TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

ITV40LT

Rein-Sander Ellip 112989

RELATSIOONILISTE ANDMEBAASIDE PIDEVA SÜNKRONISEERIMISE RAKENDUSE PLATVORM

Bakalaureusetöö

Juhendaja: Ago Luberg
magister
assistent

Tallinn 2016

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: Rein-Sander Ellip

22.05.2016

Annotatsioon

Antud lõputöö eesmärgiks on kirjeldada Java rakendust, millega on võimalik erinevate andmemudelitega erinevate relatsiooniliste andmebaasisüsteemide vahel andmeid sünkroniseerida, ning sellega seonduvat funktsionaalsust.

Andmemuudatuste sünkroniseerimine andmebaaside vahel peaks toimuma asünkroonselt peaaegu reaalajas ning omama minimaalselt mõju ja seoseid teiste andmebaasi tarbijate tegevusega.

Töö tulemusena valmib põhjalik kirjeldus, kuidas antud probleemi saab lahendada, koos selleks vajaliku programmi asjakohaste koodinäidetega. Töös võrreldakse loodud lahendust ka juba olemasolevate programmidega, mis analoogsete probleemide lahendamisega tegelevad.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 39 leheküljel, 7 peatükki, 5 joonist, 1 tabelit.

Abstract

Application platform for continuous synchronization of relational databases

The goal of this thesis is to describe a Java application and related functionality, which would be able to continuously synchronize data between different relational databases with different data models.

Synchronization of data changes between databases should work asynchronously in near real-time and should not intertwine with and have minimal effect on other database users.

The result of this thesis should be a detailed description of an application and related functionality which solves the aforementioned problem with relevant code examples. The result also contains comparison of the created solution with similar software.

The thesis is in Estonian and contains 39 pages of text, 7 chapters, 5 figures, 1 table.

Lühendite ja mõistete sõnastik

REST-API	**RE**presentational **S**tate **T**ransfer, **A**pplication **P**rogramming **I**nterface REST arhitektuurile ja omadustele vastavaid HTTP teenuseid pakkuv rakenduse liides. [36]
NRT	**N**ear **r**eal-**t**ime Peaaegu reaalajas töötavateks süsteemideks, nimetatakse süsteeme, mis toimivad sisuliselt reaalajas kuid mitte sellise täpsusega nagu reaalaja süsteemid. [31]
SoC	**S**eparation **o**f **C**oncerns Ülesannete lahususe printsiip. Disaini printsiip, mille järgi tuleb programm jagada eristatavateks sektsioonideks niimoodi, et iga sektsioon tegeleb konkreetse ülesandega. [4]
Map	Kujutis Väärtuste hulk, kus väärtused on kindlas vastavuses teise hulga suuruste või väärtustega. [9]
Join point	Ühenduspunkt Punkt programmi käituses. Spring AOP'is on ühenduspunktiks alati meetodi täitmine (*execution*).[14]
Advice	Aspekti tegevus konkreetses ühenduspunktis. Eristatakse mitut erinevat tüüpi *advice*'i, näiteks *before*, mille puhul toimub tegevus alati enne ühenduspunkti, ja *after*, mille puhul toimub tegevus pärast ühenduspunkti.[14]
Pointcut	Predikaat, mille järgi tuvastatakse ühenduspunkte.[14]
POJO	**P**lain **O**ld **J**ava **O**bject Tavaline Java objekt, mis ei sisalda erifunktsionaalsust ega loogikat. [36]
Polling	*Polling* on tegevus, mille raames näiteks arvuti programm käib mingi intervalli tagant kontrollimas mingit staatust. [24]

Sisukord

1 Sissejuhatus 10
1.1 Taust ja probleem 10
1.2 Ülesande püstitus 11
1.3 Ülevaade tööst 12
2 Sündmused 13
2.1 Sündmuste tüübid 14
2.2 Sündmuste staatused 15
3 Funktsionaalsus andmebaasides 17
3.1 Andmebaasitrigerid 17
3.2 Andmebaasimuutujad 17
3.3 Tagasiviited algandmebaasidele sihtandmebaasis 19
3.4 Sündmuste tabel 19
4 Sünkroniseerimisrakenduse arhitektuur 21
4.1 Perioodilise töö kiht 22
4.2 *Polling* kiht 23
4.3 Sündmuste töötluskiht 23
4.4 Äriloogika kiht 25
4.5 Andmebaasikiht 26
4.6 Domeeniobjektid 27
4.7 Veahaldus 27
4.8 Paketistruktuur 29
4.9 Logimine 29
5 Testimine 31
5.1 Ühik-testid 31
5.2 Integratsioonitestid 31
5.3 Manuaaltestimine 32
5.4 Kontrollpäringud 33
6 Olemasolevate lahenduste ülevaade 34
6.1 DBSync 34

6.2 Cross-Database Studio 34
6.3 Datanamic SchemaDiff MultiDB 34
6.4 Võrdlus 35
7 Kokkuvõte 37
Viited 38
Lisa 1 – Oracle andmebaasitriger sünkroniseerimissündmuse tekitamiseks 40
Lisa 2 - Postgres andmebaasitriger sünkroniseerimissündmuse tekitamiseks 41
Lisa 3 – Sünkroniseerimise näidis-seadistus 41
Lisa 4 - Sündmuste tabeli loomislause Oracle andmebaasisüsteemis 42
Lisa 5 - Sündmuste tabeli loomislause Postgres andmebaasisüsteemis 42
Lisa 6 - Näidis integratsioonitest 43
Lisa 7 - *Switch-case* kasutusnäide sünkroniseerimisrakenduses 44
Lisa 8 – Oracle sessioonimuutuja kasutusnäide 45
Lisa 9 – Postgres sessioonimuutuja kasutusnäide 45
Lisa 10 – *Map* tüüpi tabel andmebaasidevaheliste viidete hoidmiseks 45
Lisa 11 – Andmebaasiindeksi loomislause 45
Lisa 12 – Rakenduse kihid 46
Lisa 13 – *Job*-i implementatsioon 47
Lisa 14 – *Spring scheduling* kolme lõime konfiguratsioon 47
Lisa 15 – *PollingService*-i näidisimplementatsioon 48
Lisa 16 – Näidis aspekti implementatsioon 48
Lisa 17 – *JdbcTemplate* ja *NamedParameterJdbcTemplate* kasutusnäide 49
Lisa 18 – *BeanPropertyRowMapper* kasutusnäide 49
Lisa 19 – Rakenduses kasutatava erindi definitsiooni näide 49
Lisa 20 – *Package-by-layer* ja *package-by-feature* näited 50
Lisa 21 – Näidis *unit-test* 50
Lisa 22 – Jadadiagramm sündmuse töötlemisel vea tekkimisest 51

Jooniste loetelu

Joonis 1 - rakenduse kihid 21
Joonis 2 - sünkroniseerimise näidis-seadistus 41
Joonis 3 - *map* tüüpi tabel andmebaasidevaheliste viidete hoidmiseks 45
Joonis 4- erinevad paketistruktuurid 50
Joonis 5 - jadadiagramm sündmuse töötlemisel tekkivast veast 51

Tabelite loetelu

Tabel 1 - olemasolevate lahenduste võrdlus 35

1 Sissejuhatus

Antud lõputöö eesmärgiks on kirjeldada Java rakendust ning sellega seonduvat funktsionaalsust, millega on võimalik erinevate andmemudelitega relatsiooniliste andmebaasisüsteemide vahel andmeid sünkroniseerida. Analoogne rakendus on autori osalusel valminud ühes Eesti ettevõttes 2015. aastal ning seda kasutatakse antud töö alusena. Töö autor on osalenud rakenduse tehnoloogiate juurutamisel ning teostanud rakenduses nii arendustöid kui tehnilist analüüsi. Konkreetne rakendus on antud ettevõttes sünkroniseerinud erinevate andmebaaside vahel rohkem kui 1,5 miljonit andmete muudatust.

Käesolevas töös on konkreetse ettevõtte probleem üldistuse huvides abstraheeritud.

1.1 Taust ja probleem

Ühinevad kaks ettevõtet, A ja B. Mõlemal ettevõttel on oma ettevõttega seotud olemite hoidmiseks andmebaasisüsteemid. Konkreetsete ettevõtete näitel on tegemist suhteliselt vanade monoliitsete Oracle andmebaasisüsteemidega. Ettevõtete sisuliseks ühinemiseks on tarvis need andmebaasisüsteemid kokku koondada. Kuna tegemist on, nagu eelnevalt mainitud, vanade monoliitsete süsteemidega, siis ei soovita neist kummagi platvormiga edasi liikuda ehk valikus ei ole ainult ettevõtte A või ainult ettevõtte B andmebaasisüsteem.

Ettevõtte uue arhitektuuri kohaselt soovitakse:

- Lõhkuda monoliitsed süsteemid valdkonnapõhisteks mooduliteks (näiteks tootehaldus, arveldus, kliendihaldus jne).
- Kasutada andmebaase andmete hoidmiseks ning vältida nende sidumist konkreetse ajahetke äriloogikaga, mis on ajas muutuv.
- Piirata andmebaasi otsetarbijate arvu, et vältida olukorda, kus erinevad rakendused kasutavad andmete tõlgendamiseks erinevat äriloogikat.

Konkreetse ettevõtte kliendihaldus valdkonna näitel on uueks andmebaasi-platvormiks valitud Postgres. Andmebaasiga suhtlus peaks toimuma läbi selleks ettenähtud vahekihi, milleks on *REST-API*, mis sisaldab nii andmete presentatsiooniks kui muutmiseks vajalikke liidestuspunkte ning operatsioonidega seonduvat äriloogikat.

Esimese etapina uue andmebaasi kasutusele võtmisel on tarvis teha andmete migratsioon andmebaasidest A ja B uude andmebaasi, nimetagem seda andmebaasiks C. Kuna andmebaasides A ja B eksisteerib kasutajaliidestes väga palju funktsionaalsust, mida kogu ettevõtte töötajad ning paljud rakendused igapäevaselt kasutavad, siis peab üleminek andmebaasile C toimuma järk-järgult. See tähendab, et teatud perioodil on kõik kolm andmebaasi A, B ja C aktiivses kasutuses. Siit jõuamegi täpsemalt antud töös lahendatava probleemini. Kuna andmed muutuvad kõigis kolmes süsteemis ning peavad samal ajal olema kättesaadavad kõigis kolmes süsteemis, siis peab andmeid andmebaasides A, B ja C omavahel sünkroniseerima (vt seadistust Lisa 3).

Nagu eelnevalt mainitud, on varasemate andmebaasisüsteemide näol tegemist monoliitsete süsteemidega, kus on põimunud erinevad valdkonnad. Seetõttu on uude andmebaasisüsteemi C välja töötatud ka uus andmemudel, mis peegeldab ainult konkreetse valdkonna vajadusi.

1.2 Ülesande püstitus

Eesmärk on kirjeldada lahendust, mis võimaldaks kahe-suunaliselt sünkroniseerida andmete muudatusi erinevate andmemudelitega mitmete andmebaasisüsteemide vahel. Lahendus peaks toimima muudest protsessidest sõltumatult ning asünkroonselt peaaegu reaalajas (*near real-time - NRT*). Mõiste „peaaegu reaalajas“ on täpselt defineerimata, aga oluline on, et sünkroniseerimine toimuks pidevalt ning ei takistaks ärilisi protsesse [31] . Lahendus peaks olema realiseeritud selliselt, et sellest oleks võimalik üleminekuperioodi lõpus lihtsalt vabaneda. See tähendab, et lahendus peaks omama minimaalseid seoseid muude rakenduste ja protsessidega.

Kolme andmebaasisüsteemi näitel, kus kaks on nii-öelda vanad andmebaasid ning üks sihtandmebaas, millele soovitakse üle minna, peaks sünkroniseerimine toimuma sihtandmebaasist lähtuvalt. See tähendab, et vanade andmebaaside otsest omavahelist

sünkroniseerimist ei toimu. Nende sünkroonsus on tagatud läbi sihtandmebaasi kahesuunalise sünkroniseerimise (vt joonist Lisa 3).

1.3 Ülevaade tööst

Teises peatükis antakse ülevaade sünkroniseerimise tarkvaralahenduse huviobjektidest ehk sündmustest. Kolmandas peatükis on juttu lahenduse jaoks vajalikust andmebaasidesse loodavast funktsionaalsusest ning selle implementatsioonist. Neljandas peatükis kirjeldatakse detailselt Java rakendust, mis tegeleb sünkroniseerimisega, ja selle arhitektuuri. Viiendas peatükis antakse ülevaade sünkroniseerimise testimisest. Kuuendas peatükis võrreldakse erinevaid olemasolevaid lahendusi ning põhjendatakse nende sobivust või mittesobivust konkreetse probleemi lahendamiseks. Seitsmendas peatükis on käesoleva lõputöö kokkuvõte.

2 Sündmused

Sündmus on olem, mis on sünkroniseerimisrakenduse sisend- ja töötlusobjekt. Sündmusi hoitakse kõikides sünkroniseeritavates andmebaasides `sync_event` tabelis.

Sündmuse koostisosad (ehk `sync_event` tabeli veerud):

- `id` - sündmuse primaarvõti. Tekib *auto-incrementi* tulemusel.
- `action_type` - sündmuse tekitanud tegevuse tüüp. Täidab triger. Täpsemalt peatükis 2.1 „Sündmuste tüübid“.
- `table_name` - tabel, kus sündmus esines. Täidab triger.
- `table_column` - tabeli veerg, millel sündmus esines (näiteks `UPDATE` operatsioon, mis on SQL'is veeru põhine). Täidab triger.
- `old_value` - tabeli veerus varasemalt olnud väärtus. Veeru olemasolu säästab ühe baasipäringu, et uuendatud kirjet küsida. Täidab triger.
- `new_value` - tabeli veeru uus väärtus. Annab sama lisaväärtuse, mis `old_value` veerg. Täidab triger.
- `table_pk_value` – kirje primaarvõti, millel sündmus esines. Täidab triger.
- `event_ts` - sündmuse esinemise aeg. Täidab triger.
- `created_by` - sündmuse tekitanud andmebaasi kasutaja. Täidab triger.
- `status` - sündmuse staatus. Täpsemalt peatükis 2.2 „Sündmuste staatused”. Täidab triger, muudab sünkroniseerimisrakendus.
- `error_message` - sünkroniseerimisel tekkinud vea teade ja pinujada. Täidab sünkroniseerimisrakendus.
- `processed_ts` - sündmuse töötlemise aeg. Täidab sünkroniseerimisrakendus.
- `target` - andmebaas, kuhu antud sündmus on tarvis sünkroniseerida. Täidab triger.
- `business_transaction_name` - kokkuleppeline nimi ärilisele transaktsioonile. Täidab rakendus, mis ärilist transaktsiooni läbi viib.

2.1 Sündmuste tüübid

- INSERT (I) - seda tüüpi sündmus tekib, kui andmebaasi sisestakse kirje ehk käivitatakse SQL `insert` operatsioon. Tavaliselt sellist tüüpi sündmuse töötlemisel luuakse vastav kirje ka sünkroniseerimise sihtandmebaasi ning lisatakse sellele tagasiviide algandmebaasi primaarvõtme näol.
- UPDATE (U) - seda tüüpi sündmus tekib, kui andmebaasis uuendatakse kirjet ehk käivitatakse SQL `update` operatsioon. Tavaliselt sellist tüüpi sündmuse töötlemisel uuendatakse tagasiviite abil vastavat kirjet ka sünkroniseerimise sihtandmebaasis.
- DELETE (D) - seda tüüpi sündmus tekib, kui andmebaasis kustutakse kirje ehk käivitatakse SQL `delete` operatsioon. Tavaliselt sellist tüüpi sündmuse töötlemisel kustutatakse tagasiviite abil vastavad kirjed või tagasiviited ka sünkroniseerimise sihtandmebaasist.
- BUSINESS_TRANSACTION (B) – seda tüüpi sündmus ei teki otseselt andmebaasi muudatusest, vaid neid tekitab andmebaasiga suhtlev rakendus (näiteks *REST-API*). Kui eelnevalt kirjeldatud sündmused on ühe tabeli konkreetse kirje põhised, siis B-tüüpi sündmus võib endas kujutada mitmete tabelite kirjete komplekti sünkroniseerimist. Näiteks on andmebaasis A vaja mingi olemiga koos luua veel teisi olemeid (näiteks kliendil peab olema kontaktinfo). See tähendab, et andmebaasist C mingite olemite sünkroniseerimist ei saa teha tabelipõhiselt vaid sünkroniseerimise hetkel peab olema veendunud, et kõik andmebaasi A sünkroniseeritavad olemid on olemas. Selleks lülitab seda ärilist transaktsiooni läbiviiv rakendus andmebaasis C muudatuse tegemise hetkel sünkroniseerimistrigerid välja ning kirjutab kirjed kõikidesse vajalikesse tabelitesse (vt peatükk 3.2 “Andmebaasimuutujad”). Seejärel loob see rakendus sünkroniseerimissündmuste tabelisse B-tüüpi sündmuse, mille võtmeks määratakse ärilise transaktsiooni nii-öelda põhiobjekti (kokkuleppeline) primaarvõti ning täidetakse ka transaktsiooni nimi (`business_transaction_name` - näiteks create_customer, vt peatükk 2 “Sündmused”). Sellise lahenduse puhul võib kindel olla, et sünkroniseerimisel andmebaasist C andmebaasi A, on andmebaasis C olemas kogu vajalik informatsioon, mis tagab ka andmebaasi A ärilise terviklikkuse.

- REINSERT (R) - sellise tüübiga sündmus ei ole tekkinud otseselt andmebaasi muudatusest, vaid neid tekitab sünkroniseerimisrakendus ise. Näiteks, käivitatakse andmebaasis A SQL `insert` operatsioon, millest tuleneva sündmuse töötlemine ebaõnnestub ebakorrektsete andmete tõttu. Kui nüüd andmeid andmebaasis A parandada siis tekib vastav UPDATE sündmus, mille töötlemise käigus ei leia sünkroniseerimisrakendus andmebaasist C tagasiviidet uuendatud andmebaasi A olemile. Järelikult on tegemist andmeparandusega ning loogika lahushoidmiseks tekitab sünkroniseerimisrakendus R-tüüpi sündmuse, mida käsitletakse tavaliselt samamoodi nagu INSERT tüüpi sündmust.

2.2 Sündmuste staatused

- TODO - sündmus on veel töötlemata ning ootab sünkroniseerimisrakenduse poolt töötlemist. Selle staatuse määrab sündmusele triger.
- OK – rakendus on sündmuse edukalt töödelnud. Selle staatuse määrab sündmusele sünkroniseerimisrakendus.
- ERROR - sündmuse töötlemisel tekkis viga. Täpsem info veast (pinujada koos teatega) asub `error_message` veerus (vt. peatükk 2 “Sündmused”). Selle staatuse määrab sündmusele sünkroniseerimisrakendus.
- DISABLED - sündmuse töötlemine rakenduses on välja lülitatud. See on vajalik näiteks juhul, kui antud tabeli sündmuse töötlemise funktsionaalsuse arendus on veel pooleli ning konkreetsel ajahetkel seda funktsionaalsust ei soovita kasutada. Selle staatuse määrab sündmusele sünkroniseerimisrakendus.
- TERMINATED - sündmuse töötlemine on lõpetatud. Staatus on kasutusel juhul, kui konkreetne sündmus on loodud sünkroniseerimisrakenduse poolt ning seda ei soovita vastupidises suunas uuesti tagasi sünkroniseerida, et vältida tsüklit. Näiteks sisestatakse andmebaasi A mingi kirje x, seepeale tekib vastav sündmus, mille sünkroniseerimisrakendus töötleb ning tekitab andmebaasi C mingi kirje y. Kirje y sisestamise järel andmebaasi C, tekib sündmus, mis peaks sünkroniseeritama omakorda andmebaasi A. Nii võib tekkida lõputu tsükkel. Selle staatuse määrab sündmusele triger. Selle staatuse määramise kohta saab täpsemalt lugeda peatükist 3.2 “Andmebaasimuutujad”.

- MANUAL - sündmus vajab käsitsi töötlust. Kasutusel näiteks juhul, kui mingi sündmuse töötlemise automatiseerimine on liiga keeruline ning sündmus on harva esinev. Selle staatuse määrab sündmusele sünkroniseerimisrakendus.
- IGNORED - sündmuse töötlemine ei ole vajalik ehk sündmust ignoreeritakse. Sellise staatuse määrab sündmusele sünkroniseerimisrakendus. Sisuliselt saaks seda juhtida ka trigeris, kuid sellisel juhul ei tekiks ka sündmuse kirjet. Soov on mitte lisada sellist loogikat trigeritesse, kuna mingil hetkel võib tekkida vajadus varasemalt ignoreeritud sündmuse sünkroniseerimise järele ning siis oleks tarvis muuta nii trigereid kui sünkroniseerimisrakendust.

3 Funktsionaalsus andmebaasides

Sünkroniseerimisrakenduse tööks on tarvis luua teatav funktsionaalsus andmebaasidesse. Lähemalt nendest funktsionaalsustest on juttu järgnevates peatükkides.

3.1 Andmebaasitrigerid

Sündmuste jälgimiseks andmebaasis on vajalikud trigerid jälgitavatele tabelitele. Trigeri tööülesandeks on luua sündmused sündmuste tabelisse (`sync_event`). Tuleks hoiduda trigerisse äriloogika kirjutamisest, et pidada rakenduste komplektis kinni ülesannete lahususe disainimustrist (*SoC - Separation of Concern*). Sellest disainimustrist kinnipidamine aitab vältida erinevate rakenduste komplekti osade tihedat sidestatust (*tight coupling*), mis omakorda tagab parema hooldatavuse ja muudetavuse [4] Äriloogika peaks olema rangelt sünkroniseerimisrakenduses. Trigerites võib kokkuleppeliselt sisalduda lihtsat loogikat sünkroniseerimisrakenduse koormuse vähendamiseks. Näiteks kui mingit kirjet uuendatakse samade väärtustega, mis seal enne olid, siis on teada, et sünkroniseerimisrakendusel ei ole nende sündmustega tarvis midagi teha.

Triger peab sündmuse sisestamisel täitma kohustuslikud veerud. Näited Oracle ja Postgres lihtsustatud sünkroniseerimistrigeri loomislausest tabelile `entity`, millel on jälgitav veerg `attribute`, ning mille primaarvõtmeks on veerg `id` leiab Lisa 1 ja Lisa 2.

3.2 Andmebaasimuutujad

Sõltuvalt sünkroniseerimisrakenduse loogikast võib tekkida lõputu sünkroniseerimistsükli oht. Näiteks lisatakse andmebaasi A uus kirje, see sünkroniseeritakse andmebaasi C ehk kirje lisatakse andmebaasi C. Seepeale hakkab sünkroniseerimisrakendus seda kirjet taaskord sünkroniseerima andmebaasi A ja võibki tekkida lõputu tsükkel.

Selle probleemi lahenduseks tuleb luua andmebaasidesse mingil kujul sessioonimuutujad, mida triger jälgib. Erinevate muutujate arv ühes andmebaasis sõltub sellest, mitmel suunal sellest andmebaasist andmeid väljapoole sünkroniseeritakse. Näiteks, toimub andmebaasist A sünkroniseerimine ainult andmebaasi C. Sellisel juhul piisab andmebaasis A ühest muutujast `disable_synchronization`. Selle muutuja olemasolu peab jälgima sünkroniseerimistriger. Juhul kui muutuja on määratud, siis peaks triger tekitama sünkroniseerimissündmuse olekuga TERMINATED.

Muutuja määramisega tegeleb sünkroniseerimisrakendus. See tähendab, et kui lisatakse kirje andmebaasi C, siis selle sünkroniseerimisel andmebaasi A määrab rakendus sessioonimuutuja, millest lähtuvalt kõikide andmebaasi A muudatuse kohta loob triger TERMINATED olekuga sündmused (vt peatükk 4.3 “Sündmuste töötluskiht”).
Teise näitena toimub andmebaasist C sünkroniseerimine andmebaasidesse A ja B. Sel juhul on tarvis kahte eraldi muutujat `disable_synchronization_to_A` ja `disable_synchronization_to_B`. Kui nüüd näiteks lisatakse kirje andmebaasi A siis selle sünkroniseerimisel andmebaasi C määratakse sessioonimuutuja `disable_synchronization_to_A`, millest lähtuvalt tekitab triger sündmuse, mille TARGET on A, olekuga TERMINATED ning sündmuse, mille TARGET on B, olekuga TODO.

Oracle andmebaasi puhul sobib antud olukorras sessioonimuutujana kasutada `CLIENT_INFO` veergu virtuaalses `V$SESSION` tabelis. Selle väärtuse juhtimiseks on olemas protseduur `SET_CLIENT_INFO`, `DBMS_APPLICATION_INFO` paketis [5] .
Näide kuidas antud funktsionaalsust kasutada on leitav Lisa 8.

Postgres andmebaasi puhul võib sessioonimuutujana kasutada näiteks konfiguratsiooni parameetreid. Alates PostgreSQL versioonist 9.2 on võimalik mugavalt kasutada *custom options* funktsionaalsust, mis võimaldab käitusajal punktiga eraldatult seadeid defineerida [6] [7] . Eraldaja vasakul poolel on eesliide ning paremal parameetri nimi.
Näide antud funktsionaalsuse kasutusest on leitav Lisa 9.

3.3 Tagasiviited algandmebaasidele sihtandmebaasis

Selle töö aluseks oleva probleemi puhul on sihtandmebaasiks andmebaas C. Sellest lähtuvalt on antud kolme baasi sünkroniseerimise juures alati üheks osapooleks andmebaas C. (vt Lisa 3)

Sünkroniseerimise võimaldamiseks peaks hoidma sihtandmebaasi olemite juures viiteid algandmebaasidele. Näiteks, olgu andmebaasis C mingi olem nimega `entity`, mis on sinna loodud andmete migreerimisel andmebaasist A. Kui nüüd seda olemit andmebaasis C muuta (`update` operatsioon), siis peab sünkroniseerimisrakendusel olema mingi identifikaator, mille järgi oleks võimalik tuvastada, missugust olemit on tarvis uuendada andmebaasis A. Selle probleemi lahendamiseks tuleb hoida algandmebaaside vastavate tabelite primaarvõtmeid sihtandmebaasi olemi juures. Näiteks, eelnevalt kirjeldatud olukorras, peaks andmebaasi C tabelil `entity` olema veerud `a_entity_id` ja `b_entity_id`. See lahendus võimaldab sünkroniseerida mõlemas suunas kuna alati on võimalik leida andmebaasist C viide alg-olemile.

Kui sihtandmete (andmebaas C) ja algandmete (andmebaasid A ja B) vahel kehtib üks-mitmele seos, siis oleks lahenduseks hoida neid eraldi *map*-tüüpi tabelis. See olukord võib tekkida mitmetel põhjustel. Näiteks on andmebaasis A duplitseeritud kirjed, mida mingil põhjusel on algsüsteemis keeruline parandada ning neid soovitakse sihtandmebaasis C üheks koondada. Kuid siiski on tarvis muudatusi selle andmebaasi C loodava kirje juures sünkroniseerida tagasi kõikidesse vastavatesse algandmebaasi A kirjetesse. Mainitud *map*-tüüpi tabel omab andmebaasi C olemitabeliga üks-mitmele seost. Näide `a_entity_map`, mis omab veerge `a_entity_id` (viide tabelile `entity` andmebaasis A) ning `entity_id` (viide tabelile `entity` andmebaasis C), on leitav Lisa 10.

3.4 Sündmuste tabel

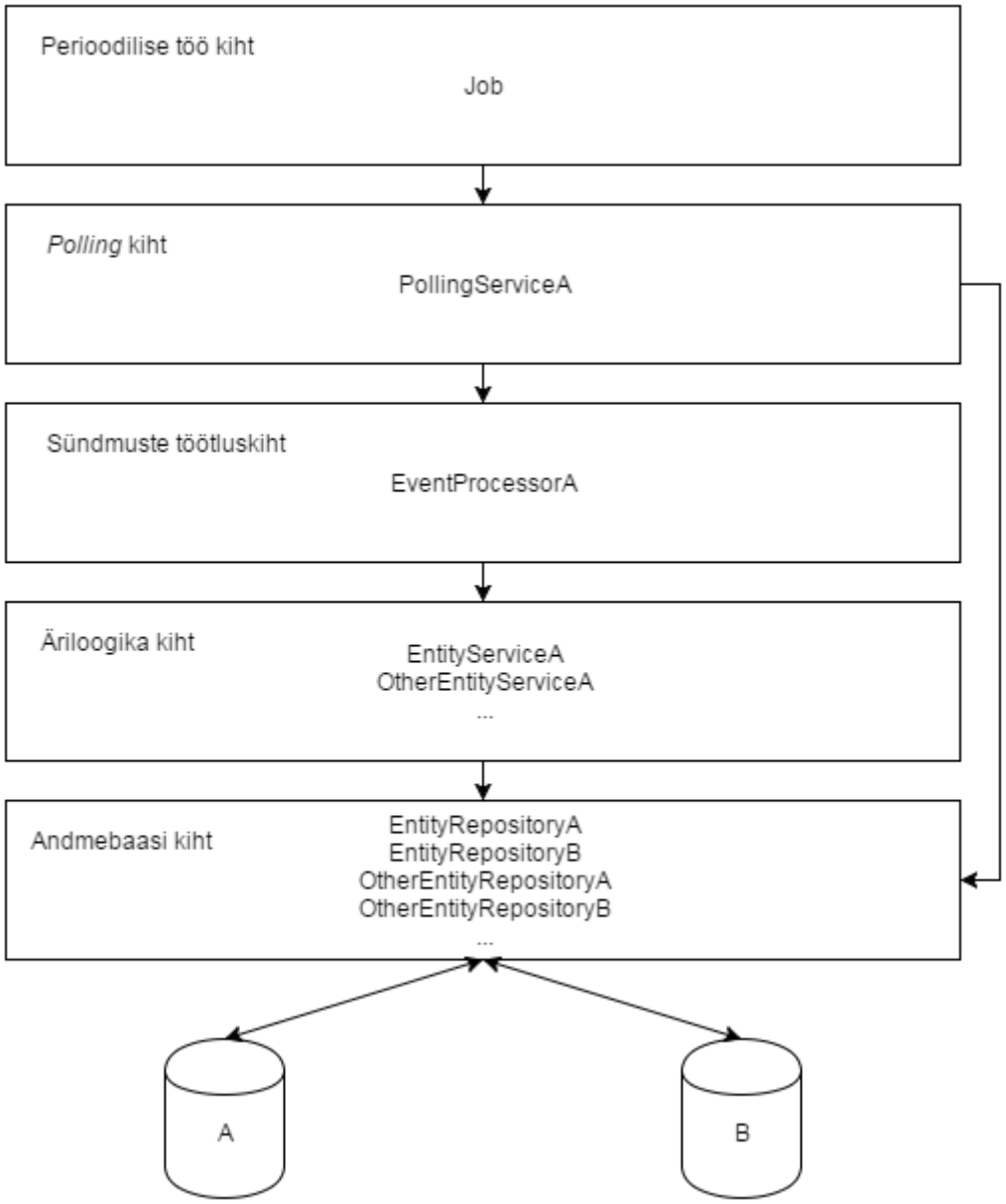
Muudatuste jälgimiseks andmebaasides on tarvis sündmuste tabelit, kuhu triger kirjutab kirjeid, mille töötlemisega tegeleb sünkroniseerimisrakendus. Sündmuste sisu on täpsemalt lahti kirjeldatud peatükis 2 “Sündmused”. Sündmuste tabeli loomislausete näited Oracle ja Postgres andmebaasides leiab vastavalt Lisa 4 ja Lisa 5 [10] [11] [12] .

Suurema koguse kirjete puhul on vajalik, et tabeli veergudel, mille järgi sünkroniseerimisrakendus sündmuseid pärib, oleksid indeksid. Vastasel juhul muutuvad päringud aina aeglasemaks. Kuna sünkroniseerimisrakendus kasutab sündmuste pärimise tingimuses `status` veergu, siis peaks sellele kohaldama indeksi. Lisaks järjestab sünkroniseerimisrakendus sündmused `event_ts` veeru järgi, millele peaks samuti indeksi kohaldama. Indeksi loomislause näide, mis sobib muutmata kujul nii Oracle kui Postgres andmebaasisüsteemidesse, on leitav Lisa 11.

4 Sünkroniseerimisrakenduse arhitektuur

Sünkroniseerimisrakenduse arhitektuuri võib nimetada n-kihiliseks. Kihte eristab üksteisest nende otstarve ning omavaheline suhtlus. Rakenduses suhtleb reeglina iga kihi klass otse ainult samas kihis asuvate või üks tase madalamas kihis asuvate klassidega.

Visuaalne ülevaade rakenduse kihtidest on näha järgneval joonisel.



Joonis 1 - rakenduse kihid

Nagu jooniselt näha on reeglile, et iga kiht suhtleb ainult endast üks tase madalama kihiga, üks erand, milleks on *polling* kiht. Selline erand on loodud kuna *polling* kiht ei tegele

mitte äriliste olemite vaid rakenduse huviobjektide ehk sündmustega. Seetõttu puudub seal vajadus läbida sündmuste töötluskihti või äriloogika kihti ning see võib suhelda sündmustega seonduvalt otse andmebaasi kihiga.

Täpsemalt kihtidest ja nende otstarbest järgnevates peatükkides.

4.1 Perioodilise töö kiht

Perioodilise töö kihis paikneb üks klass - `Job`. `Job` sisaldab endas kolme plaanilist tööd, mis käivitavad vastavalt andmebaasi A, B ja C `PollingService`-i. Plaanilisus on realiseeritud kasutades selleks Spring rakendusraamistiku pakutavat `@Scheduled` annotatsiooni [1] . Antud annotatsioon võimaldab perioodilisuse realiseerida kolmel erineval moel:

- *fixedDelay* - meetod käivitub iga n millisekundi järel. Aega mõõdetakse eelmise plaanilise töö lõpetamise hetkest.
- *fixedRate* - meetod käivitub iga n millisekundi järel. Aega mõõdetakse eelmise plaanilise töö käivitumise algusest.
- *cron* - meetod käivitub perioodiliselt vastavalt ette antud *cron*'i avaldisele, mis võimaldab töö käivitamist näiteks kindlal kella-ajal, kindlatel päevadel, kuudel jne. Cron on UNIX'i utiliit, mis võimaldab teha mingeid tegevusi kindlate ajavahemike tagant. [2] [29]

Millisekundite arvu n või *cron*'i avaldist ei pea sisestama avaldatud kujul annotatsiooni juurde vaid saab kasutada ka Spring-i pakutavat avaldiste keelt (*Expression Language - EL*), millega on võimalik nimeliselt viidata konfiguratsiooniparameetrile. See on kasulik selleks, et vajadusel saaks lihtsalt perioodilisust juhtida ilma, et rakendust oleks tarvis uuesti kompileerida [3] .

Konkreetse töö aluseks oleva probleemi lahenduse juures on oluline, et andmebaaside vaheline sünkroniseerimine toimuks *near real-time*. Sellest tingimusest lähtuvalt võib määrata ajaliseks viiteks töö käivitumiste vahel näiteks 1 sekundi. `Job`'i implementatsiooni näide on leitav Lisa 13.

Plaanilised tööd töötavad üksteisest sõltumatult eraldi lõimedes. See on vajalik seetõttu, et ühe suuna sünkroniseerimisel potentsiaalselt tekkiv probleem ei paneks seisma teistel suundadel sünkroniseerimist. Vaikekäitumisena kasutab Spring perioodiliste tööde täitmiseks ühte lõime. Selleks, et vaikekäitumist muuta, on tarvis rakenduses implementeerida Spring'i `SchedulingConfigurer` liides [8] .

Näidisimplementatsioon, kus määratakse Spring'i *scheduling* kasutama kolme lõime on leitav Lisa 14.

4.2 *Polling* kiht

Polling kihis paiknevad iga andmebaasiga seotud `PollingService` klassid. `PollingService` tegeleb sündmuste pärimisega andmebaasist ning edasisuunamisega sündmuste töötluskihti (`EventProcessor`). Lisaks sisaldab `PollingService` ka veahaldust ning vastutab sündmuste olekute muutmise eest andmebaasis.

Näidisprobleemi lahendamiseks on vaja sünkroniseerida kolme andmebaasi ehk sellest lähtuvalt eksisteerib kolm erinevat `PollingService`'it.

Ühe *polling* tsükli jooksul võetakse töötlusesse kuni 1000 kõige vanemat töötlemata sündmust. `PollingService` loetleb edukalt töödeldud ning vigaselt töödeldud andmebaasisündmusi ning logib selle info, mis on vajalik monitooringuks.

`PollingService`-i näidisimplementatsioon on leitav Lisa 15.

4.3 Sündmuste töötluskiht

Sündmuste töötluskihis paiknevad iga andmebaasiga seotud `EventProcessor` klassid. `EventProcessor`'i ülesandeks on `PollingService`'ist sisendina saadavad sündmused suunata vastavalt sündmuse sisule edasi sobivasse äriloogika kihi klassi.

`EventProcessor`'ist algab sünkroniseerimisrakenduse transaktsioon. Transaktsioonihalduseks on rakenduses kasutusel Spring. Vaikeseadistustega

kasutuselevõtmiseks on see väga lihtne - tuleb annoteerida klass `@Transactional` annotatsiooniga ning Spring'i konfiguratsiooniklassile lisada annotatsioon `@EnableTransactionManagement`. [12]

Transaktsioon algab `EventProcessor`'ist, kuna üks kiht kõrgemal `PollingService`'is on pärast sündmuse töötlemist tarvis kirjutada sündmuste tabelisse näiteks, sündmuse staatus ja veateade, mida ei soovita vea korral koos ülejäänud transaktsiooniga tagasi keerata (*rollback*). Üks kiht madalamal, äriloogika kihis, asuvad `Service` klassid - kuna `Service` klasse on palju, siis sõltuvalt erinevatest andmebaasidest ja nendega seotud olemitest on ebamõistlik neid kõiki transaktsioonilisteks seadistada, mis tooks kaasa keerukama hallatavuse. Lisaks võib olla teatud `Service`'ites vajadus suhelda teiste `Service`'itega, mis võib teatud olukordades transaktsioonihaldusele probleeme tekitada.

`EventProcessor`'i ümber on lisaks transaktsioonile ka trigerite juhtimiseks vajalik funktsionaalsus. Eesmärk on alati transaktsiooni alguses lülitada sihtandmebaasis trigerid välja ning pärast transaktsiooni need uuesti sisse lülitada (vt täpsemalt 3.2 "Andmebaasimuutujad"). Selle eesmärgi saavutamiseks on kasutatud aspekt-orienteeritud programmeerimist (*AOP*) koostöös Spring raamistikuga. *AOP* eesmärk on moduleerida funktsionaalsusi, mis läbivad mitmeid klasse ehk on nii-öelda ühisosad. Näiteks antud juhul oleks tarvis igas `EventProcessor` klassis enne sündmuse `Service` klassi saatmist kutsuda välja `Repository` meetod sessioonimuutuja määramiseks ning pärast `Service` klassist väljumist teha teine pöördumine `Repository` klassi, et sessioonimuutuja eemaldada. *AOP* võimaldab selle tegevuse jaoks defineerida aspekti.

Lähtudes AOP mõistetest, on eelpool kirjeldatud eesmärgi täitmiseks vaja defineerida aspekt, mille ühenduspunktiks oleks `EventProcessor`. *Advice*'e, peaks olema mitut erinevat tüüpi - *before* tüüpi *advice* sessioonimuutuja määramiseks ning *after* tüüpi *advice* sessioonimuutuja eemaldamiseks. Erinevate andmebaaside jaoks peaks olema defineeritud eraldi ühenduspunktid, *pointcut*'id ja *advice*'id. Koodinäide sobivast aspektist on leitav Lisa 16.

`EventProcessor`'ist sündmuste edasi suunamine äriloogika kihti on juhitud kolme tüüpi *switch* lausungitega (*switch statement*). Esimeses *switch* lausungis vaadatakse

sündmuse tüüpi, teises tabeli nime ning kolmandas tabeli veergu, kui tegemist on veerupõhise operatsiooniga. *Switch* lausungi vaikekäitumine on erindi viskamine, et saata sündmus manuaaltöötlusesse (vt peatükk 4.7 “Veahaldus”).

Näite sellisest *switch-case* konstruktsioonist andmebaasi A, olemi `entity`, millel on atribuudid `attributeOne` ja `attributeTwo`, alusel, leiab Lisa 7. [30]

4.4 Äriloogika kiht

Äriloogika kihis asuvad andmebaaside olemitega seotud `Service` klassid. Äriloogika kihis paikneb sünkroniseerimisrakenduse äriloogika. Äriloogikana käsitletakse antud rakenduse kontekstis reegleid, mille alusel tehakse andmete teisendust ühest andmebaasimudelist teise.

Äriloogika kihi meetodeid kutsub välja `EventProcessor`. `EventProcessor` annab kihile kaasa töödeldava sündmuse tekitanud andmebaasikirje primaarvõtme väärtuse, mille alusel päritakse andmebaasikihist juba konkreetne algandmebaasi objekt. Selle objekti omaduste ja vajadusel ka seotud objektide omaduste alusel tehakse otsused, missuguseid kirjeid on tarvis luua sihtandmebaasi. Vajadusel võib `EventProcessor`’ist `Service`’isse kaasa anda ka muid sündmuse atribuute, näiteks UPDATE sündmuse puhul uus veerus asetsev väärtus.

Äriloogika kihi klassidel on lubatud suhelda ka omavahel. Olgu näiteks kaks seotud olemit - `entity` ja `other_entity`. Kui nüüd `entity`’st tuleneva sündmuse töötlemisel on tarvis teha mingeid operatsioone `other_entity` olemiga, siis on kohane vastavate olemitega tegelemine kapseldada vastavatesse `Service` klassidesse, näiteks `EntityService` ja `OtherEntityService`. Sellise näite puhul on sobilik , et `EntityService` kutsub välja meetodeid `OtherEntityService`’ist ja vastupidi.

Andmebaasioperatsioonide tegemiseks kasutavad äriloogika kihi klassid andmebaasikihti.

4.5 Andmebaasikiht

Andmebaasikihis toimub sünkroniseerimisrakenduse suhtlus andmebaasidega. Otsesuhtlust andmebaasidega ei tohi toimuda mitte üheski teises rakenduse kihis. Andmebaasikihis ei tohiks sisalduda äriloogikat. Andmebaasikiht peab lihtsalt vahendama olemeid rakenduse äriloogika kihi ning andmebaasi vahel.

Andmebaasi kihis paiknevad iga andmebaasi tabeliga seotud `Repository` klassid. `Repository` klassid on defineeritud vastavalt andmebaasitabelitele - ehk iga rakenduse tööks tarviliku andmebaasi X vajalik tabel `entity` omab vastavat `Repository` klassi, nimetusega `EntityRepositoryX`.

Andmebaasiühenduse ning lausungite (*statement*) halduseks on rakenduses kasutusel Spring rakendusraamistiku pakutav `JdbcTemplate`, mis lihtsustab andmebaasidega suhtlust. `JdbcTemplate`'i kasutades peab rakenduse arendaja defineerima konfiguratsioonis vajalikud andmebaasiühendused ning vastavates meetodites kirjeldama SQL'i, päringusse kaasa antavad parameetrid ning tagastatavate andmete kaardistamise Java objektideks. Kogu ülejäänud vajaliku loogika eest andmebaasiga suhtluseks hoolitseb `JdbcTemplate`.

Parameetrite päringusse kaasa andmise lihtsustamiseks on kasutusel `NamedParameterJdbcTemplate`, mis võimaldab SQL päringus defineerida parameetrid nimeliselt ning oskab neid täita samanimelise Java objekti väärtusega. Kui päringul on palju parameetreid, mis kõik pärinevad ühest olemist, siis saab koostöös `NamedParamterJdbcTemplate`'iga kasutada `BeanPropertySqlParameterSource`'i, mis oskab parameetreid täita Java objekti alusel.

Illustreeriv koodinäide `JdbcTemplate`-i kasutamisest võrdluses `NamedParameterJdbcTemplate` ja `BeanPropertySqlParameterSource`'iga on leitav Lisa 17.

Andmebaasist tagastuvate andmete Java objektideks teisendamise lihtsustamiseks on kasutusel `BeanPropertyRowMapper`, mis sarnaselt

`BeanPropertySqlParameterSource`'ile oskab andmebaasi olemid automaatselt teisendada Java objektideks. [15]

Koodinäide `BeanPropertyRowMapper`'i kasutusest on leitav Lisa 18.

Kõik andmebaasikihi klassid peavad olema annoteeritud Spring'i pakutava `@Repository` annotatsiooniga. Selle annotatsiooni kasutamine tagab klassis Spring'i poolse veatöötluse, mille käigus teisendatakse laialt levinud andmebaasidest pärinevad vead Spring'i defineeritud vigadest, mis on teatud olukordades selgemad ning vajadusel lihtsamini eristatavad. [16] [17]

4.6 Domeeniobjektid

Sünkroniseerimisrakenduse domeenimudel peegeldab otseselt sünkroniseeritavate andmebaaside tabeleid.

Domeeniobjektid ei tohiks üldjuhul sisaldada äriloogikat. Domeeniobjektid on *POJO*'d (*Plain Old Java Objects*).

Domeeniobjektide koodi vähendamiseks ja lihtsustamiseks on rakenduses kasutusel Lombok'i nimeline teek, mis võimaldab annotatsioonidega määrata objektidele erinevat tüüpi utiliitmeetodeid. Näiteks annotatsioon `@Getter` genereerib kompileerimisel objekti atribuutidele `getter` meetodid, `@Setter` genereerib *setter* meetodid jne. Kõige tavapärasemalt leiab kasutust `@Data` annotatsioon, mis genereerib objektile `getter` meetodid, `setter` meetodid, kohustuslike argumentidega konstruktormeetodi, `toString()` meetodi ning `equals()` meetodi [28] . See on kasulik kuna suuremate objektide puhul saab sellega kokku hoida sadu ridu koodi, saades tulemusena loetavam kood.

4.7 Veahaldus

Sünkroniseerimisrakenduse käitusajal tekkivaid erindeid hallatakse *polling* kihis, kuna *polling* kiht on kõige madalam rakenduse kiht, mis asub väljaspool transaktsiooni. See on oluline, sest käitus-erindi (`RuntimeException`) korral on tarvis transaktsiooni käigus

andmebaasides tehtud muudatused tagasi võtta (*rollback*). Spring'i transaktsiooni vaikeseadistusena tehakse transaktsioonile *rollback*, kui esineb käitus-erind (`RuntimeException` - kaasaarvatud selle alamklassid) või muu kontrollimatu erind (*unchecked exception*) [13] . Kontrollimatuks erindiks loetakse kõik erindid, mida ei pea kinni püüdma või deklareerima meetodi signatuuris. Nimetus kontrollimatu tulenebki sellest, et sellist tüüpi erindit ei kontrollita kompileerimise ajal. [18]

Polling kihis püütakse kinni kõik madalamal tasemel esinevad käitus-erindid ning vastavalt erindi tüübile ja sisule määratakse töödeldava sündmuse staatus ja veateade. Sünkroniseerimisrakenduses on kasutusel mitmeid ise defineeritud erindeid erinevate äriloogiliste situatsioonide tarbeks. Kõik ise defineeritud erindid laiendavad `RuntimeException`'it.

Polling kihis püütavad erindid (järjestatud vastavalt püüdmise järjekorrale):

- `CannotGetJdbcConnectionException` - Spring rakendusraamistiku visatav viga, mis visatakse juhul, kui ei õnnestu mingil põhjusel *JDBC* vahendusel andmebaasiga ühendust saada [19] . Sellist tüüpi vea puhul ei muudeta töödeldava sündmuse staatust, kuna *JDBC* ühenduse taastumise korral on võimalik sündmus uuesti töötlusele võtta.
- `IgnoredEventException` - ise defineeritud erind, mida kasutatakse juhtudel, kui mingit tüüpi sündmusega ei ole vaja midagi teha ehk seda ignoreeritakse. Sellise vea püüdmisel märgitakse sündmusele staatus IGNORED (vt peatükk 2.2 "Sündmuste staatused").
- `MustBeUpdatedManuallyException` - ise defineeritud erind, mida kasutatakse juhtudel, kui mingil põhjusel konkreetse sisuga sündmuse sünkroniseerimist ei ole automatiseeritud ning see tuleb sünkroniseerida manuaalselt. Sellise vea püüdmisel märgitakse sündmusele staatus MANUAL.
- `DisabledEventException` - ise defineeritud erind, mida kasutatakse juhul, kui mingit tüüpi sündmuse töötlemine on konfiguratsiooniparameetriga välja lülitatud. Sellise vea püüdmisel märgitakse sündmusele staatus DISABLED.
- `Exception` - kõige üldisem erind Java keeles. Sellise vea püüdmisel märgitakse sündmusele staatus ERROR.

Sünkroniseerimisrakenduses on veel üks ise defineeritud erind, mida *polling* kihis eraldi ei käsitleta, kuna sellega kaasnevad tegevused ei erine üldisema `Exception` tüüpi erindi käsitlemisest.

- `SyncException` - sellist tüüpi erindit kasutatakse äriloogiliste vigade puhul. Näiteks, kui sünkroniseeritava sündmuse tekitanud kirjel on puudu mingid äriloogiliselt vajalikud eeldusseosed.

Näide rakenduses kasutatavast ise defineeritud erindist on leitav Lisa 19. Jadadiagramm veaolukorrast sündmuse töötlemisel on leitav Lisa 22.

4.8 Paketistruktuur

Kuna konkreetne sünkroniseerimisrakendus, millel antud töö põhineb, tegeleb ühe konkreetse valdkonnaga, siis on antud juhul otstarbekas jagada klassid pakettidesse kihi järgi (*package by layer*), sest rakenduse erinevad klassid on omavahel tihedalt seotud. Näiteks mingi olemi `entity` sünkroniseerimisel andmebaasist A andmebaasi C on tarvis luua andmebaasis C mitmeid olemeid ning kasutatakse andmebaasi A äriloogika kihis erinevaid andmebaasi C äriloogika kihi klasse.

Juhul, kui sünkroniseerimisrakendus peaks tegelema erinevate valdkondadega, millel puuduvad sünkroniseerimise vaates omavahelised seosed, tasuks kaaluda paketistruktuuri loomisel valdkonnast lähtuvat lähenemist (*package by feature*). Nii hoitakse valdkonnad ka kooditasemel lahus, mis tagab lihtsama navigatsiooni koodis ning parema modulaarsuse. Valdkondadest lähtuva paketistruktuuri puhul võib siiski madalamal tasemel kasutada ka kihi põhist pakettideks jaotamist, näiteks, kui konkreetse valdkonna andmete sünkroniseerimiseks on kasutusel palju klasse [20] [21] . Ilmestavad näited kahest mainitud paketistruktuuris on leitavas Lisa 20.

4.9 Logimine

Logimise jaoks sünkroniseerimisrakenduses palju nõudeid ei eksisteeri. Kõige olulisem on, et logitaks programmi töös tekkivad erindid. Erindite logimine on realiseeritud *polling* kihi klassides erindite püüdmisel (vt lisa peatükist 4.2 "*Polling* kiht" ning peatükist 4.7

“Veahaldus”). Kogu ülejäänud logimine on kokkuleppeline. Näiteks võib logida igas sünkroniseerimistsüklis töödeldud kirjete arvu.

Logimiseks on rakenduses kasutusel *slf4j-simple* logimisplatvorm koostöös *slf4j-api* fassaadiga, mis tagab logimisplatvormi lihtsa kasutamise ning lihtsustab vajadusel logimisplatvormi vahetamist. [27]

Selleks, et igas klassis, kus soovitakse logimist kasutada, ei peaks eraldi `Logger`’it defineerima, peaks annoteerima klassi `@Slf4j` annotatsiooniga. Selle annotatsiooni funktsionaalsuse annab rakendusele Lombok raamistik. [28]

5 Testimine

Sünkroniseerimise puhul on väga oluline selle testimine, kuna vigaselt sünkroniseerimine võib tekitada palju probleeme. Järgnevates peatükkides on lähemalt juttu sünkroniseerimise testimisvõimalustest.

5.1 Ühik-testid

Kogu sünkroniseerimisrakenduse äriloogika peab olema kaetud ühik-testidega. Selle tagamiseks on rakenduse arendamisel eelistatud *test-driven development* ehk *TDD* praktika kasutamine. *TDD* tähendab, et mingi funktsionaalsuse arendusprotsessi ei alustata mitte selle funktsionaalsuse implementatsioonist, vaid eelnevalt sellele testide kirjutamisest. Pärast testide kirjutamist tuleb implementeerida funktsionaalsus vastavalt testidele. *TDD* praktikast kinni pidamine tagab parema kvaliteediga koodi ning aitab võimalikke defekte varakult tuvastada. Koodi kõrge testidega katvus teeb ka koodi muutmise ja refaktoreerimise lihtsamaks, sest hästi kirjutatud unit-teste saab kasutada ka madala taseme dokumentatsioonina. [22] [23]

Ühik-testide tarbeks on sünkroniseerimisrakenduses kasutusel JUnit testimisraamistik, mis pakub testimiseks vajalikku baasfunktsionaalsust. Spring rakendusraamistiku hallatavate sõltuvuste *mock*'imiseks on kasutusel Mockito raamistik. *Mock*'imine on vajalik selleks, et testida klasse üksteisest sõltumatult.[25] [26]

Näide ühik-testist, mis testib Lisa 15 esitatud koodinäidet on leitav Lisa 21. Testi eesmärk on kontrollida, et `PollingService` annaks sündmuse töötlemisel `EventProcessor`'isse kaasa õige sündmuse.

5.2 Integratsioonitestid

Sünkroniseerimisrakendusele peavad olema realiseeritud *E2E* (*end-to-end*) integratsioonitestid kõikide võimalike andmebaasiga suhtlevate programmi teede ulatuses.

Iga konkreetse integratsioonitesti raames luuakse kõikidesse vajalikesse andmebaasidesse vajalikud algandmed. Enne algandmete loomist lülitatakse sünkroniseerimine sellest andmebaasist välja (vt peatükk 3.2 “Andmebaasimuutujad”) ning pärast algandmete loomist uuesti sisse.

Seejärel tehakse andmebaasis testitava sündmuse tekitav tegevus, näiteks kui testitakse andmebaasi A, `entity` tabeli INSERT tüüpi sündmuse (vt peatükk 2.1 “Sündmuste tüübid”) sünkroniseerimist andmebaasi C, siis tuleb testis käivitada vastava andmebaasi vastava tabeli `insert` operatsioon. Seejärel tuleb käivitada konkreetse andmebaasiga seotud `PollingService`’i `poll()` meetod. Sellele järgnevalt peaks valideerima sünkroniseerimisrakenduse loodud kirjeid andmebaasis C vastavalt äriloogikale ning andmebaasis A asuva sündmuse staatust.

Sünkroniseerimisrakenduse integratsioonitestide tehtud muudatusi ei salvestata andmebaasidesse ning muudatustele võetakse tagasi. See on tagatud sellega, et testid on transaktsioonilised ning integratsioonitestide Spring’i transaktsiooni konfiguratsioon on vastavalt seadistatud [13] . Näidise integratsioonitestist leiab Lisa 6.

Integratsioonitestides on testandmete loomiseks kasutusel eraldi andmebaasikihi klassid, mille nimetusse on lisatud sõna Test, näiteks `EntityTestRepositoryA`. Testides andmebaasikihi klasside eraldamise põhjuseks on soov mitte siduda testides kasutatavaid andmebaasimeetodeid toodangu koodiga.

5.3 Manuaaltestimine

Märkimisväärset manuaaltestimist sünkroniseerimisrakendusele ei tehta, kuna koodil peab olema väga kõrge katvus ühik-testide ja *E2E* integratsioonitestidega. Sõltuvalt konkreetse sünkroniseerimisrakenduse loogika keerukusest võib manuaaltestimine osutuda ebaefektiivseks ja kulukaks.

5.4 Kontrollpäringud

Sünkroniseeritavad andmebaasid A, B ja C peavad olema kaetud kontrollpäringutega. Kontrollpäringud on SQL päringud, mis loendavad erinevaid andmevigu andmebaasides. Kontrollpäringutega kontrollitakse kahte tüüpi vigade eksisteerimist:

- Andmete terviklikkus ühe andmebaasi vaates (*data integrity*) - see tähendab, et andmed moodustavad täpse terviku. Selleks peavad andmebaasis olevad andmed vastama selle andmebaasi ärireeglitele ning omama vajalikke seoseid konkreetse andmebaasi sees.
- Andmete sünkroonsus sünkroniseeritavate andmebaaside vaates. Näiteks konkreetse töö näidisprobleemi puhul peavad vastavalt sünkroniseerimiskaardistusele andmebaasid A, B ja C omama samasid andmeid ehk olema sünkroonis.

Andmete terviklikkus on suures osas tagatud andmebaasi kitsendustega, aga teatud keerukamate reeglite puhul ei ole kitsenduste kasutamine mõistlik. Kontrollpäringuga andmete terviklikkust kontrollides on võimalik vigastele andmetele kiirelt jälile saada ning konkreetseid vigu tekitanud loogikasse parandused sisse viia.

Andmete sünkroonsuse kontrollimiseks on loodud andmebaasi C eraldi skeemid (*database schema*), kuhu on loodud tabelid andmebaasidest A ja B sünkroniseeritavate tabelite struktuuriga. Teatud aja tagant laetakse andmebaaside A ja B sünkroniseeritavatest tabelitest andmed andmebaasi C vastavasse skeemi. Pärast andmete laadimist on võimalik andmebaasis C teostada päringuid, mis kontrollivad ärireeglite alusel, kas andmed on andmebaaside A, B ja C vahel sünkroonsed.

6 Olemasolevate lahenduste ülevaade

6.1 DBSync

DBSync on firma DMSoft Technologies poolt pakutav tarkvara, mis võimaldab *near real-time* sünkroniseerida andmeid erinevate andmebaasisüsteemide vahel.
DBSync kasutab andmemuudatuste jälgimiseks andmebaasitrigereid. DBSync võimaldab ka kahesuunalist sünkroniseerimist ning tarkvaral on palju konfigureeritavaid seadeid. Näiteks on võimalik ise defineerida erinevate andmetüüpide vahelisi kaardistusi ning ka defineerida filtreerimismeetodeid konditsionaalseks sünkroniseerimiseks. DBSync toetab laialt levinuimad relatsioonilisi andmebaasisüsteeme - MSSQL, Oracle, PostgreSQL, MySQL, Firebird. [32]

6.2 Cross-Database Studio

Cross-Database Studio on ettevõtte DBBalance pakutav tarkvara, mis võimaldab sünkroniseerida andmeid erinevate andmebaasisüsteemide vahel.
Cross-Database Studio't ei ole võimalik kasutada *near real-time*, kuna tarkvara ei tegele ühe andmebaasi muudatuste jälgimise vaid kahe andmebaasi võrdlemisega. Sünkroniseerimist on võimalik automatiseerida planeeritud käivituste näol (*scheduling*). Sünkroniseerimisprotsess koosneb andmete võrdlemisest, võrdluse alusel SQL skriptide genereerimisest ning nende käivitamisest. Cross-Database Studio võimaldab läbi konfiguratsiooni ka paljude sünkroniseerimise aspektide juhtimise. Tarkvara toetab kõiki ODBC draiveriga (*Open Database Connectivity*) töötavaid andmebaase [33] . *ODBC* on andmebaaside ühenduvuse standard. ODBC toetab teiste seas näiteks andmebaasisüsteeme MSSQL, Oracle, PostgreSQL, MySQL jne. [34]

6.3 Datanamic SchemaDiff MultiDB

Datanamic SchemaDiff MultiDB on firma Datanamic Solutions BV pakutav tarkvara, mis võimaldab sünkroniseerida andmeid erinevate andmebaasisüsteemide vahel.

Datanamic SchemaDiff MultiDB toimib sarnaselt eelnevas alapeatükis kirjeldatud Cross-Database Studio tarkvarale, võimaldades andmebaase omavahel võrrelda ning selle alusel sünkroniseerimisskripte genereerida ja käivitada ning neid tegevusi plaaniliselt teha. Datanamic toetab samuti levinumaid relatsioonilisi andmebaasissüsteeme - MSSQL, Oracle, PostgreSQL, MySQL jne. [35]

6.4 Võrdlus

Punktid, mille alusel andmebaaside sünkroniseerimisega tegelevat tarkvara omavahel võrrelda tulenevad otseselt antud töö probleemidest ja ülesandepüstitusest.

1. Kahesuunaline sünkroniseerimisvõimekus
2. Automatiseeritavus
3. *Near real-time* sünkroniseerimisvõimekus
4. Andmetüübi teisendused
5. Andmemudeli teisendused
6. PostgreSQL ja Oracle andmebaasisüsteemide tugi

Tabelis on kasutusel kolm tingmärki:

- + - antud funktsionaalsus/võimekus on olemas
- - - antud funktsionaalsus/võimekus puudub
- / - antud funktsionaalsuse/võimekuse osas puudub informatsioon

Tabel 1 - olemasolevate lahenduste võrdlus

	1.	2.	3.	4.	5.	6.
DBSync	+	+	+	+	-	+
Cross-Database Studio	/	+	-	+	-	+
Datanamic SchemaDiff MultiDB	/	+	-	+	-	+
Antud töös kirjeldatav tarkvaralahendus	+	+	+	+	+	+

Nagu eelpool toodud tabelist võib näha, on antud töös käsitletava tarkvara vajadus peamiselt ajendatud asjaolust, et andmeid on tarvis sünkroniseerida absoluutselt erineva andmemudeli ja äriloogikaga andmebaaside vahel. Mitte ükski olemasolev lahendus ei paku funktsionaalsust, kus oleks võimalik sünkroniseerimise protsessi osaks seadistada

otsustusprotsessi, mille alusel ja kuidas luua sünkroniseeritavaid olemeid sünkroniseeritavasse andmebaasi.

7 Kokkuvõte

Lõputöö eesmärgiks oli kirjeldada tarkvaralahendust, mille abil oleks võimalik omavahel sünkroniseerida kahte või enamat andmebaasi. Lõputöös on kirjeldatud ning näidetega toetatud selle eesmärgi täitmiseks nii andmebaasidesse loodav funktsionaalsus kui Java rakendus, mis sünkroniseerimisega tegeleb. Lisaks on kirjeldatud ning töös käsitletava rakendusega võrreldud mitmeid olemasolevaid lahendusi ning selgitatud nende sobivust või mittesobivust eesmärgi täitmiseks.

Töös kirjeldatav tarkvaralahendus vastab ülesande püstituses kirjeldatud nõuetele.

- Rakendus võimaldab sünkroniseerida andmeid erinevate andmebaaside vahel kahesuunaliselt. Kahesuunalise sünkroniseerimise juures on jälgitud, et ei tekiks lõputut sündmuste tsüklit.
- Rakendus töötab ilma märkimisväärsete ajaliste viideteta ehk sünkroniseerimist võib nimetada antud kontekstis *near real-time*. Rakenduse perioodiliste töötlustsüklite ajalist viidet on võimalik ise kontrollida.
- Rakenduse koodis on võimalik kirjeldada keerukat loogikat andmemudeli teisendamiseks. Loogikat on võimalik kirjeldada ka selliselt, et sünkroniseerida omavahel samasuguste andmemudelitega andmebaase.
- Rakendusest ning sellega seonduvast loogikast andmebaasides on võimalik ülemineku perioodi lõpus suhteliselt lihtsalt vabaneda, kuna sünkroniseerimine omab väga vähe seoseid muude rakenduste ja protsessidega.

Töös loodud tarkvaralahendust on loomulikult võimalik ka täiendada. Praegu vajab analoogse rakenduse kasutuselevõtt uute andmebaaside puhul palju uut rakenduse koodi. Ühe edasise suunana oleks võimalik luua lahenduse juurde utiliite, mis näiteks andmemudelite ja muu sarnase info põhjal oskaks genereerida sünkroniseerimiseks vajaliku programmikoodi.
Töö raames valminud rakenduse kood on leitav https://gitlab.com/Reins/sync-app.

Viited

[1] Spring Framework Reference. Task Execution and Scheduling. [WWW] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html#scheduling-annotation-support-scheduled (18.04.2016)

[2] Wikipedia. Cron. [WWW] https://en.wikipedia.org/wiki/Cron#CRON_expression (18.04.2016)

[3] Paraschiv, E. The @Scheduled Annotation in Spring. [WWW] http://www.baeldung.com/spring-scheduled-tasks (18.04.2016)

[4] Deviq. Separation of Concerns. [WWW] http://deviq.com/separation-of-concerns/ (19.04.2016)

[5] Toad World. DBMS_APPLICATION_INFO.SET_CLIENT_INFO. [WWW] http://www.toadworld.com/platforms/oracle/w/wiki/2367.dbms-application-info-set-client-info (20.04.2016)

[6] PostgreSQL Documentation. Customized options. [WWW] http://www.postgresql.org/docs/9.4/static/runtime-config-custom.html (20.04.2016)

[7] PostgreSQL Documentation. Setting parameters. [WWW] http://www.postgresql.org/docs/9.4/static/config-setting.html (20.04.2016)

[8] Spring JavaDoc API. Interface SchedulingConfigurer. [WWW] http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/SchedulingConfigurer.html (18.04.2016)

[9] Eesti Keele Instituut. IT terministandardi sõnastik. [WWW] http://eki.ee/dict/its/ (22.05.2016)

[10] Oracle Documentation. CREATE TABLE statement. [WWW] http://docs.oracle.com/javadb/10.6.1.0/ref/rrefsqlj24513.html (21.04.2016)

[11] Oracle-base. Identity Columns in Oracle Database 12c Release 1. [WWW] https://oracle-base.com/articles/12c/identity-columns-in-oracle-12cr1 (20.04.2016)

[12] PostgreSQL Documentation. CREATE TABLE. [WWW] http://www.postgresql.org/docs/9.4/static/sql-createtable.html (20.04.2016)

[13] Spring Framework Reference. Transaction Management. [WWW] http://docs.spring.io/autorepo/docs/spring/4.2.x/spring-framework-reference/html/transaction.html (21.04.2016)

[14] Spring Framework Reference. Aspect Oriented Programming with Spring. [WWW] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html (21.04.2016)

[15] Spring Framework Reference. Data access with JDBC. [WWW] https://docs.spring.io/spring/docs/current/spring-framework-reference/html/jdbc.html (22.04.2016)

[16] Spring JavaDoc API. Annotation Type Repository. [WWW] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Repository.html (22.04.2016)

[17] Spring Framework Reference. DAO support. [WWW] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/dao.html (22.04.2016)

[18] Singh, C. Checked and unchecked exceptions in java with examples. [WWW] http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/ (25.04.2016)

[19] Spring JavaDoc API. Class CannotGetJdbcConnectionException. [WWW] http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/CannotGetJdbcConnectionException.html (25.04.2016)

[20] Collected Java Practices. Package by feature, not layer. [WWW] http://www.javapractices.com/topic/TopicAction.do?Id=205 (25.04.2016)

[21] Shaun Childers blog. [WWW] http://shaunchilders.com/node/15 (25.04.2016)

[22] Hill, S. The pros and cons of Test-Driven Development. [WWW] https://leantesting.com/resources/test-driven-development/ (26.04.2016)

[23] Agile Data. Introduction to Test Driven Development (TDD). [WWW] http://agiledata.org/essays/tdd.html (26.04.2016)

[24] Wikipedia. Polling (computer science). [WWW] https://en.wikipedia.org/wiki/Polling_(computer_science) (22.05.2016)

[25] Junit. [WWW] http://junit.org/junit4/ (26.04.2016)

[26] Mockito. [WWW] http://mockito.org/ (26.04.2016)

[27] SLF4J user manual. [WWW] http://www.slf4j.org/manual.html (27.04.2016)

[28] Project Lombok. Overview. [WWW] https://projectlombok.org/features/ (27.04.2016)

[29] Kaldaru, U. Cron ja crontab. [WWW] https://wiki.itcollege.ee/index.php/Cron_ja_crontab (28.04.2016)

[30] Oracle Java documentation. The switch Statement. [WWW] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html (28.04.2016)

[31] Wikipedia. Real-time computing. [WWW] https://en.wikipedia.org/wiki/Real-time_computing#Near_real-time (28.04.2016)

[32] DBConvert. Features. [WWW] https://dbconvert.com/features (29.04.2016)

[33] DBBalance. Cross-Database Studio features. [WWW] http://www.dbbalance.com/db_studio_features.htm (29.04.2016)

[34] DataAccess. ODBC. [WWW] http://www.dataaccess.com/whitepapers/odbc.htm (29.04.2016)

[35] Datanamic. Datanamic SchemaDiff MultiDB features. [WWW] http://datanamic.com/schemadiff/features.html (29.04.2016)

[36] Spring. Understanding POJOs. [WWW] https://spring.io/understanding/POJO (22.05.2016)

[37] Rouse, M. RESTful API. [WWW] http://searchcloudstorage.techtarget.com/definition/RESTful-API (22.05.2016)

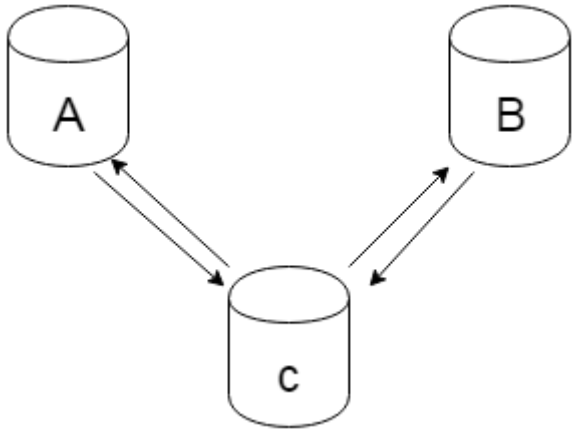
Lisa 1 – Oracle andmebaasitriger sünkroniseerimissündmuse tekitamiseks

```
CREATE OR REPLACE TRIGGER entity_sync
AFTER
INSERT OR DELETE OR UPDATE OF attribute
ON entity
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN

  IF INSERTING THEN
     INSERT INTO sync_event(event_ts, table_name, action_type,
table_pk_value, created_by, status)
       VALUES (SYSTIMESTAMP, 'ENTITY', 'I', :new.ID, USER, 'TODO');
     RETURN;
   END IF;

   IF DELETING THEN
     INSERT INTO sync_event(event_ts, table_name, action_type,
table_pk_value, created_by, status)
       VALUES (SYSTIMESTAMP, 'ENTITY', 'D', :old.ID, USER, 'TODO');
     RETURN;
   END IF;

   IF COALESCE( :new.ATTRIBUTE, '-') != COALESCE( :old.ATTRIBUTE, '-')
THEN
     INSERT INTO sync_event(event_ts, table_name, action_type,
table_column, old_value, new_value, table_pk_value, created_by,
status)
       VALUES (SYSTIMESTAMP, 'ENTITY', 'U', 'ATTRIBUTE',
:old.ATTRIBUTE, :new.ATTRIBUTE, COALESCE(:new.ID, :old.ID), USER,
'TODO');
   END IF;


END;/
```

Lisa 2 - Postgres andmebaasitriger sünkroniseerimissündmuse tekitamiseks

```
CREATE OR REPLACE TRIGGER entity_sync
 AFTER INSERT OR UPDATE OF attribute OR DELETE
 ON entity
 FOR EACH ROW
DECLARE
BEGIN
 IF (tg_op = 'INSERT') THEN
   INSERT INTO sync_event (event_ts, table_name, action_type,
table_column, new_value, old_value, table_pk_value, created_by,
status)
   VALUES (now(), 'ENTITY', 'I', null, null, null, new.id,
session_user, 'TODO');
 ELSIF (tg_op = 'DELETE') THEN
   INSERT INTO sync_event (event_ts, table_name, action_type,
table_column, new_value, old_value, table_pk_value, created_by,
status)
   VALUES (now(), 'ENTITY', 'D', null, null, null, old.id,
session_user, 'TODO');
 ELSE
   IF new.attribute IS DISTINCT FROM old.attribute THEN
     INSERT INTO sync_event (event_ts, table_name, action_type,
table_column, new_value, old_value, table_pk_value, created_by,
status)
     VALUES (now(), 'ENTITY', 'U', 'ATTRIBUTE', new.attribute,
old.attribute, new.id, session_user, 'TODO');
   END IF;
 END IF;
END;
```

Lisa 3 – Sünkroniseerimise näidis-seadistus



Joonis 2 - sünkroniseerimise näidis-seadistus

A ja B on varasemad andmebaasid. C on uus andmebaas.

Lisa 4 - Sündmuste tabeli loomislause Oracle andmebaasisüsteemis

```
CREATE TABLE sync_event
(
 id                          NUMBER GENERATED ALWAYS AS IDENTITY,
 event_ts                    TIMESTAMP(6) WITH TIME ZONE,
 action_type                 VARCHAR2(1 CHAR),
 table_name                  VARCHAR2(128 CHAR),
 table_column                VARCHAR2(128 CHAR),
 old_value                   VARCHAR2(2000 CHAR),
 new_value                   VARCHAR2(2000 CHAR),
 table_pk_value              NUMBER,
 created_by                  VARCHAR2(64 CHAR),
 status                      VARCHAR2(10 CHAR),
 error_message               VARCHAR(2000 CHAR),
 processed_ts                TIMESTAMP(6) WITH TIME ZONE,
 target                      VARCHAR2(10 CHAR),
 business_transaction_name VARCHAR2(128 CHAR)
);/
```

Lisa 5 - Sündmuste tabeli loomislause Postgres andmebaasisüsteemis

```
CREATE TABLE sync_event
 (
    id                          BIGSERIAL PRIMARY KEY,
    action_type                 CHARACTER VARYING(1) NOT NULL,
    table_name                  CHARACTER VARYING(128) NOT NULL,
    table_column                CHARACTER VARYING(128),
    old_value                   CHARACTER VARYING(2000),
    new_value                   CHARACTER VARYING(2000),
    table_pk_value              BIGINT NOT NULL,
    event_ts                    TIMESTAMP WITH time zone DEFAULT
now(),
    created_by                  CHARACTER VARYING(64) NOT NULL,
    status                      CHARACTER VARYING(10),
    error_message               CHARACTER VARYING(2000),
    processed_ts                TIMESTAMP WITH time zone,
    target                      CHARACTER VARYING(6),
    business_transaction_name CHARACTER VARYING(128)
 )
```

Lisa 6 - Näidis integratsioonitest

```
@ContextConfiguration("classpath:junitApplicationContext.xml")
@RunWith(SpringJUnit4ClassRunner.class)
@Transactional
@TransactionConfiguration(defaultRollback = true)
public class AEntityUpdateIntegrationTest {

	@Autowired
	private AEntityTestRepository aEntityTestRepository;
	@Autowired
	private ASyncEventTestRepository aSyncEventTestRepository;
	@Autowired
	private COtherEntityTestRepository cOtherEntityTestRepository;
	@Autowired
	private SyncJob syncJob;

	@Test
	public void updateEntityAttributeOneShouldUpdateOtherEntityAttributeTwoInC() {
		String ATTRIBUTE_ONE_VALUE = "ABC";
		String ATTRIBUTE_ONE_UPDATED_VALUE = "CBA"

		disableASyncTriggers();

		Long entityId = aEntityTestRepository.insertEntity(
					new Entity(ATTRIBUTE_ONE_VALUE));
		Long otherEntityId = cOtherEntityTestRepository
				.insertOtherEntityWithBackReference(
				    new OtherEntity(entityId));

		enableASyncTriggers();

		aEntityTestRepository.updateAttributeOneById(
					entityId, ATTRIBUTE_ONE_UPDATED_VALUE
		);

		syncJob.pollA();

		OtherEntityC otherEntityC = cOtherEntityTestRepository
					.findOtherEntityById(otherEntityId);
		SyncEvent syncEventA = aSyncEventTestRepository
					.getEventByTablePkValueAndType(entityId,
"U");

		assertEquals(ATTRIBUTE_ONE_UPDATED_VALUE,
otherEntityC.getAttributeTwo());
		assertEquals("OK", syncEventA.getStatus());
	}
}
```

Lisa 7 - *Switch-case* kasutusnäide sünkroniseerimisrakenduses

```
public void processEvent(SyncEvent event) {
   switch (event.getActionType()) {
     case I:
       processInsert(event);
       break;
     case U:
       processUpdate(event);
       break;
     case D:
       processDelete(event);
       break;
     default:
       throw new MustBeUpdatedManuallyException(event);
   }
 }

void processInsert(SyncEvent event) {
       switch (event.getTableName()) {
        case "ENTITY":
              entityServiceA.processEntityInsert(event.getTablePkValue());
              break;
        default:
              throw new MustBeUpdatedManuallyException(event);
       }
}

void processUpdate(SyncEvent event) {
       switch (event.getTableName()) {
        case "ATTRIBUTE_ONE":
              entityServiceA.processAttributeOneUpdate(event.getTablePkValue(),
event.getNewValue());
              break;
        case "ATTRIBUTE_TWO":
              entityServiceA.processAttributeTwoUpdate(event.getTablePkValue(),
event.getNewValue());
              break;
        default:
              throw new MustBeUpdatedManuallyException(event);
       }
}

void processDelete(SyncEvent event) {
       switch (event.getTableName()) {
        case "ENTITY":
              entityServiceA.processEntityDelete(event.getTablePkValue());
              break;
        default:
              throw new MustBeUpdatedManuallyException(event);
       }
```

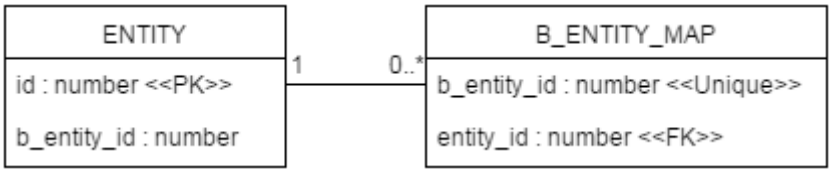
Lisa 8 – Oracle sessioonimuutuja kasutusnäide

```
CALL dbms_application_info.set_client_info('disable_synchronization');
CALL dbms_application_info.set_client_info('enable_synchronization');
```

Lisa 9 – Postgres sessioonimuutuja kasutusnäide

```
SET sync.disable_synchronization TO true;
RESET sync.disable_synchronization;
```

Lisa 10 – *Map* tüüpi tabel andmebaasidevaheliste viidete hoidmiseks

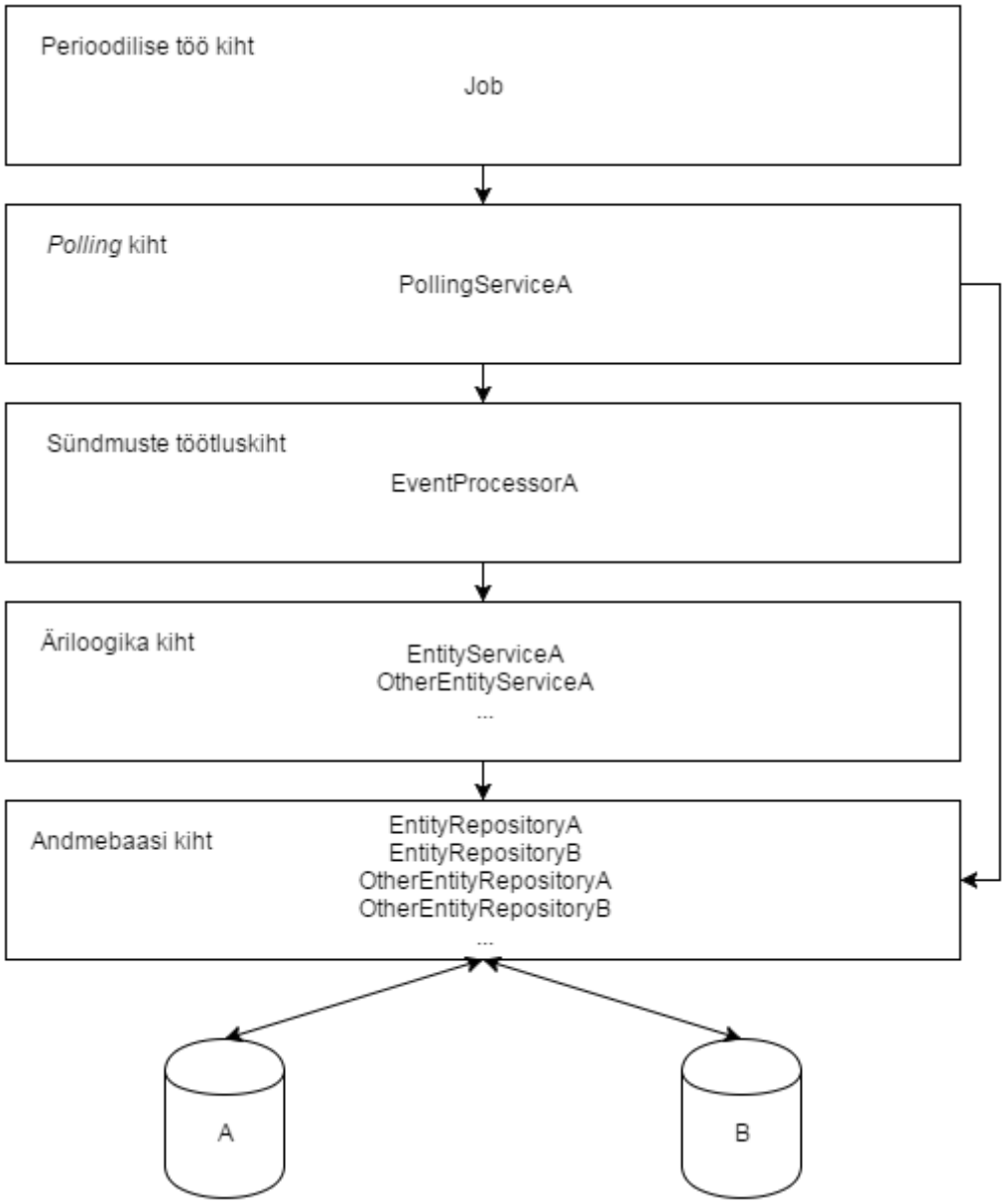


Joonis 3 - *map* tüüpi tabel andmebaasidevaheliste viidete hoidmiseks

Lisa 11 – Andmebaasiindeksi loomislause

```
CREATE INDEX sync_event_status_idx
ON sync_event(status);
```

Lisa 12 – Rakenduse kihid



Lisa 13 – *Job*-i implementatsioon

```
@Component
public class SyncJob {
   @Autowired
   private PollingServiceA pollingServiceA;
   @Autowired
   private PollingServiceB pollingServiceB;
   @Autowired
   private PollingServiceC pollingServiceC;

   @Scheduled(fixedDelayString = "${fixed-delay.poll.a}")
   private void pollA(){
       pollingServiceA.poll();
   }

   @Scheduled(fixedDelayString = "${fixed-delay.poll.b}")
   private void pollB(){
       pollingServiceB.poll();
   }

   @Scheduled(fixedDelayString = "${fixed-delay.poll.c}")
   private void pollC(){
       pollingServiceC.poll();
   }
}
```

Lisa 14 – *Spring scheduling* kolme lõime konfiguratsioon

```
@Configuration
public class SchedulingConfiguration implements SchedulingConfigurer {

   @Override
   public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
       ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
       taskScheduler.setPoolSize(3);
       taskScheduler.initialize();
       taskRegistrar.setTaskScheduler(taskScheduler);
   }
}
```

Lisa 15 – *PollingService*-i näidisimplementatsioon

```
@Service
@Slf4j
public class PollingServiceA implements PollingService {
   private int syncedEvents;
   private int errorEvents;

   @Autowired
   private SyncEventRepositoryA syncEventRepositoryA;
   @Autowired
   private EventProcessorA eventProcessorA;

   @Override
   public void poll() {
       syncedEvents = 0;
       errorEvents = 0;
       syncEventRepositoryA.getUnprocessedRows().forEach(this::processEvent);
       log.info("A -> C sync: Successfully synced " + syncedEvents + " event(s). " +
               "There were errors while trying to sync " + errorEvents + " event(s)."
       );
   }

   private void processEvent(SyncEvent syncEvent) {
       try{
           eventProcessorA.processEvent(syncEvent);
           syncEventRepositoryA.markChangeProcessed(syncEvent);
           syncedEvents++;
       } catch (Exception e){
           syncEventRepositoryA.markChangeProcessedWithStatus(syncEvent, e, ERROR);
           log.error("Error processing event " + syncEvent.getRowId(), e);
           errorEvents++;
       }
   }
}
```

Lisa 16 – Näidis aspekti implementatsioon

```
@Component
@Aspect
public class SyncAspect {
 @Autowired
 private SyncEventRepositoryA syncEventRepositoryA;

 @Pointcut("within(ee.ttu.sync.service.a.EventProcessorA)")
 public void aServices() {}

 @Before("aServices()")
 public void aServicesBefore(JoinPoint joinPoint) {
   syncEventRepositoryA.disableASynchronization();
 }

 @After("aServices()")
 public void aServicesAfter(JoinPoint joinPoint) {
       syncEventRepositoryA.enableASynchronization(); }}
```

Lisa 17 – *JdbcTemplate* ja *NamedParameterJdbcTemplate* kasutusnäide

```
   //plain JdbcTemplate
   public void insertEntity(Entity entity){
       String sql = "INSERT INTO entity(attribute_one, attribute_two,
attribute_three, attribute_four) VALUES (?, ?, ?, ?)";
       getJdbcTemplate().update(sql,
                                entity.getAttributeOne(), entity.getAttributeTwo(),
                                entity.getAttributeThree(),entity.getAttributeFour());
   }

   //NamedParameterJdbcTemplate and BeanPropertySqlParameterSource
   public void insertEntity(Entity entity){
       String sql = "INSERT INTO entity(attribute_one, attribute_two,
attribute_three,  attribute_four) VALUES (:attributeOne, :attributeTwo,
:attributeThree, :attributeFour)";
       getNamedParameterJdbcTemplate().update(sql,
                           new BeanPropertySqlParameterSource(entity));
   }
```

Lisa 18 – *BeanPropertyRowMapper* kasutusnäide

```
public List<Entity> findEntitiesByAttributeOne(String attributeOne){
       String sql = "SELECT attribute_one, attribute_two, attribute_three,
attribute_four FROM entity WHERE attribute_one = ?";
       return getJdbcTemplate().query(sql,
BeanPropertyRowMapper.newInstance(Entity.class), attributeOne);
   }
```

Lisa 19 – Rakenduses kasutatava erindi definitsiooni näide

```
public class IgnoredEventException extends RuntimeException {
 public IgnoredEventException(String message) {
   super(message);
 }
}
```

Lisa 20 – *Package-by-layer* ja *package-by-feature* näited

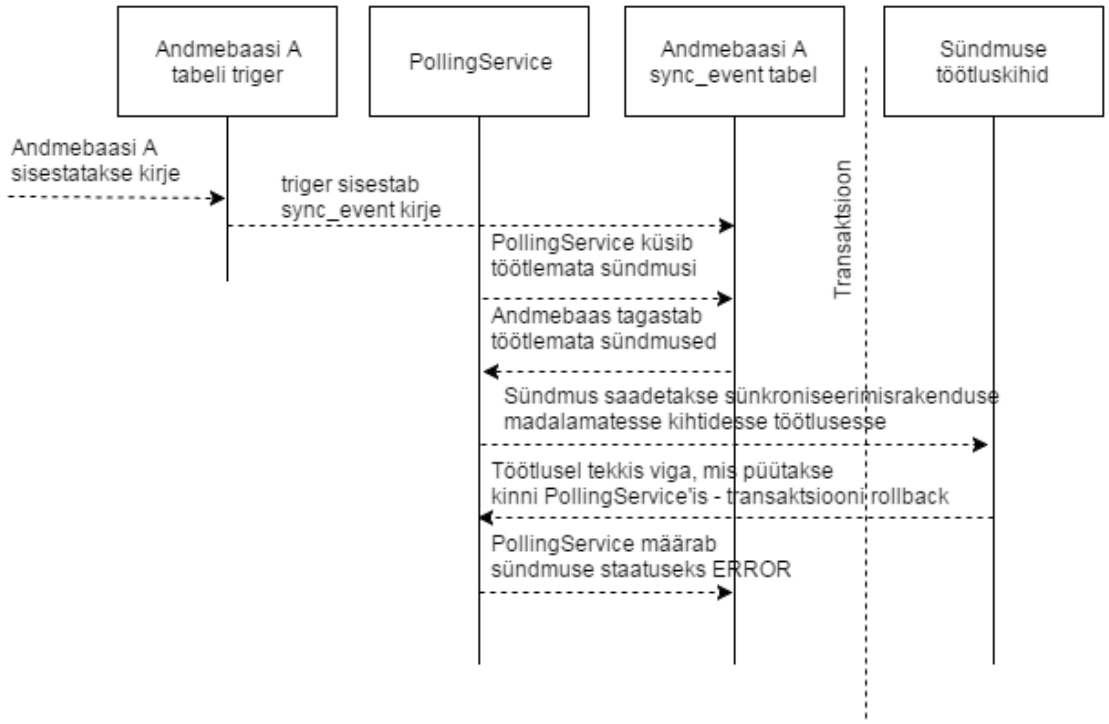
Joonis 4- erinevad paketistruktuurid

(vasakult: *package-by-layer*, *package-by-feature*)

Lisa 21 – Näidis *unit-test*

```
@RunWith(MockitoJUnitRunner.class)
public class PollingServiceATest {

 @Mock
 private EventProcessorA eventProcessorA;

 @InjectMocks
 PollingServiceA pollingServiceA;

 @Test
 public void pollEventShouldSendEventToProcessing() throws Exception {
   SyncEvent event = new SyncEvent();
   when(syncEventRepositoryA.getUnprocessedRows()).thenReturn(asList(event));

   pollingServiceAramis.poll();

   verify(eventProcessorA).processEvent(event);
   verify(syncEventRepositoryA).markChangeProcessed(event);
 }

}
```

Lisa 22 – Jadadiagramm sündmuse töötlemisel vea tekkimisest



Joonis 5 - jadadiagramm sündmuse töötlemisel tekkivast veast