TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kaur-Kristofer Remmel 19233IADB

Nõudmiseni hoiuse intresside väljamakse süsteemi kaasajastamine

Bakalaureusetöö

Juhendaja: Toomas Lepikult
PhD
Kaasjuhendaja Erik Ehrbach
BSc

Tallinn 2025

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: Kaur-Kristofer Remmel

06.01.2025

Annotatsioon

Nõudmiseni hoius on pangandusteenuste üks keskseid komponente, mis võimaldab klientidel säilitada vabadust oma rahalisi vahendeid igal ajal kasutada. Arvestades, et tegemist on panga põhiteenusega, on nõudmiseni hoiuse sujuv ja usaldusväärne toimimine äärmiselt oluline. Kõrvalekalded või tõrked selle teenuse toimimisel võivad põhjustada märkimisväärseid tagajärgi, sealhulgas negatiivset mõju kliendirahulolule ja halvimal juhul kahjustada panga mainet. Seetõttu on hädavajalik, et nõudmiseni hoiusega seotud protsessid, sealhulgas intresside väljamaksed, oleksid paindlikud ja töötaksid tõrgeteta.

Olemasolev AS-i LHV nõudmiseni hoiuse intresside väljamakse protsess ei vasta enam täielikult panga kasvavatele vajadustele. Seetõttu on otsustatud kaasajastada kogu nõudmiseni hoiuse intresside väljamakse protsess.

Uue protsessi väljatöötamisel rakendati parimaid kaasaegseid tehnoloogilisi lahendusi, et saavutada kõik nõuetest tulenevad eesmärgid. Eelnevalt kasutatud mahukad andmebaasi päringud jaotati väiksemateks osadeks ning äriloogika jaotati kolme põhilisse etappi. Paindlikuse ja skaleeritavuse tagamiseks integreeriti protsessi sõnumiserver. Uus lahendus kaeti täielikult automaattestidega ning lisaks viidi läbi põhjalik manuaalne testimine. Toodangu keskkonnas mindi edukalt üle vanalt lahenduselt uuele, lahendades sealjuures ka varasemad andmebaasi tupikprobleemid (ingl k *deadlock*).

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 36 leheküljel, 6 peatükki, 26 joonist.

Modernization of the Interest Payment System for Demand Deposits

Demand deposit is one of the core components of banking services, enabling clients to maintain the freedom to access their funds at any time. Considering that it is a fundamental service of the bank, the seamless and reliable operation of demand deposits is of utmost importance. Deviations or failures in the functionality of this service can lead to significant consequences, including a negative impact on customer satisfaction and, in the worst case, damage to the bank's reputation. Therefore, it is essential that processes related to demand deposits, including interest payments, are flexible and operate without disruptions.

The existing AS LHV demand deposit interest payment process no longer fully meets the bank's growing needs. Consequently, it was decided to modernize the entire demand deposit interest payment process.

In developing the new process, the best modern technological solutions were applied to achieve all the objectives stemming from the requirements. The previously used large database queries were broken down into smaller components, and the business logic was divided into three main stages. To ensure flexibility and scalability, the process incorporated the ActiveMQ messaging platform. The new solution was fully covered with unit tests, and thorough manual testing was also conducted. The transition from the old solution to the new one in the production environment was successfully completed, resolving previous database deadlock issues.

The thesis is in Estonian and contains 36 pages of text, 6 chapters, 26 figures.

Lühendite ja mõistete sõnastik

Accounts	LHV panga IT-süsteem, mis tegeleb tehingute ja kontode haldamisega.
CoreCF	LHV panga aegunud monoliitsüsteem.
Customer	LHV panga IT-süsteem, mis tegeleb kliendi ja kliendi andmete haldamisega.
Kontsernikonto	Konto, mis annab ülevaate kontserni kuuluvate ettevõtete rahasaldodest ja võimaldab kontserni sisest laenamist.
Maksuresidentsus	Maksuresidentsus määrab ära, kuidas toimub kliendi tulu maksustamine.
Nõudmiseni hoius	Hoiustamisviis, kus raha saab hoiustada ja sellele ligi pääseda igal ajal ilma etteteatamiseta või mingeid lisatingimusi täitmata.
Nõudmiseni hoiuse intress	Intress, mida pangad maksavad klientidele nõudmiseni hoiuste eest. Hoiuse intressimäär võib olla fikseeritud või muutuv ning seda arvutatakse tavaliselt aastase protsendimäära alusel.
Safeguarding konto	Konto, mida kasutatakse klientide vahendite eraldamiseks ettevõtte tegevusfondidest, tagades nende kaitse maksejõuetuse korral.

Sisukord

Jooniste loetelu

1 Sissejuhatus

Nõudmiseni hoius on üks olulisemaid pangandusteenuseid, pakkudes klientidele paindlikkust oma raha igal hetkel kasutada [1]. Nõudmiseni hoiuse intressi arvutatakse kõikidele panga klientidele, kelle arvelduskontodel on piisav rahaline jääk. Kuna tegemist on ühe panga põhifunktsiooniga, on nõudmiseni hoiuse toimimine kriitilise tähtsusega. Probleemid selle teenuse haldamisel võivad viia kaugeleulatuvate tagajärgedeni, mis halvimal juhul võivad põhjustada pangale mainekahju. Seetõttu on äärmiselt oluline, et nõudmiseni hoiusega seotud funktsionaalsused, sealhulgas intresside väljamaksed, töötaksid tõrgeteta ning oleksid võimalikult paindlikud ja efektiivsed.

LHV Pangal on äriline motivatsioon konkurentsis püsimiseks arendada paindlik ja kaasaegne nõudmiseni hoiuse intressi väljamakse lahendus. Uus lahendus aitaks suurendada hoiuste mahtu ning pakkuda Eesti tihedal pangandusturul konkurentide ees parimat teenust.

Käesoleva lõputöö teemaks on nõudmiseni hoiuse intresside väljamakse süsteemi kaasajastamine. Eesmärk on luua uus nõudmiseni hoiuse intresside väljamakse süsteem, mis saab hakkama kasvavate andmemahtudega, võimaldab kiiret ja efektiivset äriloogika arendust ning ei häiri teiste panga baasfunktsionaalsuste tööd. Valmis protsess võetakse pangas kasutusele.

Lõputöö teises peatükis kirjeldatakse töö käigus lahendatavat probleemi. Vana väljamakse protsessi kaardistamine ning uue protsessi planeerimine toimub kolmandas peatükis. Töö neljandas peatükis kirjeldatakse detailselt, kuidas uus intressi väljamakse protsess töötab, mis tehnoloogiaid kasutab ning kuidas teises peatükis väljatoodud probleemi lahendab. Tulemuse analüüs koos esinenud probleemide ja statistikaga on esitatud viiendas peatükis.

Autori panus antud protsessi väljatöötamisel hõlmas nõuete kogumist, tehnilist analüüsi, prototüübi loomist ja testimist. Nõuete kogumise ja tehnilise analüüsiga toetas autorit

kaasjuhendaja. Prototüübi arendamisel osales AS-i LHV Pank tarkvaraarendaja, Mario Sepp, kes koostas kontode agregeerimise etapi.

2 Probleemi kirjeldus

Teema on aktuaalne, kuna eksponentsiaalselt kasvavad andmemahud pakuvad nii uusi võimalusi kui ka olulisi väljakutseid erinevatele IT-süsteemidele. Pangandussektor on olnud aeglane innovatsioonide omaksvõtmisel, mistõttu ei ole enamik aegunud süsteeme suutlikud toime tulema kasvava töökoormusega [2], [3].

LHV pank seisis olukorra ees, kus sooviti erinevate pakkumistega meelitada kliente enda juurde hoiuseid kasvatama, kuid pidevalt kasvavad andmemahud, olid märkamatult hakatanud tekitama probleeme panga infrastruktuuriga. Esmased probleemid ilmnesid erinevate andmebaasi päringute ajalõppudega (ingl k *timeout*) ning lähemal uurimisel selgus, et nende põhjustaja oli nõudmiseni hoiuse väljamakse protsess. See tähendas, et pank pidi edasi lükkama klientide meelitamist personaalsete pakkumistega, mis tegi otsest kahju panga finantseesmärkidele. Antud protsessi töötamise ajal, oli panga baasfunktsionaalsus aina rohkem häiritud. Lisaks tekkis oht, et mingil hetkel võib jääda klientidele intressitulu välja maksmata, mis tooks mainekahju LHV pangale.

Protsessi analüüsist ilmnes kaks peamist probleemi:

1. Protsessi töötamisaeg oli andmemahtude kasvamise tõttu veninud kaks korda pikemaks võrreldes kahe aasta taguse ajaga.
2. Protsessi töötamise ajal lukustati kõikide panga klientide saldod, kellele intressi välja maksti. Sellel ajal ei saanud kliendid kasutada teenuseid, mis vajasid saldot.

Kliendimõju protsessi töötamise ajal:

- kliendid ei saa teha makseid;
- kliendid ei saa vaadata konto väljavõtet;
- häiritud on ka teised teenused, mis vajavad saldot.

Lahenduse valimisel mängivad rolli erinevad AS-i LHV Pank süsteemidest lähtuvad nõuded. Uus intressi väljamakse protsess kasutab *Java Spring Boot* raamistikku, *ActiveMQ* sõnumiserverit ja *Microsoft SQL* serverit. Protsess kirjutatakse *Accounts*

rakendusse, milles asetsevad erinevad panga baasteenused. Nt tehingute tegemine, kontode avamine ja konto saldo vaatamine.

Probleemi lahendamine jagatakse kaheks osaks:

1. analüüs ja nõuete kogumine uue protsessi koostamiseks;
2. uue protsessi ehitamine.

Tehniline lahendus näeb endas täielikku loobumist kauakestvatest andmebaasi päringutest. Äriloogika kirjutatakse kolme loogilisse komponenti, mis täidavad kindalt äriloogilist eesmärki ning infovahetus nende äriloogika komponentide vahel toimub kasutades *ActiveMQ'd*. Eesmärk on tagada võimalikult suur paindlikkus kogu protsessis, mis omakorda vähendab kriitiliste vigade tekkimist ning vastupidavust andmemahtude kasvule

3 Uue väljamakse protsessi lahenduse analüüs ja metoodika

Käesolevas peatükis kirjeldatakse uue väljamakse protsessi nõudeid ja analüüsitakse olemasoleva protsessi toimimist.

3.1 Vana väljamakse protsessi kaardistamine

Vana nõudmiseni hoiuse intressi väljamakse protsess, oli koostatud kahe *ColdFusion* protseduuri abil, kus segunes *ColdFusion*-kood SQL-koodiga. Protseduurid käivitati *ColdFusion* sündmusega. Esimeses protseduuris lukustati andmebaasi tabelid ja valideeriti, et protseduuri poleks juba käivitatud ning intressid makstaks välja õige kuu kohta.

Teine protseduur koondas endasse kogu nõudmiseni hoiuse intressi väljamakse loogikat:

- intressi teeninud klientide leidmine;
- intresside agregeerimine;
- intressi summade väljamaksmine;
- tulumaksukannete tegemine;
- raamatupidamiskannete koostamine;
- auditeerimiskirjete loomine.

Kogu protsess oli kirjutatud ühte faili, kus puudus loogiline jaotus meetoditeks. Kood oli kirjutatud *SQL*-päringute ja *ColdFusion*-koodi abil, mis põhjustasid koodi raske loetavuse ja äriloogika segunemise andmehalduskihiga. Kõik andmebaasi päringud jooksid ühe andmebaasi tehingu (ingl k *transaction*) sees, mis andmebaasi mahtude kasvamise korral, hakkas probleeme valmistama. Tekkisid tupikud (ingl k *deadlock*), mis segasid teiste protsesside ligipääsu andmebaasi tabelitele. Lisaks tupikutele tekitas ühe andmebaasi tehingu kasutamine olukorra, kus ühe kliendi vigased andmed võisid põhjustada kogu protsessi peatumist ja kõigi muudatuste tagasipööramist (ingl k *rollback*).

Mõlemal protseduuril puudusid automaattestid ja integratsioonitestid. Iga muudatust tuli testida käsitsi, mis suurendas oluliselt koodi muudatustele kuluvat aega. Lisaks asuvad protseduurid aegunud *CoreCf* monoliitrakenduses, mille haldamine plaanitakse esimesel võimalusel ära lõpetada.

Vana nõudmiseni hoiuse protsessi raamatupidamiskannete koostamise loogika on välja toodud joonisel 1.

```
<cfloop index="ind" from="1" to="#get_accrued_totals.RecordCount#">
	<cfscript>
	get_currency = Evaluate("Request.A.Q_CURRENCIES.ID_" &
get_accrued_totals.currency_id[ind]);
	amount_interest =
Request.CF_RoundCcy(get_accrued_totals.amount_interest[ind],
get_accrued_totals.currency_id[ind]);
	// convert interest from old currency to EUR after euro date
	if (Request.CF_CheckEURTrans(get_currency.shortname)) {
		temp_eur_trans_rate = Request.A["EUR_TRANS_" &
get_currency.shortname & "_RATE"];
		get_currency = Request.A.Q_CURRENCIES.ID2_EUR;
		amount_interest =
Request.CF_RoundCcy(get_accrued_totals.amount_interest[ind]/temp_eur_trans
_rate, get_currency.currency_id);
	}
	amount_interest = (-1)*amount_interest;
	erp_account_pair =
Request.CF_ERPAccountPair(get_accrued_totals.erp_account_no[ind],
amount_interest);
	</cfscript>

	<cfif amount_interest IS NOT 0>
		<cfif Attributes.submit IS 2>
			<cfmodule template="/functions/proc_al_erp_insert.cfm"
				currency_id="#get_currency.currency_id#"
				amount="#amount_interest#"
				date_inv="#Attributes.date_pay#"
				erp_account_pair="#erp_account_pair#"
				bale_id="#get_accrued_totals.bale_id[ind]#">
		</cfif>
	</cfif>
</cfloop>
```

Joonis 1. Vana nõudmiseni hoiuse protsessi raamatupidamiskannete koostamine.

3.2 Uue väljamakse protsessi nõuded

Antud alampeatükis defineeritakse uue protsessi funktsionaalsed ja mittefunktsionaalsed nõuded.

3.2.1 Funktsionaalsed nõuded

Uue protsessi funktsionaalsed nõuded on tuletatud, analüüsides AS LHV Panga olemasolevaid IT-süsteeme ja potentsiaalseid tuleviku murekohti.

Funktsionaalsed nõuded:

- protsessi suhtlus andmebaasiga peab toimuma *Microsoft SQL* andmebaasi päringukeeles;
- protsessi äriloogika peab olema kirjutatud *Java* programmeerimiskeele *Spring Boot* raamistikus;
- protsess peab olema loodud *Accounts* rakendusse;
- protsessi äriloogika ja andmehalduskiht peavad olema eraldi;
- protsess ei tohi tekitada tupikuid (ingl k *deadlock*) ja liigset andmebaasi koormust;
- protsess peab olema suuteline veaolukordade puhul sõnumeid uuesti menetleda;
- protsess peab olema täielikult jälgitav ja monitooritav.

3.2.2 Mittefunktsionaalsed nõuded

Uue protsessi mittefunktsionaalsed nõuded on koostatud tootejuhi ja arendajate omavahelise arutelu põhjal.

Mittefunktsionaalsed nõuded:

- protsess ei tohi segada panga teiste funktsionaalsuste tööd.
- protsess peab olema lihtsasti muudetav.
- protsess peab pidama vastu andmemahtude kasvule.
- Tootejuhtidel peab tekkima võimalus ise kontrollida, kas protsess töötas korrektselt.

3.3 Lahenduse valimine

Nõuete täitmiseks on kaks peamist võimalust:

1. Kaasajastada vana protsess.
2. Luua uus kontopõhine protsess.

Vana protsessi kaasajastamine kujutaks endast protsessi tõstmist *CoreCF* rakendusest *Accounts* rakendusse. Protsessi *ColdFusion* koodi asendamist *Java* koodiga ning suurte andmebaasi päringute väiksemaks lammutamist, koos indeksite optimeerimisega. Protsess töötleks kõiki intressi teeninud kontosid samal ajal, mis tagaks kiire tööaja, kuid selline lahendus ei ole väga paindlik ja kergesti hallatav. Andmebaasi päringute optimeerimine ei pruugi tagada, et protsess oleks jätkusuutlik kasvavate andmemahtudega. Samuti uue äriloogika loomine võib olla aeganõudev protsess.

Uus kontopõhine protsess kujutab endast paindlikku, aga mahukat lahendust. Kogu protsess on jagatud kolmeks peamiseks osaks: kontode leidmise etapp, agregeerimise etapp, väljamakse etapp. Kõik etapid on omakorda jagatud väikesteks eraldiseisvateks osadeks ning etappide vaheline info liiklus toimub läbi *ActiveMQ* järjekordade. Protsessi alguses leitakse tuhande kaupa kontod, mis on intressi teeninud, ja neid hakatakse üks haaval töötlema. Selline lähenemine tagab protsessile paindlikkuse ja skaleeritavuse. Samas kontopõhine lähenemine tõstab oluliselt kogu protsessi ajalist kestvust.

Lahenduse valimisel arvestati nõudmiseni hoiuse intressi väljamakse protsessi kriitilisust, arendusressursi olemasolu, potentsiaalset halduskulu ning protsessi ajalise kestvuse nõuete puudumist. Otsustati luua uus kontopõhine lahendus, sest see vähendab protsessi haldamisele ja muutmisele kuluvat aega, mis hüvitab pikas perspektiivis uue protsessi arendusele kulunud aja. Uus lahendus tagab parema ülevaate kogu protsessist ning võimaldab paremat veahaldust kui vana protsessi kaasajastamine.

3.4 Uue kontopõhise protsessi analüüs

Peamine probleem, mida uue lahendusega prooviti vältida, oli andmebaasi ülekoormamine ja tupikute teke (ingl k *deadlock*). Selle tõttu võeti vastu otsus, keskenduda ühe konto töötlemisele korraga. See võimaldab täpselt maha logida ühe konto töötlemise, andes parema auditeerimise võimaluse – vea tekkimisel tekib probleem ainult ühe kontoga, kõik teised intressi teeninud kontod ei jää selle pärast intressituluta.

Skaleeritavuse tagamiseks rakendati erinevate etappide vahel *ActiveMQ* järjekorda. See võimaldab etappidel töötada täiesti iseseisvalt. Kui üks etapp lisab sõnumeid järjekorda kiiremini, kui teine etapp neid töödelda suudab, siis sellest ei teki probleeme, sest kõik sõnumid töödeldakse vastavalt järjekorrale ära. Lisaks on võimalik kindlale järjekorrale lisada juurde paralleelsust (ingl k *concurrency*), mis tähendab, et järjekorral on rohkem sõnumitarbijaid. See võimaldab eraldada kindlale etapile rohkem ressurssi, et ajalist kestvust vähendada. Järjekorrad tagavad parema veahalduse. Kui järjekorrast töötlusesse võetud sõnumi töötlemisel tekib viga, siis kukub sõnum vigaste sõnumite järjekorda (ingl k *dead letter queue*). Sealt võtab automaatne protsess sõnumi välja ja proovib seda uuesti töödelda. Kui automaatset töötlemist ei toimu, peab pangatöötaja sekkuma, andmed ära parandama ning manuaalselt sõnumi töötlusesse tagasi tõstma. Sõnumit hakatakse uuesti töötlema sealt järjekorrast, kust ta töötlusesse võeti. Näiteks kui viga tekib väljamakse etapis, siis teisi etappe enam uuesti ei käivitata.

Protsessi peab olema võimalik manuaalselt käivitada kindla konto, kliendi või kõikide kontode kohta. See tagab arendustiimile võimaluse kergemini parandada valesti välja makstud intresse, ilma et peaks suures mahus andmebaasi päringuid koostama.

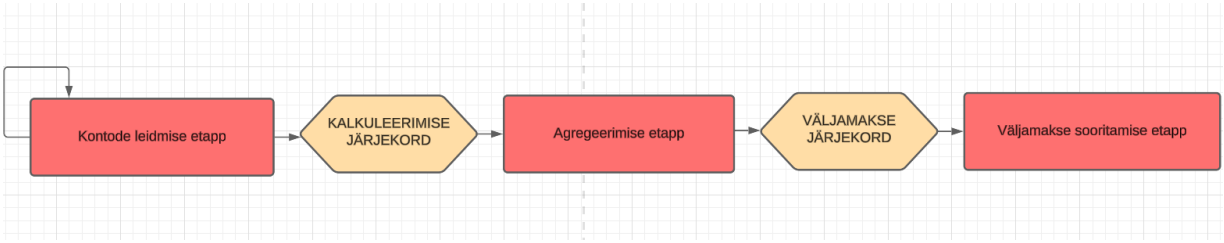
Protsessi üle parema kontrolli saavutamiseks ning arenduse kiirendamiseks, peaks iga etapp algama funktsiooni lipuga (ingl k *feature flag*), mis võimaldab kasutajaliidesest hõlpsasti etappi sisse või välja lülitada. Selline lähenemine võimaldab protsessi tükk haaval valmis teha ja koheselt toodangu keskkonda ülesse laadida, ilma et protsess reaalselt tööle hakkaks. See kiirendab arendust ning annab parema kontrolli protsessi üle. Protsessi jooksutamisel toodangu keskkonnas, annab funktsiooni lipp väga hea võimekuse viga tekitav element välja lülitada.

Uue lahenduse kirjutamisel tuleb rakendada *Clean Code* printsiipe ja *Domain Driven Design'i*. Selle eesmärk on tagada võimalikult kiire ja efektiivne uue funktsionaalsuse loomine. Lisaks aitab ühtse ja kvaliteetse koodistiili hoidmine ära hoida segadust ja ettearvamatuid probleeme [4], [5].

Lisaks peab koodis kasutama kasutusjuhtumi (ingl k *use case*) arendusmustrit koos käsu (ingl k *command*) mustriga, et tagada ühtlane disain üle terve *Accounts* rakenduse ja äriloogika eraldatus andmehalduskihist. Kasutusjuhtumi muster aitab paremini hallata äriloogikat ning käsu muster tagab koodi kergema muutmise [6], [7].

3.4.1 Uue kontopõhise protsessi skeem

Nõuete põhjal koostati uus protsessi skeem (Joonis 2). Otsustati jätkata kahe olemasoleva protsessi andmebaasitabeli kasutamist. Kasutusse jäid tabel *account_interest_tax*, mis säilitab tasutud tulumaksu infot, ja tabel *portfolio_interest_cache*, mis hoiab arvutatud intressitulu andmeid. Nimetatud tabelid viiakse üldkasutatavast *dbo* skeemist üle *Accounts* rakenduse skeemi alla. Protsessi paremaks jälgitavuseks luuakse tabel *account_interest_entry* ning selleks, et vältida ühe konto mitmekordset töötlemist, luuakse tabel *account_interest_cache_lookup*. Äriloogika otsustati jagada kolmeks erinevaks etapiks: kontode leidmise etapp, agregeerimise etapp ja väljamakse etapp. Iga etapp koosneb omakorda väiksematest äriloogilistest komponentidest. Erinevaid etappe ühendavad kalkuleerimise ja väljamakse järjekorrad. Protsessi detailne skeem on kujutatud Lisa 2.



Joonis 2. Lihtsustatud intressi väljamakse skeem.

4 Uue väljamakse protsessi prototüüpimine

Käesolevas peatükis tutvustatakse, kuidas toimub intressitulu leidmine ja selle väljamaksmine klientidele.

4.1 Kasutatavad tehnoloogiad

Intresside väljamakse protsess on loodud Java programmeerimiskeeles, kuna *Accounts* rakendus, kuhu vastav protsess integreeriti, on kirjutatud Java koodis. Java on objektorienteeritud programmeerimiskeel, mis on 2024. aastal populaarsuselt kuues programmeerimiskeel ning *TIOBE* indeksi kohaselt, mis mõõdab programmeerimiskeelte populaarsust otsingumootorite tulemuste ja turu nõudluse alusel, on *Java* kolmas kõige populaarsem programmeerimiskeel [8], [9].

Erinevate äriloogiliste komponentide omavaheliseks infovahetuseks kasutati *ActiveMQ* sõnumiserverit (ingl k *message broker*), mis pakkus paindlikku lahendust suure mahuga andmeedastuse haldamiseks [10]. Andmebaasina kasutati töökindlat ja laialt levinud *Microsoft SQL Serverit*, mis võimaldas arendajatel kasutada täiendavaid tööriistu uue süsteemi jõudluse analüüsimiseks. Protsessi loomisel rakendati *Spring Boot* raamistikku, et lihtsustada intresside väljamakse süsteemi loomist, integreerides andmebaasi, sõnumiserveri ja äriloogika komponendid ühtseks tervikuks [11].

4.2 Ettevalmistavad tööd

Analüüsi käigus tehti otsus viia osa vana intressi väljamakse süsteemiga seotud andmebaasi tabelid üle üldkasutatavalt skeemilt *Accounts* rakenduse skeemi alla. Selline lähenemine aitab paremini organiseerida andmebaasi struktuuri, tagades, et kõik *Accounts* rakenduse alla kuuluvad tabelid paikneksid samas skeemis. Üleviimisel kasutati *MSSQL* sünonüümi (Joonis 3), mis võimaldab andmebaasi tabeli ümber nimetada, ilma et oleks vaja koodis vana tabeli nime muuta [12]. Samuti lisati vastavad

tabelid *Accounts* rakenduse *Liquibase'i* konfiguratsiooni, mis võimaldab tõhusamalt hallata nende tabelite struktuurimuudatusi.

Accounts rakenduse skeemi alla viidud tabelid on järgmised:

- ***ACCOUNT_INTEREST_TAX***: See tabel hoiustab nõudmiseni hoiuse tulumaksukannete infot, sisaldades viidet tulumaksu kandele, kontole, millelt tulumaks maha arvestati, ning makstud summa suurusele. Skeemi muutmise käigus lisati viiteväljale indeks, mis kiirendab vastavas tabelis andmebaasi päringuid viitevälja kaudu.
- ***PORTFOLIO_INTEREST_CACHE***: Selles tabelis talletatakse andmed intressiarvutuste kohta. Intress arvutatakse iga päev konto ja valuuta põhiselt ning vastavad arvutustulemused lisatakse tabelisse. Tabelis hoitakse teavet intressi summa, konto saldo, intressiprotsendi, valuuta ja konto numbri kohta. Skeemi muutmise käigus nimetati tabel ümber ning see sai uueks nimeks *ACCOUNT_INTEREST_CACHE*.

```
ALTER SCHEMA SC_ACCOUNT TRANSFER dbo.account_interest_tax;
CREATE SYNONYM dbo.account_interest_tax FOR
SC_ACCOUNT.account_interest_tax;
IF NOT EXISTS(SELECT * FROM sys.indexes WHERE name = 'IDX_ACIT_reference'
AND object_id = OBJECT_ID('SC_ACCOUNT.account_interest_tax'))
CREATE NONCLUSTERED INDEX IDX_ACIT_reference ON
SC_ACCOUNT.account_interest_tax (reference)

ALTER SCHEMA SC_ACCOUNT TRANSFER dbo.portfolio_interest_cache;
EXEC sp_rename 'SC_ACCOUNT.portfolio_interest_cache',
'account_interest_cache';
CREATE SYNONYM dbo.portfolio_interest_cache FOR
SC_ACCOUNT.account_interest_cache;
```

Joonis 3. Vanadele andmebaasi tabelitele sünonüümi loomine.

Lisaks vanast protsessist üle toodud tabelitele oli vajalik luua kaks uut tabelit:

- ***ACCOUNT_INTEREST_ENTRY***: Selle tabeli eesmärk on anda põhjalik ülevaade kõigist välja makstud intressidest. Tabelis sisalduvad andmed hõlmavad intressi teeninud kontot, intressi summat, valuutat, intressiprotsenti, makse viidet, makse staatust, makseperioodi, kontot, millele intress maksti, raamatupidamiskonto lühinumbrit ning tulumaksu protsenti. Tabelit kasutatakse nii intressi väljamakse teabe kogumiseks kui ka väljamaksete auditeerimiseks.
- ***ACCOUNT_INTEREST_CACHE_LOOKUP***: See tabel tagab, et kõik intressi teeninud kontod saadetakse agregeerimise etappi täpselt ühe korra. Tabeli abil on võimalik leida andmebaasist intressi teeninud kontosid ilma duplikaatideta, võimaldades tõhusamat andmete töötlemist.

4.3 Konto leidmise etapp

Arendati intressi väljamakse protsessi käivitamise objekt (*ExecuteInterestPayoutJob)*, mis aktiveeritakse *Cron* ajastustööriista poolt. Programm käivitub iga kuu 5. kuupäevast kuni 12. kuupäevani, alustades tööd öösel kell 3.30 ning taaskäivitudes iga tunni tagant kuni kell 23.30 (Joonis 4). Selline taaskäivitamise strateegia tagab, et programm käivitatakse automaatselt uuesti, kui tekib viga või programm mingil põhjusel ei käivitu. Lisaks tagab selline lähenemine, et arendajad saavad reageerida, probleemid likvideerida ning ei pea protsessi uuesti käivitamiseks IT-halduselt abi paluma.

```
@Override
@SystemProcessContext
@QuartzScheduled(cronProperty =
"${lhv.services.cron.interest.demand-deposit-interest}",
 auditUser = SystemConstants.SYSTEM_USER_ID)
public void execute() {
       log().info("Scheduled demand deposit interest job has started.");

       var command = PrepareMonthlyInterestCalculationCommand.builder()
              .periodCode(YearMonth.now().minusMonths(1))
              .build();
       prepareMonthlyInterestCalculationUseCase.execute(command);

       log().info("Scheduled demand deposit interest job has ended.");
}
```

Joonis 4. Ajastatud intressi väljamakse protsessi käivitamine.

Peale intressi väljamakse protsessi aktiveerimist, käivitatakse alamprogramm (*PrepareMonthlyInterestCalculationUseCase*) (Joonis 5), mis alustab eelneval kuul intressi teeninud kontode töötlemist *ACCOUNT_INTEREST_CACHE* tabelist. *ACCOUNT_INTEREST_CACHE* tabelist võetakse tuhande kaupa intressi teeninud kontosid ja lisatakse need *PENDING* staatusega *ACCOUNT_INTEREST_CACHE_LOOKUP* tabelisse. Tuhande kaupa kontode töötlemine, väldib andmebaasi lukkude teket ning annab võimaluse teistel protsessidel andmebaasi tabelile edukalt ligi pääseda. *LOOKUP* tabel aitab hoida ülevaadet juba töötlusesse võetud kontodest ning tagab, et ühelegi kontole ei makstaks intressi rohkem kui on lubatud. Peale *LOOKUP* tabelisse lisamist, luuakse kontodest sisendobjekt (*SendAccountsToInterestCalculationCommand*) ja alustatakse ettevalmistust kontode saatmiseks agregeerimise etappi.

```
@Component
@RequiredArgsConstructor
public class PrepareMonthlyInterestCalculationUseCase {

    private final PrepareMonthlyInterestCalculationUseCaseSteps steps;

    public void execute(PrepareMonthlyInterestCalculationCommand command)
    {
        List<AccountInterestCalculationCandidate> candidates;
        do {
            candidates = steps
                    .createInterestCalculationCandidates(command);
            steps.sendAccountsToInterestCalculation(candidates);
        } while (!candidates.isEmpty());
    }
}
```

Joonis 5. Intressi teeninud kontode leidmine.

Sisendobjekti sees olevatest konto ID-dest ja perioodi infost koostatakse sõnumid ja pannakse ükshaaval *CALCULATION* järjekorda (Joonis 6). Kui konto sõnum on edukalt järjekorda lisatud, muudetakse vastava kirje staatus *SUCCESS* staatuseks *ACCOUNT_INTEREST_CACHE_LOOKUP* tabelis. Staatuse muutmine annab parema ülevaate, millised kontod on töötlusesse võetud ja millised on edukalt saadetud agregeerimise etappi. Meetod, mis saadab konto sõnumeid järgmisesse etappi, on

varustatud *@Transactional* annotatsiooniga. See aitab tagab, et kõik sisendobjekti kirjed saavad staatuseks *SUCCESS* ja ükski kirje ei lähe kaduma. Vea tekkimisel, pööratakse kõik meetodis tehtud muudatused tagasi. Kirjed, millel jäi külge *PENDING* staatus, võetakse hiljem uuesti töötlusesse.

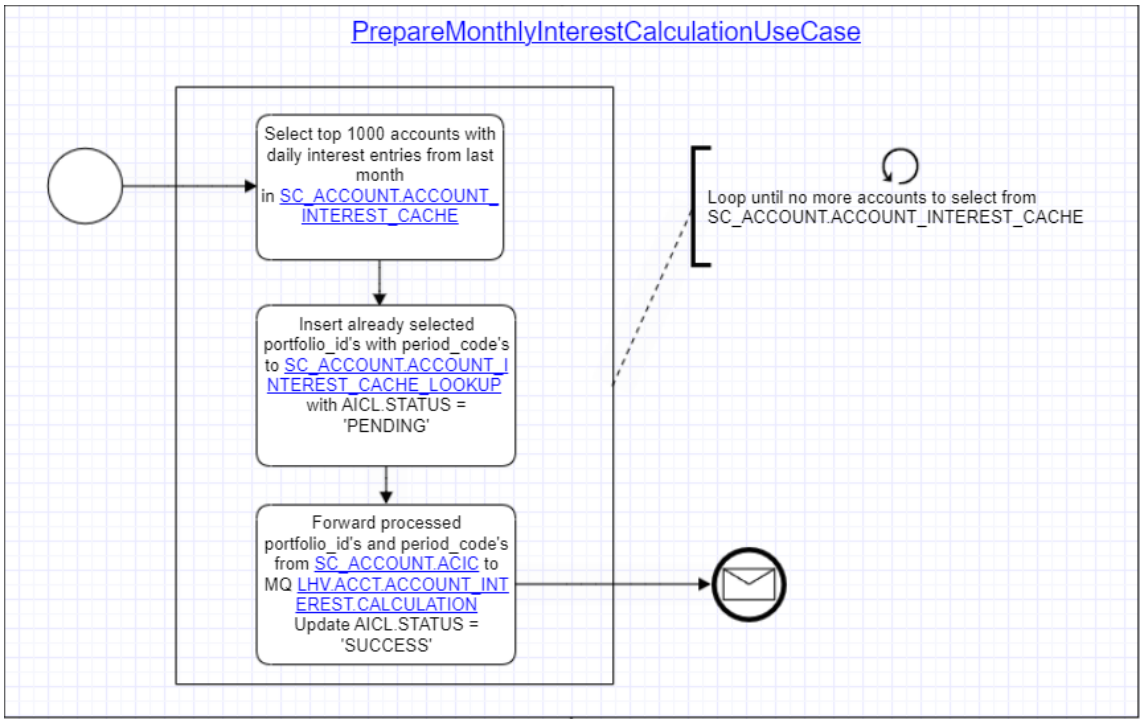
```
public void sendAccountsToInterestCalculation
       (List<AccountInterestCalculationCandidate> candidates) {
       candidates.forEach(this::sendAccountToInterestCalculation);
}

private void sendAccountToInterestCalculation
       (AccountInterestCalculationCandidate candidate) {
       var interestCalculationInfo = AccountInterestCalculationInfo
              .builder()
              .accountId(candidate.getAccountId())
              .periodCode(candidate.getPeriodCode())
              .build();
       accountInterestCalculationSender
              .sendAccountToInterestCalculation(interestCalculationInfo);
       updateCandidateStatus(candidate);
}

private void updateCandidateStatus
       (AccountInterestCalculationCandidate candidate){
       candidate
       .setStatus(AccountInterestCalculationCandidateStatus.SUCCESS);
       accountInterestCalculationCandidateRepository.save(candidate);
}
```

Joonis 6. Konto ja perioodi info saatmine järjekorda.

Pärast kõigi sisendobjekti kontode töötlemist luuakse uus sisendobjekt tuhande kontoga ja nii edasi, kuni kõik eelmisel kuul intressi teeninud kontod on saadetud agregeerimise etappi.

Intressi väljamakse protsessi käivitamise objekt (*ExecuteInterestPayoutJob*) on manuaalselt käivitatav koos parameetritega või ilma. Kui programm käivitatakse ilma parameetriteta, makstakse intress välja kõigile, kellele see on arvutatud. Programmile saab sisendiks anda parameetritena kindlad kontod või kliendid, kellele intress välja makstakse. Manuaalne käivitamine nõuab IT-halduse abi.

Joonis 7 esitab detailse ülevaate tervest kontode leidmise etapist.



Joonis 7. Kontode leidmise etapi tehniline skeem.

Etapi töökindluse suurendamiseks loodi töötlemata jäänud kirjete haldamise programm (*HandlePendingCalculationCandidatesJob*). Programmi eesmärk on lahendada olukord, kus kontod jäävad *ACCOUNT_INTEREST_CACHE_LOOKUP* tabelis *PENDING* staatusesse, kui agregeerimise etappi saatmisel tekivad probleemid. Programm käivitatakse *Cron* parameetriga iga kuu 5. kuupäevast kuni 12. kuupäevani, alustades varahommikul kell 5:00. See käivitub uuesti iga kolme tunni tagant ning võtab *ACCOUNT_INTEREST_CACHE_LOOKUP* tabelist *PENDING* staatusega konto kirjed ja loob nendest sisendobjekti (*SendAccountsToInterestCalculationCommand*). Objekt antakse sisendiks programmile, mis loob kontodest sõnumid ja saadab need agregeerimise etappi (Joonis 8).

```
public void execute(HandlePendingCalculationCandidatesCommand command){
	var entries = accountInterestCalculationCandidateRepository
				.findCalculationCandidatesWithPendingStatus(
				command.getPeriodCode(),
				command.getStatus());

	if (entries.isEmpty()){
		log.info("No calculation candidates found with PENDING
status.");
		return;
	}

	var sendToCalculationCommand =
SendAccountsToInterestCalculationCommand.builder()
		.candidates(entries)
		.build();
	sendAccountsToInterestCalculationUseCase.execute(sendToCalculation
Command);
}
```

Joonis 8. *PENDING* staatuses kirjete uuesti töötlemine.

4.4 Agregeerimisetapp

Agregeerimisetapp algab automaatselt, kui esimene sõnum on jõudnud *CALCULATION* järjekorda. Sõnum võetakse järjekorrast välja ning alustatakse selle töötlemist. Sõnumist leitakse konto ja periood, mis saadetakse sisendiks alamprogrammile (*CalculateAccountInterestUseCase*), mille eesmärk on kontole arvutatud intressid kokku agregeerida (Joonis 9).

```
@SneakyThrows
@JmsListener(destination = "${queue.account-interest.calculation}")
public void handleAccountInterestCalculationMessage(Message message) {
	var jsonPayload = JmsMessageUtil.getMessageContent(message);
	Assert.hasText(jsonPayload, "Failed to retrieve message contents
from incoming account interest calculation message.");
	var accountInterestCalculationMessage = objectMapper
		.readValue(jsonPayload,
	AccountInterestCalculationMessage.class);
	var command = getCommand(accountInterestCalculationMessage);
	calculateAccountInterestUseCase.execute(command);
}
```

Joonis 9. Kalkuleerimise järjekorrast sõnumite kuulamine.

Konto intresside agregeerimine on tähistatud *@Transactional* annotatsiooniga, mis tagab, et vea ilmnemisel tühistatakse kõik agregeerimise käigus tehtud muudatused. Samuti suunatakse töötlemisel olnud sõnum vigaste sõnumite järjekorda (ingl k *dead letter queue*). Taustaprotsessid tuvastavad sõnumi vigaste sõnumite järjekorras ja alustavad uuesti selle töötlemist. Selline lahendus tagab, et ajutised probleemid andmebaasiga ei takista intresside väljamakseprotsessi toimimist.

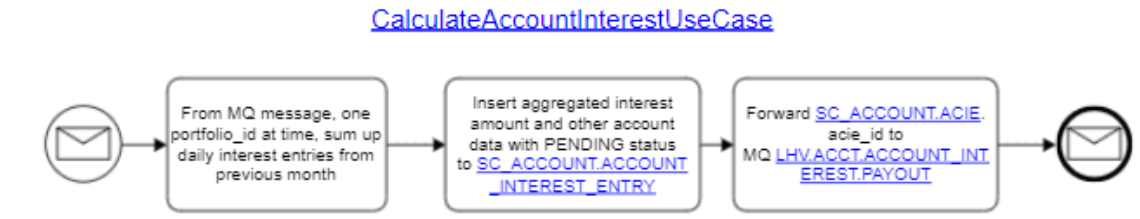
Enne konto intresside summeerimist kontrollitakse, et vastavalt kontol ei oleks juba intressi agregeeritud. Selleks tehakse päring *ACCOUNT_INTEREST_ENTRY* tabelisse, et veenduda, et vastava konto ja perioodi kohta ei ole juba kirjeid olemas. Kui sellised kirjed leitakse, luuakse *WARN* tasemega logikirje. Seejärel lõpetatakse vastava konto töötlemine ning alustatakse uue konto käsitlemist.

Olemasoleva *ACCOUNT_INTEREST_ENTRY* kirje puudumisel, käivitatakse andmebaasipäring (Joonis 10), mis tagastab agregeeritud tulemused *ACCOUNT_INTEREST_CACHE* tabelist. *ACCOUNT_INTEREST_CACHE* tabeli kirjed agregeeritakse kokku konto, kuu, valuuta, intressimäära ja raamatupidamiskonto numbri alusel. Summeeritud tulemused salvestatakse *ACCOUNT_INTEREST_ENTRY* tabelisse, kus iga kirje saab *PENDING* staatuse. Salvestamisel genereeritud *ID* saadetakse *PAYOUT* järjekorda.

```
@Query(value = """
       SELECT portfolio_id AS account_id,
              currency_id,
              period_code,
              interest_rate,
              erp_account_no,
              SUM(amount_interest) AS amount_interest
       FROM SC_ACCOUNT.account_interest_cache
       WHERE date >= :startDateTime
              AND date < :endDateTime
              AND period_code = :periodCode
              AND portfolio_id = :accountId
              AND adjustment_type IS NULL
              AND erp_account_no IN (4515, 4004)
              AND type = 'P'
       GROUP BY portfolio_id,
       currency_id,
       period_code,
       interest_rate,
       erp_account_no""", nativeQuery = true)
List<AccountInterestAggregatedJpaEntity>
findAggregatedAccountInterestEntriesForAccount(
       @Param("accountId") Long accountId,
       @Param("periodCode") Integer periodCode,
       @Param("startDateTime") LocalDateTime startDateTime,
       @Param("endDateTime") LocalDateTime endDateTime);
```

Joonis 10. Konto intresside agregeerimine.

Joonis 11 illustreerib Agregeerimisetappi ja etapi sees toimuvat protsessi.



Joonis 11. Agregeerimisetapi tehniline skeem.

4.5 Väljamakseetapp

Väljamakseetapp käivitub automaatselt, kui sõnum on lisatud *PAYOUT* järjekorda. Sõnum võetakse järjekorrast välja ning alustatakse selle töötlemist. Sõnumi sisust võetakse *ACCOUNT_INTEREST_ENTRY* tabeli *ID*, millest koostatakse väljamakseetapi sisendobjekt (*InterestPayoutUseCaseCommand*). Sisendobjektiga käivitatakse intressi väljamakset teostav alamprogramm (InterestPayoutUseCase) (Joonis 12).

```
@Transactional(propagation = Propagation.REQUIRED, timeout = 5)
public void execute(InterestPayoutUseCaseCommand command){
    var interestEntry = steps.prepareData(command);
    interestEntry.ifPresent((AccountInterestEntry entry) -> {
        var result = steps.createInterestPayoutTransaction(entry);
        entry.setReference(result.getReference());
        steps.createInterestTaxTransactionIfNeeded(entry);
        steps.createReverseRecord(entry);
        steps.updateInterestEntryStatusSuccess(entry);
        }
    );
}
```

Joonis 12. Intressi väljamaksmine.

Intressi väljamaksmiseks koostatakse alamprogrammiga (*PrepareInterestEntryDataForTransactionUseCase*) väljamakseinfoobjekt, mis peegeldab endas *ACCOUNT_INTEREST_ENTRY* tabeli kirje sisu. Lisaks olemasolevale *ACCOUNT_INTEREST_ENTRY* kirje infole, lisatakse väljamakseinfoobjekti konto, mille peale kantakse intressi tulu, ning vajadusel ka tulumaksuprotsent.

Pärast täiendava teabe kogumist, valideeritakse saadud tulemused ja alustatakse tehingu ettevalmistamist. Selle käigus määratakse väljamakse tehingule sobiv operatsioonitüüp ja tehingu suund, lähtudes väljamakseinfoobjekti raamatupidamiskonto lühinumbrist. Operatsioonitüüp aitab paremini organiseerida ja kategoriseerida makseid, mis võimaldab paremini hallata pangas toimuvaid tehinguid. Juhul kui väljamakseinfoobjekti raamatupidamiskonto lühinumber on 4515, määratakse tehingu suunaks *CREDIT* ja operatsioonitüübiks *INTR_DEMAND_DEPOSIT*, mis tähendab, et kliendile makstakse raha juurde. Kui lühinumber ei ole 4515, määratakse tehingu suunaks *DEBIT* ja operatsioonitüübiks *INTR_DEMAND_DEPOSIT_NEG*, mis tähendab, et kliendi kontolt võetakse raha maha.

Kliendi keele seadete alusel koostatakse tehingu kirjeldus. Kui kliendi keeleks on määratud eesti keel, koostab süsteem eestikeelse maksekirjelduse, muul juhul saab klient ingliskeelse kirjelduse. Tehingu kirjeldust koostav meetod võtab lisaargumendina sisendiks *boolean* väärtuse *isTaxTransaction*, mis võimaldab eristada, kas tegemist on intressi väljamakse või tulumaksukande makse tüübiga. Pärast makse tüübi määramist, kutsutakse esile tõlgete leidmismeetod (*TranslatableMessageHandler*), mis leiab õige maksekirjelduse *properties* failidest (Joonis 13).

```
@Component
@RequiredArgsConstructor
public class TranslatableMessageHandler {

    private final MessageSourceAccessor messageSourceAccessor;

    public String getTranslatableMessage(TranslatableMessage message,
Locale locale){
        if (message instanceof ParameterizedTranslatableMessage
parameterizedMessage) {
            return messageSourceAccessor.getMessage(
                parameterizedMessage.getFullTranslationCode(),
                parameterizedMessage.getParameters(),
                locale);
        }
        return messageSourceAccessor
             .getMessage(message.getFullTranslationCode(), locale);
          }
}
```

Joonis 13. Maksekirjelduse leidmise abiklass.

Maksekirjeldus koos ülejäänud olulise infoga, sealhulgas konto, operatsioonitüüp, summa, valuuta ja tehingusuund, lisatakse tehingu teostamise alamobjekti ning antud infoga sooritatakse väljamakse. Makse õnnestumisel tagastatakse makseviide, mis lisatakse väljamakseinfoobjekti.

Pärast intressi väljamaksmist kontrollitakse, kas on vajalik täiendava tulumaksukande teostamist. Selleks valideeritakse väljamakseinfoobjekti küljes olevat tulumaksuprotsenti. Veendutakse, et tulumaksuprotsendi väärtus ei oleks null ja intressi summa korrutis tulumaksuprotsendiga oleks vähemalt 0,01 eurot. Kui tingimused on täidetud, teostatakse uus makse, mille puhul tehingu suunaks määratakse *DEBIT*,

operatsioonitüübiks *INTR_TAX* ja summaks intressi summa korrutis tulumaksuprotsendiga.

Eduka tulumaksukande sooritamise puhul tagastatakse makseviide. Tagastatud makseviide, koos vastava konto numbri, summa ja tulumaksuprotsendiga, salvestatakse *ACCOUNT_INTEREST_TAX* tabelisse. Tulumaksu andmete talletamine selles tabelis, võimaldab lihtsat auditeerimist ning tagab, et kõik maksukohustused on täidetud (Joonis 14).

```
public void createInterestTaxTransactionIfNeeded
(AccountInterestEntry entry)
{
    if (taxTransactionNotRequired(entry)) {
        return;
    }
    var result = executeTaxTransaction(entry);
    createAccountInterestTaxEntry(entry, result.getReference());
}
```

Joonis 14. Tulumaksukande teostamine.

Peale maksete sooritamist on vaja lisada *ACCOUNT_INTEREST_CACHE* tabelisse auditeerimiskirje, mis võimaldab analüütikutel kontrollida, et kogu kliendile arvutatud intress on täielikult välja makstud. Selleks käivitatakse alamprogramm (*CreateAccountInterestDailyReverseRecordUseCase)*, mille sisendiks on väljamakseinfoobjekt, mille summa muudetakse negatiivseks.

Antud sisendi alusel, luuakse *ACCOUNT_INTEREST_CACHE* tabeli kirje, mille tüübiks määratakse R, mis eristab seda tavapärastest intressikirjetest. Auditeerimise protsessis liidetakse kokku auditeerimiskirjed intressi arvutuse kirjetega. Kui intressid on täies mahus välja makstud, peab tulemuseks olema null.

Pärast auditeerimiskirje loomist, määratakse väljamakseinfoobjekti staatuseks *SUCCESS* ja lisatakse kommentaariks *COMPLETED*. Seejärel salvestatakse uuendatud kirje *ACCOUNT_INTEREST_ENTRY* tabelisse. Peale kirje salvestamist algab väljamakseetapp uuesti. Võetakse töötlemiseks uus sõnum *PAYOUT* järjekorrast ning seda protsessi korratakse, kuni kõik järjekorra sõnumid on täielikult töödeldud. Lisa 2 illustreerib väljamakse etappi ja selles toimuvaid protsesse.

4.5.1 Väljamakse etapi veahaldus

Väljamakse teostamise alamprogramm (*InterestPayoutUseCase*) on varustatud annotatsiooniga *@Transactional(propagation = Propagation.REQUIRED, timeout = 5)*, mis tagab andmete terviklikkuse. Vea tekkimisel, tühistatakse kõik andmebaasi muudatused, et vältida olukorda, kus tehing oleks sooritatud, kuid auditeerimiskirje jääks tegemata. Samuti, kui väljamakseprotsess ületab 5-sekundilise ajapiirangu, tühistatakse kõik muudatused, et vältida protsessi seisma jäämist ühe vigase kirje tõttu. Vea tekkimisel muudetakse vastava *ACCOUNT_INTEREST_ENTRY* kirje staatus *FAILED*'iks ja lisatakse kommentaariks *INTEREST_PAYOUT_FAILED*. Seejärel luuakse *TechnicalException* erind (ingl k *exception*), kuhu lisatakse täiendava infona nii *ACCOUNT_INTEREST_ENTRY ID* kui ka esialgne vea kirjeldus (Joonis 15).

```
@SneakyThrows
@JmsListener(destination = "${queue.account-interest.payout}")
public void handleAccountInterestPayoutMessage(Message message) {
	var jsonPayload = JmsMessageUtil.getMessageContent(message);
	Assert.hasText(jsonPayload, "Failed to retrieve message contents
from incoming account interest payout message.");
	var accountInterestPayoutMessage = objectMapper
		.readValue(jsonPayload, AccountInterestPayoutMessage.class);

	try {
		interestPayoutUseCase
			.execute(getCommand(accountInterestPayoutMessage));
	} catch (RuntimeException e){
		handleExecutionFailure(accountInterestPayoutMessage);
		throw new TechnicalException(String.format("Interest payout
failed for account interest entry ID %s. Reason: %s",
				accountInterestPayoutMessage
					.getAccountInterestEntryId(),
				e.getMessage()), e);
	}
}

private void handleExecutionFailure(AccountInterestPayoutMessage message){
	accountInterestEntryRepository
		.findById(message.getAccountInterestEntryId())
		.ifPresent(
			(AccountInterestEntry entry) -> {
			entry.setStatus(AccountInterestEntryStatus.FAILED);
			entry.setLogComment(
				AccountInterestEntryLogComment
					.INTEREST_PAYOUT_FAILED);
			accountInterestEntryRepository.save(entry);
		});
}
```

Joonis 15. Väljamakse etapi veahaldus.

4.5.2 Intressi väljamakse info kogumine

Intressi väljamakseks vajaminev lisainfo leitakse *PrepareInterestEntryDataForTransactionUseCase* alamprogrammi abil. Programm saab sisendiks *ACCOUNT_INTEREST_ENTRY* kirje *ID*, mille põhjal leitakse andmebaasist vastav kirje. Pärast kirje leidmist, luuakse sellest väljamakseinfoobjekt ning veendutakse, et objekti küljes olev konto, mille pealt arvutati intressi, oleks endiselt aktiivne.

Kui konto on suletud, siis tuleb kogunenud intress kanda panga raamatupidamiskontole, kus raamatupidajad saavad sellega edasi tegeleda. Kõigepealt hakatakse koostama raamatupidamiskannet, mille käigus kontrollitakse *ACCOUNT_INTEREST_ENTRY* kirje küljes olevat raamatupidamiskonto lühinumbrit. Kui konto lühinumber on 4515, määratakse raamatupidamiskande operatsioonitüübiks *INTR_CLOSED_ACCOUNT*. Kui tegemist on mõne muu lühinumbriga, siis määratakse operatsioonitüübiks *INTR_NEG_CLOSED_ACCOUNT*. Need operatsioonitüübid tagavad, et kogunenud intress jõuaks õigele raamatupidamiskontole.

Seejärel lisatakse raamatupidamiskandele *ACCOUNT_INTEREST_ENTRY* kirjes olev summa, valuuta ning vastavalt operatsioonitüübile ka kommentaar. Pärast seda sooritatakse raamatupidamiskanne ning luuakse *ACCOUNT_INTEREST_CACHE* tabelisse auditeerimiskirje. Kui raamatupidamiskande ja auditeerimiskirje loomised on edukad, siis uuendatakse olemasoleva *ACCOUNT_INTEREST_ENTRY* kirje staatust. Uueks staatuseks saab *ABANDONED* ning kommentaariks määratakse *ACCOUNT_CLOSED*. Peale suletud konto haldamist, logitakse süsteemis maha *WARN* tasemega logikirje ja lõpetatakse antud *ACCOUNT_INTEREST_ENTRY* kirje töötlemine (Joonis 16).

```
protected void handleClosedAccount(AccountInterestEntry entry){
    var result = executeInternalAccountingTransactionUseCase
.execute(getAccountingTransactionCommand(entry));
    createAccountInterestDailyReverseRecordUseCase.execute(entry);

    entry.setReference(result.getReference());
    entry.setLogComment(AccountInterestEntryLogComment.ACCOUNT_CLOSED);
    entry.setStatus(AccountInterestEntryStatus.ABANDONED);
    accountInterestEntryRepository.save(entry);
}
```

Joonis 16. Suletud konto haldamine.

Peale intressi arvutuskonto aktiivse staatuse valideerimist, leitakse konto, millele tuleb kogunenud intressitulu välja maksta. Selleks käivitatakse alamprogramm (*FindInterestPayoutAccountUseCase*), mille sisendiks on intressi teeninud konto ja raamatupidamiskonto lühinumber.

Esmalt valideeritakse, et raamatupidamiskonto lühinumber oleks kas 4515 või 4004. Lühinumbri 4515 korral leitakse konto, millele saab kliendi intressitulu välja maksta. Lühinumbriga 4004 otsitakse konto, millelt tuleb intressisumma maha võtta. Juhul, kui kliendi konto saldo ületab intressitulu teenimislävendit, hakkab klient teenima negatiivset intressi. See tähendab, et intressitulu teenimise asemel, tuleb kliendile nõue raha hoiustamise eest pangas.

Olukorras kui raamatupidamiskonto lühinumber on 4515, alustatakse sobiva väljamakse konto otsingut, lähtudes konto tüübist. Tavalise arvelduskonto korral määratakse väljamakse kontoks sama konto, millelt teeniti intressitulu. Kui tegemist on *SAFEGUARDING* kontoga, leitakse *SAFEGUARDING* konto tulude ja kulude haldamiseks mõeldud arvelduskonto. Kui sellist kontot ei õnnestu leida või kui konto on suletud, logitakse süsteemi maha *ERROR* tasemega logikirje. Kui kontoks on määratud *KONTSER*, siis leitakse kontserni liikmete seast konto, mis on määratud saama kogu kontserni intressitulu (Joonis 17).

Juhul kui väljamaksekonto leidmine ebaõnnestub, käivitub veahaldus ning vastava *ACCOUNT_INTEREST_ENTRY* kirje töötlemine lõpetatakse. Kirje staatuseks

määratakse *FAILED* ja kommentaariks *PAYOUT_ACCOUNT_NOT_FOUND*. Seejärel uuendatakse kirje andmebaasis ning luuakse *ERROR* tasemega logikirje.

```
private Optional<Long> getCashPoolingInterestFeeAccount(Long accountId){
    return findAllCashPoolingMemberAccounts(accountId).stream()
        .filter(CashPoolingMember::isDefaultInterestAccrued)
        .findFirst()
        .map(CashPoolingMember::getAccountId);
}
```

Joonis 17. Kontserni intressi väljamakse konto leidmine.

Pärast intressi väljamaksekonto leidmist, lisatakse see väljamakseinfoobjekti külge. Seejärel kontrollitakse, kas intressi teeninud konto omanik peab maksma tulumaksu (Joonis 18). Selleks käivitatakse tulumaksu leidmise alamprogramm (*GetInterestTaxPercentageUseCase)*, mille sisendiks on intressi teeninud konto.

```
var payoutAccountId = steps.getInterestPayoutAccountId(entry.get());
return payoutAccountId.map(accountId ->
    steps.addTaxPercentageAndPayoutAccountToInterestEntry(entry.get(),
accountId));

public AccountInterestEntry
addTaxPercentageAndPayoutAccountToInterestEntry(AccountInterestEntry
entry, Long payoutAccountId){
    entry.setPayoutAccountId(payoutAccountId);
    entry.setTaxPercentageMark(getTaxPercentage(entry));
    return entry;
}
```

Joonis 18. Lisainfo liitmine väljamakseinfoobjekti külge.

Alamprogrammis käivitatakse *isEligibleForTax* meetod, mille käigus otsitakse andmebaasist intressi teeninud konto ja liidetakse sellele konto omaniku info. Järgmisena kontrollitakse, et konto omanik oleks eraisik, mitte juriidiline isik. Seejärel veendutakse, et konto, millel intress arvutati, ei oleks tavaline investeerimiskonto ega pensioni investeerimiskonto. Investeerimiskontosid ei maksustata, kui nende peale kantakse intressi tulu. Kontosid maksustatakse siis, kui klient võtab raha investeerimiskontolt välja.

Lõpuks kontrollitakse, kas klient on Eesti maksuresident. Selleks otsitakse konto omaniku *ID* järgi *CUSTOMER_TAX_RESIDENCE* tabelist kõik kliendi aktiivsed maksuresidentsuse kirjed (Joonis 19). Need kirjed järjestatakse algusaja järgi (Joonis 20). Kui ükski tema aktiivne maksuresidentsus ei ole Eesti maksuresidentsus, siis klient ei pea koheselt tulumaksu tasuma. Sellisel juhul tagastatakse tulumaksuprotsendiks 0. Kui maksuresidentsus on Eesti või kliendil puudub maksuresidentsuse kirje, siis tagastatakse maksuprotsendiks 0,2.

```
private boolean isEligibleForTax(GetInterestTaxPercentageUseCaseCommand
command) {
    var account = accountEntityService.getAccount(command.getAccountId());
    var customer = account.getCustomer();

    return customer.isPrivate()
        && accountIsEligibleForTaxation(account)
        && hasActiveEETaxResidency(customer.getCustomerId());
}
```

Joonis 19. Tulumaksu leidmine.

```
@Query(value = """
    SELECT r FROM CustomerTaxResidencyJpaEntity r
    WHERE r.customerId = :customer_id
        AND r.startTime < OFFSET DATETIME
        AND (r.endTime IS NULL OR r.endTime > OFFSET DATETIME)
    ORDER BY r.startTime""")
List<CustomerTaxResidencyJpaEntity>
findActiveTaxResidencies(@Param("customer_id") Long customerId);
```

Joonis 20. Maksuresidentuse leidmine.

Kui intressi väljamakse konto ja tulumaksuprotsent on liidetud väljamakseinfoobjekti külge, siis tagastatakse objekt *InterestPayoutUseCase* programmi.

4.6 Kasutajaliides

Protsessist parema ülevaate saamiseks loodi kasutajaliides *Accounts-Frontend* rakendusse, mis võimaldab omada selget ülevaadet väljamakstavatest intressidest. *Accounts-Frontend* on rakendus, mis lihtsustab *Accounts* rakenduses olevate

äriprotsesside haldamist ja auditeerimist. Rakendus pakub tooteomanikele ülevaadet nende toodetest ning võimalust neid hallata.

Intressi väljamakse protsessi info kuvamiseks, saadab *Accounts-Frontend HTTPS*-päringu *Accounts* rakendusse. Päringu sisuks on andmeedastusobjekt (ingl k *data transfer object*), mis sisaldab endas järgmisi välju: intressi teeninud konto *ID*, väljamakse konto *ID*, periood, mille eest intressi teeniti, *ACCOUNT_INTEREST_ENTRY* kirje staatus ja kommentaar. Antud väljade põhjal toimub kuvatavate *ACCOUNT_INTEREST_ENTRY* kirjete filtreerimine.

Accounts rakenduses valideeritakse sissetulevat andmeedastusobjekti. Selleks kasutatakse annotatsioone, et tagada konto *ID*-de suuruse vastavus *Accounts* rakenduse kriteeriumitele. Samuti kontrollitakse *ACCOUNT_INTEREST_ENTRY* staatust ja kommentaari. Selleks proovitakse *STRING* tüüpi sisend konverteerida vastavaks *ENUM* tüüpi objektiks. Kui staatus või kommentaar ei vasta *ENUM'*ites olevatele väärtustele, luuakse erind ja päringu töötlemine lõpetatakse. Lisaks sisaldab andmeedatusobjekt leheküljel kuvatavate kirjete arvu ja lehekülje numbrit, mis aitab tagada, et korraga ei päritaks liiga palju andmeid.

Kui päring vastab nõuetele, genereeritakse *Spring Frameworki Specification* funktsionaalsuse abil andmebaasi päring (Joonis 21) (Joonis 22). *Specification* funktsionaalsus aitab luua paindlike ja dünaamilisi andmebaasi päringuid [13]. Selline lähenemine tagab andmebaasi päringute efektiivsuse ning võimaldab rakendada erinevaid otsingukriteeriume.

```
@Override
public Page<AccountInterestEntry>
getDemandDepositPayoutEntries(DemandDepositPayoutReportCommand command) {
    var specification = new
DemandDepositPayoutReportSpecification(command.getFilter());
    var pageable = PageRequest.of(command.getPageNumber(),
command.getPageSize());
    return jpaRepository.findAll(specification, pageable)
        .map(jpaEntityMapper::toDomain);
}
```

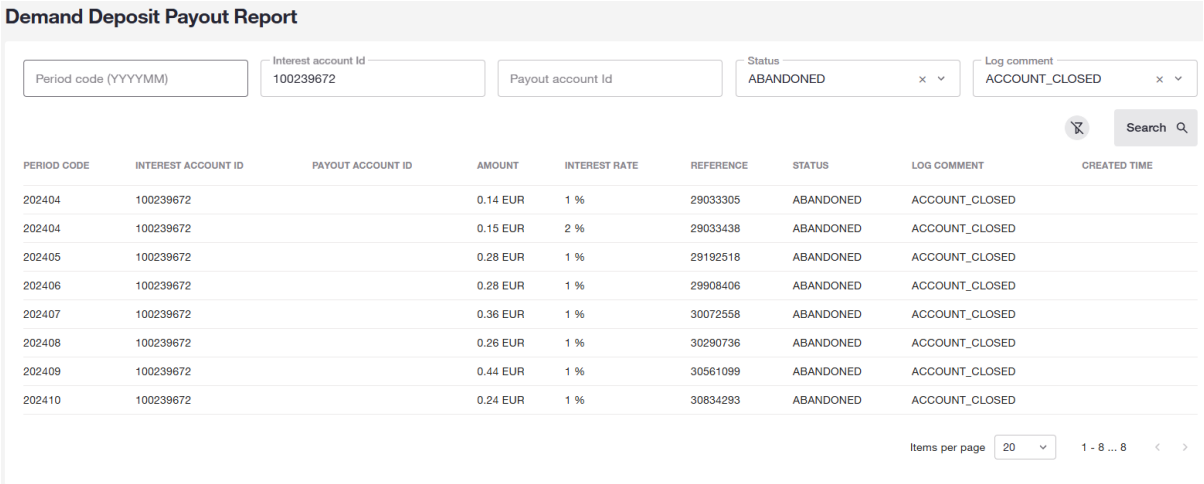
Joonis 21. *ACCOUNT_INTEREST_ENTRY* tabeli kirjete pärimine.

Lisaks andmeedastusobjektis sisalduvatele väljadele, tagastatakse andmebaasist järgmised väljad: tulumaksuprotsent, valuuta, teenitud intressiprotsent ja tehingu viitenumber. Andmebaasist leitud kirjed konverteeritakse ettenähtud kujule ning saadetakse päringu vastuse osana tagasi.

```
@Override
public Predicate toPredicate(Root<AccountInterestEntryJpaEntity> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
    var predicates = new ArrayList<Predicate>();

    if (filter.getInterestAccount() != null){
        predicates.add(cb.equal(root.get("interestAccountId"),
filter.getInterestAccount()));
    }
    if (filter.getPayoutAccount() != null){
        predicates.add(cb.equal(root.get("payoutAccountId"),
filter.getPayoutAccount()));
    }
    if (filter.getPeriodCode() != null){
        predicates.add(cb.equal(root.get("periodCode"),
filter.getPeriodCode()));
    }
    if (filter.getStatus() != null){
        predicates.add(cb.like(cb.upper(root.get("status")), "%" +
filter.getStatus() + "%"));
    }
    if (filter.getLogComment() != null){
        predicates.add(cb.like(cb.upper(root.get("logComment")), "%" +
filter.getLogComment().toUpperCase(Locale.ENGLISH) + "%"));
    }
    query.orderBy(cb.desc(root.get("createdDtime")));

    return cb.and(predicates.toArray(new Predicate[0]));
}
```

Joonis 22. Dünaamilise andmebaasipäringu ettevalmistamine.

Accounts-Frontend töötleb *Accounts* rakendusest saadud vastust ning kuvab kasutajale ülevaate *ACCOUNT_INTEREST_ENTRY* kirjetest (Joonis 23).



Joonis 23 Väljamakstud intresside kasutajaliides

4.7 Süsteemi testimine

Intressi väljamakse protsessi erinevaid äriloogilisi komponente testiti eraldi kasutades integratsiooniteste. Selle lähenemise abil valideeriti iga komponendi korrektne toimimine.

Kogu protsessi korrektsuse kontrollimiseks käivitati manuaalselt terve intressi väljamakse protsess, kasutades *JobExecutionController* klassi. See klass võimaldab *HTTPS* päringuga käivitada erinevaid protsesse *Accounts* rakenduses. Intressi väljamakse protsessi käivitamiseks, peab protsessi klass materialiseerima (ingl k *implement*) *JobExecution* liidest (ingl k *interface*). Seejärel tuleb koostada *HTTPS* päring, millele antakse kaasa *URL* parameetrina protsessi nimi *ExecuteInterestPayoutJob*. *JobExecutionController* kasutab sõltuvussüsti (ingl k *dependecy injection*) loogikat, et leida kõik klassid, mis materialiseerivad *JobExecution* liidest. *URL* parameetri alusel tuvastab *JobExecutionController* õige protsessi ja käivitab selle.

Intressi väljamakse protsessi käivitav *HTTPS* päring võimaldab lisaparameetritena kaasa anda konto *ID*'d või kliendi *ID*'d, kellele intressi välja maksta.

Koostati kolm peamist testjuhtu, et kontrollida intressi väljamakse protsessi:

1. Intressi väljamakse protsess käivitatakse ühe konto *ID'*ga, et veenduda, et intressi arvutatakse ja makstakse välja vaid valitud kontole.
2. Intressi väljamakse protsess käivitatakse ühe kliendi *ID'*ga, et tagada intresside korrektne väljamaksmine ainult valitud kliendi kontodele.
3. Intressi väljamakse protsess käivitatakse ilma lisaparameetriteta, et kõikidele sobivatele kontodele arvutatud intressid välja maksta. Samuti tuleb tagada, et ei toimuks topelt väljamakseid.

Kontrolliprotsess hõlmas järgmisi tegevusi:

- Jälgiti kõiki protsessi poolt genereeritud logi kirjeid, et tagada süsteemi täpsus ja tuvastada võimalikud vead.
- Kasutati andmebaasi päringuid, et kontrollida protsessi poolt loodud andmebaasi kirjeid ja tagada kõikide väljamaksete ja arvutuste korrektsust (Joonis 24).
- *Zabbixi* tööriistade kaudu jälgiti süsteemi koormust, et tuvastada, uue intressi väljamakse protsessi potentsiaalne koormus olemasolevale andmebaasile.
- Internetipangas veenduti, et väljamakstud intress vastab arvutatud summale ja kõik väljamaksed on edukalt sooritatud õigele kontole.

Keskkonnad, kus toimus testimine:

- Lokaalne keskkond. Lokaalne keskkond kujutas endast sülearvuti peal käivitatud *Accounts* rakendust, kus kõik testandmed tuli testijal ise luua.
- Test keskkond. Test keskkond kujutab endast ettevõtte ülest platvormi, kus kõik arendustiimid ja tooteomanikud testivad enda uusi arendusi. See tekitab olukorra, kus andmekvaliteet jätab soovida ning suuremahuline testimine ei pruugi soovitud tulemust anda. Kolmanda testjuhu kontrollimine jäeti test keskkonnas pooleli, sest andmekvaliteedi korrastamine võttis rohkem aega, kui testimine ise.
- *Prelive* keskkond. *Prelive* keskkonnas ei toimu arendustiimide ja tooteomanike suuremahulist testimist. Keskkonda kasutavad ettevõtte välispartnerid, et enda rakendusi testida. See tagab *prelive* keskkonnas parema andmekvaliteedi kui test keskkonnas. *Prelive* keskkond lubas edukalt läbi viia intressi väljamakse protsessi esimesed suuremahulised testimised.
- *Staging* keskkond. *Staging* keskkonna andmebaas kopeeritakse otse toodangu keskkonna andmebaasist kord päevas, mis tagab selle, et tegeletakse võimalikult

realistlike andmetega. *Staging* keskkonna andmekvaliteet on suurepärane. *Staging* keskkonnas pandi lisaks uuele intressi väljamakse protsessile tööle ka vana intressi makse protsess. Mõlemad protsessid said sisendandmed *ACCOUNT_INTEREST_CACHE* tabelist. Uus intressi väljamakse protsess kirjutas enda väljamakstavad intressid *ACCOUNT_INTEREST_ENTRY* tabelisse. Vana protsessi väljamakstavad intressid võeti sooritatud tehingute tabelist, kus nad lisati *TEMP_OLD_INTERESTS* tabelisse. Andmebaasi päringutega võrreldi kahe tabeli tulemusi, mis võimaldas veenduda, et väljamakstavad intressi summad olid võrdsed. Lisaks väljamakstud intressile kontrolliti üle *ACCOUNT_INTEREST_CACHE* tabelisse tekkinud auditeerimiskirjed.

Joonisel 24 kujutatud andmebaasi päring on koostatud teise arendaja poolt.

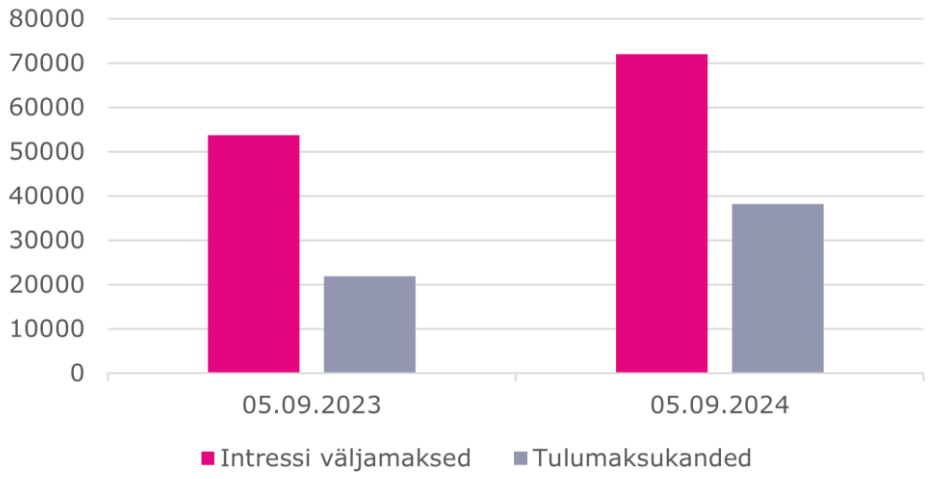
```
SELECT * FROM (
    SELECT
      NEW.account_id,
      NEW.currency_name,
      NEW.interest_rate AS new_interest_rate,
      SUM(ACIC.interest_rate) AS old_interest_rate,
      NEW.erp_account_no AS new_erp_account_no,
      SUM(ACIC.erp_account_no) AS old_erp_account_no,
      NEW.interest_amount AS new_interest_amount,
      -1 * SUM(ACIC.amount_interest) AS old_interest_amount
    FROM (
            SELECT DISTINCT
                ACIE.interest_account_id AS account_id,
                SUM(ACIE.amount) AS interest_amount,
                ACIE.currency AS currency_name,
                ACIE.interest_rate,
                ACIE.erp_account_no
            from SC_ACCOUNT.account_interest_entry ACIE (NOLOCK)
            where ACIE.period_code = 202307
            group by ACIE.currency, ACIE.interest_rate,
ACIE.erp_account_no, ACIE.interest_account_id
       ) AS NEW
            LEFT JOIN SC_ACCOUNT.currencies CURR (NOLOCK) ON
CURR.shortname = NEW.currency_name
           LEFT JOIN SC_ACCOUNT.account_interest_cache ACIC (NOLOCK)
ON ACIC.portfolio_id = NEW.account_id
      AND ACIC.adjustment_type = 'R'
      AND ACIC.currency_id = CURR.currency_id
      AND ACIC.period_code = 202307
      AND ACIC.interest_rate = NEW.interest_rate
    GROUP BY NEW.account_id, NEW.currency_name, NEW.erp_account_no,
NEW.interest_rate, NEW.interest_amount
) QUERY
WHERE new_interest_amount != old_interest_amount
```

Joonis 24. Intressi väljamakse protsessi kontrollpäring.

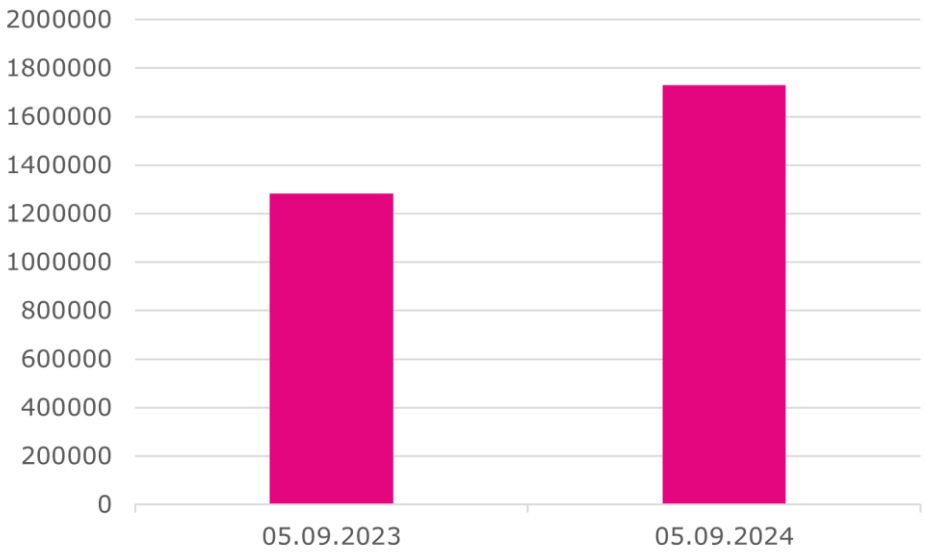
5 Tulemuste analüüs

Uue protsessi jaoks loodud funktsionaalsed ja mitte funktsionaalsed nõuded said kõik täidetud. Uus lahendus parandas ära andmebaasi tupikute ja koormuse probleemid ning andis hea ülevaate tervest protsessist. Täiustatud veahaldus vähendas oluliselt protsessile kuluvat haldusaega.

Protsess käivitus esimest korda toodangu keskkonnas 05.09.2023. Uuele lahendusele üleminek võimaldas koostada klientidele individuaalseid väärtuspakkumisi. See aitas jõudsalt kasvatada nõudmiseni hoiuse intressi teenivate klientide arvu. Joonis 25 ja 26 illustreerivad andmebaasi mahtude kasvu pärast uue lahenduse käivitamist toodangus.



Joonis 25. Intressi väljamaksete andmemahu kasv.



Joonis 26. Arvutatud intressi kirjete arv.

Protsess sai edukalt hakkama andmebaasi mahtude kasvuga, kuid protsessi ajaline kestvus venis andmemahu kasvamise tõttu tunduvalt pikemaks.

06.01.2025 seisuga on uus intressi väljamakse protsess teinud kokku 1005074 intressi väljamakset. 460 väljamakset on pooleli jäetud, sest konto, millele intressi tuli maksta, oli suletud. Üks väljamakse on ebaõnnestunud andmekvaliteedi praagi tõttu. Ebaõnnestunud intressi väljamakset ei tehta uuesti, sest väljamakstav intressi summa on 0.

5.1 Esinenud probleemid

Peamine murekoht antud projekti tegemisel, oli kogu protsessi jagamine väikesteks osadeks, et seda saaks osade kaupa arendada ja testida. Lahendus probleemile oli see, et kaardistati kogu protsess ära ja jagati äriloogiliselt erinvateks tükkideks. Tükkidele määrati kindel sissetuleva ja väljamineva info struktuur, mille alusel sai tükid eraldiseisvalt valmis teha.

Intressi väljamakse protsessi esmasel käivitumisel toodangu keskonnas, tekkis kaks probleemi:

- Eeldus, et kõikidel klientidel on olemas maksuresidentsuse kirje, osutus valeks. Nendelt klientidelt, kellel puudus maksuresidentsuse kirje, aga kes olid Eesti

maksuresidendid, ei võetud koheselt tulumaksu maha. Klientidel tuli tulumaks maha võtta andmebaasi päringuga samal päeval.

- Intressimaksed tehti vale operatsioonitüübiga. Operatsioonitüüp intressi väljamakse küljes määrab ära, milliselt raamatupidamiskontolt tuli kliendile raha välja maksta. Kasutatud operatsioonitüüp võttis raha õige raamatupidamiskonto pealt, aga raamatupidamisraportites ei kuvatud väljamakse summat õige kategooria all. Probleem parandati andmebaasi päringuga, mis määras sooritatud intressi masketele õige operatsioonitüübi.

5.2 Edasi arendamist vajavad kohad

Pärast prototüübi valmimist otsustati analüüsida loodud süsteemi ning määratleda prioriteedid edasistele arendustele intressi väljamakse süsteemis.

Analüüsi tulemusena tehti ettepanek kaheks lisaarenduseks:

- **Funktsionaalsus, mis võimaldaks klientidel valida konto, kuhu laekub intressitulu.** Hetkel selline võimalus puudub ning intressitulu kantakse automaatselt kontole, millele see arvutati. Uue funktsionaalsuse lisamine parandaks kliendi kasutajakogemust, võimaldades neil saada paremat ülevaadet teenitud nõudmiseni hoiuse intressidest. Tegemist on mahuka arendusega, mis nõuab põhjalikku kliendianalüüsi ning tihedat koostööd teiste arendusmeeskondadega. Arenduse ulatuse ja keerukuse tõttu otsustati selle teostamine edasi lükata.
- ***ACCOUNT_INTEREST_CACHE_LOOKUP* tabeli eemaldamine**. Antud tabeli eesmärk on valideerida, et intress makstakse kõikidele kontodele välja täpselt üks kord kuus. Võimalik on luua uus tabel, mis täidab sama funktsiooni ja mida on võimalik kasutada erinevates protsessides, kus on vajalik unikaalsuse kontroll. Arendus võimaldaks tulevikus vältida protsessipõhise unikaalsuse kontrolli jaoks eraldi tabelite loomist, mis omakorda vähendaks arendusele kuluvat aega. Madala prioriteetsuse tõttu otsustati selle arenduse teostamine edasi lükata

6 Kokkuvõte

Lõputöö eesmärgiks oli luua AS-ile LHV Pank paindlik ja efektiivne lahendus nõudmiseni hoiuse intressi väljamaksete teostamiseks. Selle saavutamiseks viidi esmalt läbi olemasoleva protsessi kaardistamine, mis võimaldas tagada LHV panga klientidele võimalikult sujuva ülemineku uuele süsteemile. Projekti raames koguti uue protsessi nõudeid ning koostati visioon, milline võiks välja näha tulevikus intressi väljamakse süsteemi haldamine, probleemide lahendamine ja äriloogika muutmine.

Prototüübi loomise ettevalmistamiseks viidi vana intressi väljamakse protsessiga seotud tabelid *Accounts* rakenduse andmebaasi skeemi alla. Samuti loodi uued *ActiveMQ* järjekorrad, mis võimaldavad äriloogiliste komponentide vahel andmeid efektiivselt jagada. Arendati välja võimekus sünkroniseerida maskuresidentsuse andmeid *Customer* rakendusest *Accounts* rakendusse.

Pärast ettevalmistavaid töid koostati uue intressi väljamakse süsteemi prototüüp, mis testiti põhjalikult läbi ja kaeti 100-protsendiliselt automaattestidega. Süsteemile loodi ka kasutajaliides, mis annab tooteomanikele ja arendajatele põhjaliku ülevaate välja makstud intresside kohta.

Pärast uue süsteemi kasutuselevõttu kadusid andmebaasist varasemad probleemid, mis olid seotud intressi väljamakse protsessiga. See tagas, et uue protsessi raames saavad kliendid teha makseid ja vaadata konto väljavõtet ilma tehniliste tõrgeteta. Pärast esialgsete vigade parandamist ei ole süsteemis uusi probleeme ilmnenud. Edasised arendused keskenduvad uue funktsionaalsuse loomisele, mis võimaldaks klientidel valida konto, kuhu intress välja makstakse.

Kasutatud kirjandus

[1] J. Chen, investopedia, 05.08.2024. [Võrgumaterjal]. Available: https://knowledge.anbtx.com/money-management/banking/article/what-is-a-demand-deposit. Kasutatud 21.10.2024.

[2] A. Shalimov, Easternpeak, 23.08.2023. [Võrgumaterjal]. Available: https://easternpeak.com/blog/big-data-in-banking-and-financial-services/. Kasutatud 23.12.2024.

[3] P. Taylor, Statista, 21.11.2024. [Võrgumaterjal]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/. .Kasutatud 23.12.2024.

[4] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2009.

[5] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 2003.

[6] P. Allies, Nanosoft, 19.02.2023. [Võrgumaterjal]. Available: https://nanosoft.co.za/blog/post/clean-architecture-use-cases. Kasutatud 01.05.2024.

[7] V. Khorikov, Enterprise Craftsmanship, 11.11.2015. [Võrgumaterjal]. Available: https://enterprisecraftsmanship.com/posts/is-sql-good-place-for-business-logic/. Kasutatud 25.12.2024.

[8] P. Jansen, Tiobe, [Võrgumaterjal]. Available: https://www.tiobe.com/tiobe-index/. Kasutatud 21.10.2024.

[9] L. S. Vailshery, Statista, 18.09.2024. [Võrgumaterjal]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. Kasutatud 21.10.2024.

[10] Activemq, [Võrgumaterjal]. Available: https://activemq.apache.org/. .Kasutatud 12.10.2024.

[11] Spring, VMware, [Võrgumaterjal]. Available: https://spring.io/why-spring . Kasutatud 12.10.2024.

[12] M. Ghanayem, Synonyms (Database Engine), Microsoft, 05.11.2023. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/synonyms/synonyms-database-engine?view=sql-server-ver16#using-synonyms. Kasutatud 13.10.2024.

[13] Specifications, VMware, [Võrgumaterjal]. Available: https://docs.spring.io/spring-data/jpa/reference/jpa/specifications.html. Kasutatud 13.10.2024.

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

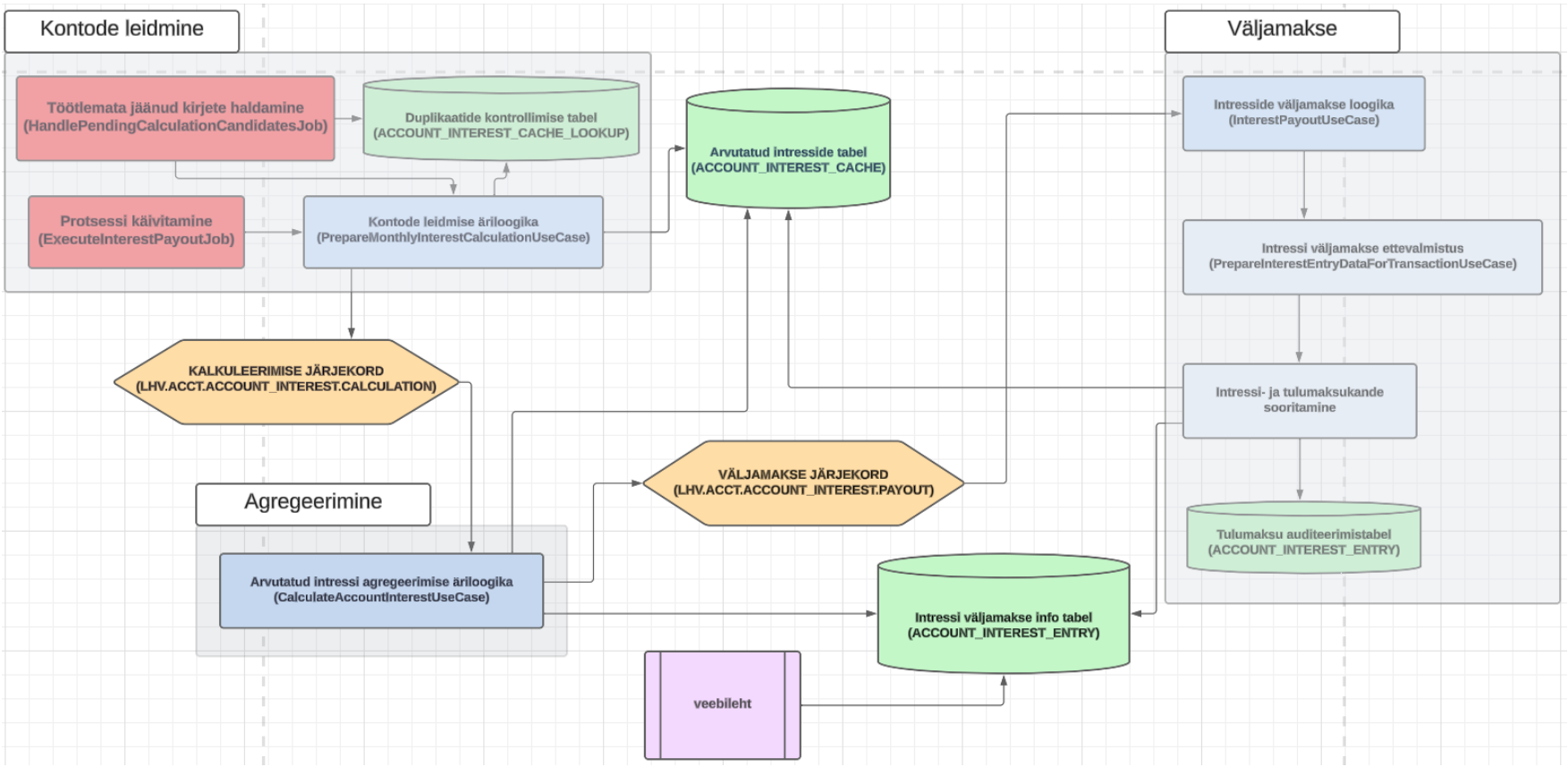
Mina, Kaur-Kristofer Remmel

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Nõudmiseni hoiuse intresside väljamakse süsteemi kaasajastamine", mille juhendaja on Toomas Lepikult ja kaasjuhendaja Erik Ehrbach.
    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.

06.01.2025

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 – Uue nõudmiseni hoiuse väljamakse skeem



Lisa 3 – Väljamakse etapi skeem