2(40 CHAR),
    INCOME_TAX NUMBER(16, 2),
    ACCOUNT VARCHAR2(4000 BYTE),
    PENSION NUMBER(16, 2),
    EXECUTION_DATE DATE,
    PERSON_CODE VARCHAR2(15 CHAR),
    NET_SUM NUMBER(16, 2),
    DECISION_SIGN_DATE TIMESTAMP(6),
    BANK_NAME VARCHAR2(4000 BYTE),
    SWIFT_CODE VARCHAR2(4000 BYTE),
    ACCOUNT_OWNER VARCHAR2(4000 BYTE),
    DECISION_NUMBER VARCHAR2(4000 BYTE),
    FIRST_NAME VARCHAR2(4000 BYTE),
    LAST_NAME VARCHAR2(4000 BYTE),
    REFERENCE_NUMBER VARCHAR2(4000 BYTE),
    APPLICATION_ID_EXT NUMBER(19, 0),
    CREATOR VARCHAR2(4000 BYTE),
    MODIFIER VARCHAR2(4000 BYTE),
    SOCIAL_TAX NUMBER(16, 2),
    DOC_NUMBER VARCHAR2(4000 BYTE),
    COUNTRY_CODE VARCHAR2(4000 BYTE),
    TAX_FREE_INCOME NUMBER(16, 2),
    LENGTH_YEAR NUMBER(3, 0),
    LENGTH_MONTH NUMBER(2, 0),
    LENGTH_DAY NUMBER(2, 0),
    AVERAGE_SALARY NUMBER(16, 2),
    COMPENSATION_MULTIPLIER NUMBER(3, 0),
    PREVIOUS_COMPENSATION NUMBER(16, 2),
    CURRENT_COMPENSATION NUMBER(16, 2),
    PERSON_EXT NUMBER(19, 0),
    PENSION_PERCENT NUMBER(3, 2),
    RESIDENT_CL VARCHAR2(4000 BYTE),
```

```
    FAILURE_REASON CLOB,
    TAX_CALCULATION_TIME TIMESTAMP(6),
    TAX_FREE_INCOME_SUM NUMBER(16, 2),
    IS_CORRECTION NUMBER(1, 0)  DEFAULT 0,
    INCOME_TAX_APPLICATION_ID NUMBER(19, 0),
    PERIOD_START_DATE DATE,
    PERIOD_END_DATE DATE,
    APPLICATION_NUMBER VARCHAR2(4000 BYTE),
    SETTLEMENT_AMOUNT NUMBER(16, 2),
    CUSTOMER_NUMBER NUMBER(19, 0),
    UNEMP_INS_TAX NUMBER(16, 2) DEFAULT 0.00
);

INSERT INTO RULE_ENGINE_CONDITION (id, target_schema, target_table,
target_column, condition_type_cl, value, next_condition_id,
next_condition_operator, created, modified, creator, modifier) VALUES

(rule_engine_condition_id_seq.nextval, 'PAYMENT', 'PAYMENT',
'NET_SUM', 'IS', 'NULL', null, null, current_timestamp,
current_timestamp, '39703192740', '39703192740');

INSERT INTO RULE_ENGINE_CONDITION (id, target_schema, target_table,
target_column, condition_type_cl, value, next_condition_id,
next_condition_operator, created, modified, creator, modifier) VALUES

(rule_engine_condition_id_seq.nextval, 'PAYMENT', 'PAYMENT',
'STATUS_CL', '=', '''PAYMENT_STATUS_APPROVED''', (select max(id) from
rule_engine_condition), 'AND', current_timestamp, current_timestamp,
'39703192740', '39703192740');

INSERT INTO RULE_ENGINE_CONDITION (id, target_schema, target_table,
target_column, condition_type_cl, value, next_condition_id,
next_condition_operator, created, modified, creator, modifier) VALUES

(rule_engine_condition_id_seq.nextval, 'PAYMENT', 'PAYMENT',
'PAYMENT_DATE', '<=', 'CURRENT_DATE', (select max(id) from
rule_engine_condition), 'AND', current_timestamp, current_timestamp,
'39703192740', '39703192740');

INSERT INTO RULE_ENGINE_INFO (id, name, first_condition_id, created,
modified, creator, modifier) VALUES (rule_engine_info_id_seq.nextval,
'Netosumma puudub mineviku kinnitatud maksel', (select max(id) from
rule_engine_condition), current_timestamp, current_timestamp,
'39703192740', '39703192740');

INSERT INTO PAYMENT (id, NET_SUM, STATUS_CL, PAYMENT_DATE) values
(333, null, 'PAYMENT_STATUS_APPROVED', '01.01.2021');

INSERT INTO PAYMENT (id, NET_SUM, STATUS_CL, PAYMENT_DATE) values
(444, null, 'PAYMENT_STATUS_APPROVED', '01.01.2021');
```