TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Artjom Pahhomov 186042IADB

Taasloodava andmebaasi kasutuselevõtt Java pärandrakenduse arenduses ja automaattestimises AS-i LHV Pank näitel

Bakalaureusetöö

Juhendaja: Ago Luberg
MSc
Kaasjuhendaja: Taavi Tänavsuu
MA

Tallinn 2021

Autorideklaratsioon

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

Autor: Artjom Pahhomov

17.05.2021

Annotatsioon

Rakenduste arendus- ja testimisprotsesside parendamine on aktuaalne teema nii AS LHV Panga kui ka paljude teiste suurettevõtete jaoks. Üks võimalus protsesside kiirendamiseks ja parendamiseks on taasloodava andmebaasi kasutuselevõtt. Käesoleva töö eesmärk oli võtta kasutusele taasloodav andmebaas Java pärandrakenduses. Rakendus, mille näitel lahendus välja töötati, on 2012. aastast arenduses olnud ettevõttesisene pärandrakendus, mis omaette kaasab teatud lisapiiranguid ja nõudeid.

Käesoleva töö analüüsi osas oli omavahel võrreldud tehnoloogiaid, mida on võimalik kasutada eesmärgi saavutamiseks, ning valituks osutus Docker tehnoloogial põhinev lähenemine. Bakalaureusetöö praktilises osas tegeles autor lahenduse elluviimisega algul arenduses ning seejärel automaattestimises, lahendades jooksvalt tekkivaid probleeme ja takistusi. Töö tulemusena said esialgsed eesmärgid täielikult täidetud; saadud lahendus muutis teiste meeskonnaliikmete tööd mugavamaks ja lihtsamaks ning muutis automaattestimise protsessi palju stabiilsemaks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 38 leheküljel, 6 peatükki, 9 joonist, 2 tabelit.

Abstract

Introducing Ephemeral Database for Development and Automated Testing in Legacy Java Application on Example of AS LHV Pank

Improving the development and testing processes of applications is a significant topic not only for AS LHV Pank but for other large companies. One way to speed up and improve processes is to use an ephemeral database. The aim of this work was to introduce an ephemeral database in a Java legacy application. The application, on the example of which the solution was developed, is an in-house legacy application that has been in development since 2012, which generates certain additional restrictions and requirements to the solution.

In the analysis part of the thesis, various technologies that can be used to achieve the goal were compared, resulting in the Docker-based approach being chosen. In the practical part of the bachelor's thesis, the author dealt with the implementation of the solution first in development and then in automatic testing, solving appearing problems and obstacles in the process. As a result of the work, the initial goals were fully achieved; the resulting solution made the work of other team members more comfortable and easier, and made the automated testing process much more stable.

The thesis is in Estonian and contains 38 pages of text, 6 chapters, 9 figures, 2 tables.

Lühendite ja mõistete sõnastik

API	Rakendusliides, komplekt rakendustarkvara ehitamiseks (ingl *Application Program Interface*)
Beta-testing	Testimise tehnika, mille puhul kvaliteedikontrolli teostavad tavakasutajad
Bean	Java objekt, mille loob Spring Boot raamistik ning mida saab edaspidi sõltuvusena süstida.
CDS	LHV sisemine rakendus, mis talletab laekunud laenutaotlusi ning annab nendele otsuseid (ingl Credit Decision System)
CI	Pidev integratsioon (ingl *Continuous Integration*)
CLI	Käsurealiides (ingl *Command Line Interface*)
End-to-end testimine	Automaattestimise tehnika, mille puhul kontrollitakse rakenduse tervet teatud töövoogu
Hibernate ORM	Java raamistik objektide kaardistamiseks relatsioonilise andmebaasi jaoks
IDE	Tarkvararakendus, mis pakub programmeerijatele erinevaid tööriistu tarkvara arendamiseks (ingl *Integrated Development Environment*)
Jira	Tarkvara arendusprojektide planeerimiseks, juhtimiseks ja haldamiseks
JVM	Abstraktne masin, mis interpreteerib kompileeritud Java baitkoodi arvuti protsessorile arusaadavateks instruktsioonideks (ingl *Java Virtual Machine*)
singleton	Disainimuster tarkvaratehnikas, mille rakendamisel on teatud klassist võimalik luua vaid üks objekt
SQL	Andmebaaside päringukeel (ingl *Structured Query Language*)
Spring Boot	Java raamistikku Spring kuuluv moodul, mis on loodud võimaldamaks kiirelt ja lihtsalt Spring rakendust seadistada
Spring Data JPA	Java raamistikku Spring kuuluv moodul, mis lihtsustab tööd andmebaasidega ning andmetele juurdepääsu kihi (ehk *Data Access Layer*) implementeerimist
UI	Kasutajaliides (ingl *User Interface*)
VPN	Virtuaalne privaatvõrk (ingl *Virtual Private Network*)

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Sisemiste rakenduste arendus- ja testimisprotsesside parendamine on aktuaalne probleem nii AS LHV Pangale kui ka paljudele teistele suurettevõtetele ning töö andmebaasiga on väga oluline osa nii arendus- kui testimisprotsessist. Käesoleva töö probleemiks, mida autor lahendama asub, on ühiste ja püsivate andmebaaside kasutamine, mis toob kaasa erinevaid konflikte ja ebamugavusi arenduses ning ebastabiilsust automaattestimises.

Käesoleva töö eesmärk on võtta kasutusele taasloodav andmebaas, mis on üks võimalus ülalmainitud protsesside kiirendamiseks ja parendamiseks; tulemusena saadud lahendus peab olema arendaja ja kvaliteedispetsialisti jaoks piisavalt mugav ja lihtsasti hallatav, muudaks arendus- ja testimisprotsessid võimalikult kiiremaks, stabiilseks ja autonoomseks.

Bakalaureusetöö analüüsimise osas uuritakse, millised võimalused ja tehnoloogiad on kiiruse, turvalisuse ja mugavuse poolest kõige sobilikumad taasloodava andmebaasi loomiseks; selgitatakse välja nende eripärad, kaasnevad potentsiaalsed takistused ja probleemid lähtudes ettevõtte vajadustest, olemasolevatest ökosüsteemidest, turvaaspektidest jm sätestatud piirangutest. Selle osa tulemusena valitakse võrdleva analüüsi abil tehnoloogia, mida hakatakse praktilises osas kasutusele võtma.

Töö praktilises osas proovitakse analüütilise osa käigus valitud variant ellu viia arenduses ja automaattestimises ning lahendatakse ära töö käigus tekkinud probleemid, uuritakse võimalusi taasloodava andmebaasi test-andmetega täitmiseks; lõpplahendus peab olema mugav ja kasutuskõlblik. Vajadusel tehakse vastavad muudatused olemasolevasse arendusprotsessi (sh töö andmebaasiga, andmebaasimuudatuste arendamise konventsioon) ning testimisprotsessi (sh automaattestide täiendamine, parandamine). Käesolev osa nõuab nii olemasolevate tööriistade kui ka kasutusele võetavate tehnoloogiate tundmaõppimist.

2 Taust

Käesolevas töö osas tutvustatakse lugejale taasloodava andmebaasi kontseptsiooni ning tuuakse välja taasloodava ja püsiva andmebaasi eelised. Autor selgitab, mis väärtus on taasloodaval andmebaasil arenduses ja automaattestimises ning üldiselt milline roll on andmebaasidel nendes keskkondades. Samuti tutvustatakse lugejale ettevõtte ja projekti tausta, kus järgnev töö teostatakse.

2.1 Taasloodav andmebaas

Andmebaase kasutatakse rakendustes ja süsteemides andmete säilitamise eesmärgil. Siiski koosneb tarkvaraprojekti töövoog erinevatest osadest ning selle tähtsamad aspektid on kahtlemata arendus ja testimine. Ülalmainitud protsesside parendamiseks ning efektiivsuse tõstmiseks võetakse kasutusele vastavates projektides taasloodavaid andmebaase.

Taasloodavaks nimetatakse sellist andmebaasi, mida vajadusel luuakse taas uues keskkonnas kasutuskõlbliku seisuni. Selle mõiste alla kuuluvad ajutised ja manustatud andmebaasid: baasid, millel on lühike eluiga või mis reeglina eksisteerivad teatud aja jooksul (nt rakenduse jooksutamise ajal või teatud protsessi jooksul). Taasloodavaks võib nimetada ka lokaalset andmebaasi, mida tihti kasutatakse arendusprotsessis. Kõik need andmebaasid ei ole mõeldud pikaajaliseks andmete talletamiseks, vaid neid kasutatakse peamiselt rakenduse arendamiseks ja testimiseks ning vajadusel saab baasi ära kustutada ja tühjana taasluua. Inglisekeelsetes allikates on taasloodavate ja ajutiste baaside kirjeldamiseks enamasti kasutusel sõna „efemeerne“ (ingl *ephemeral*), mis tähendab „lühiealine“.

2.2 Taasloodava ja püsiva andmebaasi võrdlus

Taasloodaval andmebaasil on selged eelised püsiva andmebaasi ees. Kuna taasloodav andmebaas on igal arendajal individuaalne ja mängib eelkõige ajutise andmevaramu rolli, selle kasutamise peamised eelised on:

1. sõltumatus – arendajal on vabadus teha andmebaasiga kõiki katsetusi ning proovida erinevaid lahendusi: muudatused ei saa teistele andmebaasi kasutajatele probleeme tekitada; kui andmebaas ei ole katsetuste tagajärjel kasutuskõlblik, on arendajal võimalik kiiresti luua uus baas;
2. isoleeritus – iga protsessi jaoks on võimalik luua ajutine andmebaas koos vajalike andmetega, mis aitab kaasa protsesside isoleeritusele, nt automaattestimine on tänu sellele kindlam ja tõhusam;
3. kiirus – andmebaasiga ühendamiseks ei ole vaja internetiühendust, see töötab samas masinas; andmebaasi ajutisuse tõttu ei kogune sinna palju andmeid;
4. andmemaht – taasloodav andmebaas võtab vähem ruumi, kuna andmeid ei hoita pikaajaliselt; samuti säästetakse sellega ruumi püsivas andmebaasis, sest arendamise ja testimise käigus ei toimu test-andmetega üleliigset reostamist.

Püsiva andmebaasi kasutamisel arenduses ja automaattestimises on olemas ka eelised, neist peamised on:

1. turvalisus – andmelekke oht on madalam, sest andmeid ei hoita lokaalses masinas, vaid vastavas serveris, mida saab kaitsta või sinna ligipääsu piirata, nt VPN-i abil;
2. jälgitavus ja püsivus – sisestatud andmed ei kao ära, seega vajadusel on võimalik need järgi vaadata; püsiva andmebaasi kasutaja ei pea muretsema test-andmete tekitamise pärast, kuna automaattestid ja arendajad lisavad neid pidevalt juurde;
3. kiirus käivitamise ajal – rakenduse käivitamise või muu protsessi jooksutamise ajal luuakse taasloodav andmebaas nii lokaalses masinas kui ka CI-keskkonnas uuesti, mis võib võtta palju aega; püsiva andmebaasi puhul pole vaja luua andmeid ega jooksutada skripte;
4. mugavam koostöö – kui kasutusel on ühine andmebaas, on arendajatel mugavam koostööd teha, nt koos uurida tekkinud probleemi, analüüsida salvestatud andmeid jne.

2.3 Andmebaasi kasutamine arenduses

Arendajate pädevusse võivad jõuda erinevad ja mitmekesised ülesanded. Sõltuvalt arendatava ülesande taustast ja tüübist võib arendaja eelistus kasutatava andmebaasi suhtes erineda. Selles peatükis on kirjeldatud autori ja tema meeskonnaliikmete eelistused ja nõuded andmebaasile sõltuvalt ülesande taustast.

Kui arendatava ülesande sisu sõltub peamiselt äriloogika muutmise või täiendamisega, siis arendaja jaoks on eelkõige oluline test-andmete olemasolu – tihti tuleb ette olukord, kui on vaja mitu korda katsetada uut arendust, ent suurtes pärandrakendustes võib keeruliste seoste ning suuremahulise eeltöö tõttu käsitsi test-andmete loomine üsna aeganõudev protsess olla. Autor leiab, et ülalmainitud kriteeriumile võib vastata nii taasloodav kui ka püsiv andmebaas: teatud hulka test-andmeid on võimalik lisada ka taasloodavasse andmebaasi. Suurema andmemahu puhul oleks siiski andmebaasi taasloomine aeglane protsess, seega sellistel juhtudel on arenduses asendamatu ka lokaalne andmebaas, mida arendaja iga kord rakenduse käivitamisel nullist ei loo, ent vajadusel on taasloomise võimalus ikka olemas.

Arendamise nõrgem lüli on alati andmebaasiga seotud funktsionaalsus, kuna SQL-päringukeel on üsna peen ning valesti tehtud muudatuste tagajärjel võib andmebaas kasutuskõlbmatuks muutuda. Stabiilsema pärandrakenduse puhul saab andmebaasiarendusi üldiselt kvalifitseerida kahte tüüpi:

- DDL (*Data Definition Language*) ehk andmebaasiskeemi muudatused – tabelite, seoste jm andmebaasiobjektide muudatused kasutades päringuid CREATE, DROP, ALTER jne;
- DML (*Data Manipulation Language*) ehk andmete muudatused – tabelite sees oleva informatsiooni muutmine kasutades päringuid INSERT, DELETE, UPDATE jne [1].

Mõlema muudatuse tüübi puhul on autori arvates taasloodav andmebaasi olemasolu vägagi kasulik. Selliste ülesannete puhul on tihti vaja nn katsetuskeskkonda (ingl *sandbox environment*), kus saab erinevaid variante katsetada ja veenduda, et lahendus toimib õigepäraselt, muretsemata sellest, et andmebaas muutub kasutuskõlbmatuks. Siiski pakub püsiv andmebaas suuremat andmete hulka, mida on võimalik kasutada DML-muudatuste katsetamiseks või näidetega tutvumiseks.

Kokkuvõttes saab öelda, et arendaja efektiivsuse ja süsteemi kindluse tõstmiseks võiks arendusprotsessis kasutusel olla taasloodav andmebaas nii manustatud kui lokaalse andmebaasi kujul ning arendajal on vabadus otsustada, kumba ta kasutab. Siiski ei saa taasloodav andmebaas täielikult asendada püsivat andmebaasi.

2.4 Andmebaasi kasutamine automaattestimises

Samuti kasutatakse andmebaasi automaattestide jooksutamiseks ning seda nii arendus- kui ka pideva integratsiooni keskkonnas (edaspidi CI-keskkond). Antud projekti eripäraks on aga see, et integratsiooni-, *end-to-end* ning kasutajaliidese testid (edaspidi UI-testid) otseselt andmebaasi funktsionaalsust ei kontrolli, kuid kasutavad baasi lähteandmete leidmiseks ning salvestavad sinna ka tulemusena saadud andmeid. Nende automaattestide puhul reeglina kutsutakse alguses välja abimeetodid vajaliku lähteinfo leidmiseks (nt, taotlus või leping teatud staatuses), seejärel teostatakse sellega manipulatsioonid.

On aga mitu põhjust, miks ülalkirjeldatud protsess pole optimaalne ega veakindel. Sellise lähenemise peamised probleemid on:

- pidevalt kasvav andmemaht – kuna automaatteste jooksutatakse projekti tihti (sh igas eksisteerivas keskkonnas ning koodivaramu *Pull Request*-i peal), baasi tekib igapäevaselt juurde suur hulk uusi andmeid, mis tohutult suurendab andmebaasi mahtu ebavajalike andmetega;

- veakindluse puudus – võib tekkida olukord, et sobivaid lähteandmeid baasist ei leitud ning automaattest kukub seetõttu läbi, andes valenegatiivse tulemuse; selliseid juhtumeid välistatakse projektis *assume*-meetodiga, mis jätab testi vahele, kui vajalik tingimus pole täidetud, ent testide vahele jätmine pole parim lahendus probleemile;

- isoleerituse printsiibi rikkumine – iga automaattest peab olema võimalikult isoleeritud ning tema tulemust mõjutavate tingimuste arv võiks olla minimaalne; juhuslikkus peab võimalusel olema testimisprotsessis välistatud, ent antud pärandrakenduses hetkel nii ei ole.

Ülalmainitud põhjusi arvesse võttes, on autor veendunud, et taasloodava andmebaasi kasutuselevõtt automaattestimises on vältimatu ning see on oluline samm rakenduse testimisprotsessi parendamiseks.

2.5 Ettevõtte taust

Aktsiaselts LHV Pank on 1999. aastal asutatud finantsettevõte, mis on 2021. aasta seisuga Eesti suurim kodumaine finantskontsern. Algselt oli LHV alustanud investeerimispangana, ent viimase kümne aasta jooksul on ettevõte laiendanud oma tegevust ka era- ja äriklientidele suunatud pangateenuste ja -toodete osas; hetkeseisuga on ettevõttel üle 410 000 klienti ja ligi 600 töötajat [2].

LHV pakub nii era- kui ka äriklientidele muuhulgas suurt valikut finantseerimistooteid. Olles osana LHV finantseerimisosakonnast, käesoleva töö autori pädevuses on uute finantseerimistoodete arendamine ning ülalmainitud osakonda kuuluvate süsteemide ja nendega seotud protsesside parendamine ja haldamine.

2.5.1 Süsteemi taust

Süsteem, millega teeb tööd käesoleva töö autor ning kus taasloodav andmebaas kasutusele hakatakse võtma, on CDS ehk *Credit Decision System*. CDS on süsteem, kuhu jõuavad kõik klientide poolt saadetud laenutaotlused, mis edasi läbivad otsustuspuu ning selle tulemusena süsteemis otsustatakse, kas antakse laen kliendile välja või mitte, positiivsel juhul sõlmitakse selles süsteemis laenuleping.

Süsteem ise on kirjutatud Java programmeerimiskeeles. Tegemist on alates 2012. aastast arenduses oleva pärandrakendusega, mille kallal on läbi aegade töötanud suur hulk arendajaid.

3 Analüüs

Käesoleva töö analüüsimise osas tutvustatakse lugejale rakenduse lähteseisu, so mis tehnoloogiad on hetkel kasutuses, millised on piirangud jne. Uuritakse, millised võimalused ja tehnoloogiad on kiiruse, turvalisuse ja mugavuse poolest kõige sobilikumad taasloodava andmebaasi loomiseks, selgitatakse välja nende eripärad, kaasnevad potentsiaalsed takistused ja probleemid lähtudes ettevõtte sätestatud piirangutest ja nõuetest. Analüüsi tulemuste põhjal valitakse tehnoloogia edaspidise taasloodava andmebaasi rakendamiseks.

3.1 Kasutatavad tehnoloogiad

3.1.1 Rakendus

Projekt, kus taasloodavat andmebaasi hakatakse kasutusele võtma, on Spring Boot raamistikku kasutav pärandrakendus. Paljud andmebaasiga tööd lihtsustatavad tehnoloogiad (ka need, mis on Spring Boot raamistikku sisse ehitatud) ei ole veel antud projektis kasutusel, nt ei kasutata Spring Data JPA funktsionaalsust ning Hibernate teeki kasutatakse enamasti vaid valideerimiseks. Kasutatakse aga andmebaasist andmete küsimiseks enamasti LHV-siseseid teeke, uute ja ebatavaliste lahenduste kasutuselevõtt on rakendustes reglementeeritud; andmete küsimine on rakenduses enamasti kirjutatud välja SQL-lausetena.

3.1.2 Andmebaasisüsteem

Andmebaasisüsteemiks on ettevõtte Microsoft poolt arendatav süsteem SQL Server 2017 [3]. Kõik tööriistad ja rakendused, mis suhtlevad SQL Server 2017 andmebaasidega, teevad seda kasutades keelt Transact-SQL, mis on klassikalise SQL laiendus [4].

Hetkel on seadistus selline, et arenduses kasutavad kõik arendajad ühist arenduse andmebaasi „db_cds_dev“, kus saab proovida uusi arendusi ning vajadusel on samuti võimalik vajalikke katsetusi läbi viia.

Testimiseks on sõltuvalt testimise etapist või test-keskkonnast kasutusel erinevad andmebaasid. Peamiseks andmebaasiks on „db_cds“, mida kasutab test-keskkond. Samuti on olemas andmebaasid „db_cds2“ ja „db_cds_prelive“, mida kasutavad vastavalt teine test-keskkond ja *prelive*-keskkond. Need andmebaasid on reeglina stabiilsed, kuna kokkuleppeliselt, erinevalt arenduse andmebaasist, ei tohi testimise baasides käsitsi muudatusi teha või arendusi katsetada.

Andmebaas on üsna mahukas ning peamine andmebaasiskeem, mis kannab nime CDS, koosneb üle 175 tabelist. Kuna tegemist on ühe Eesti suurema finantsettevõttega, eriti suurt tähelepanu pööratakse andmevarundusele – iga andmemuudatus salvestatakse ning iga CDS skeemi tabeli juurde on loodud vastav tabel skeemi CDS_BAK, mida täidetakse trigerite abil.

3.1.3 Andmebaasi versioonikontroll ja muudatuste haldus

Andmebaasi skeemi jälgimiseks ja haldamiseks on kasutusel Liquibase teek. Liquibase võimaldab kasutada XML-, YAML-, JSON- või SQL-formaadis muutumiskomplekte (edaspidi *changeset*) andmebaasis muudatuste tegemiseks, ja antud projektis on kõik *changeset*'id SQL-failide kujul ning on koostatud kasutades Liquibase eriformaati (nn *Liquibase Formatted SQL*), mis võimaldab edastada kommentaarides *changeset*'i kohta käiva metainformatsiooni „*--changeset autor:id võti1:väärtus1 võti2:väärtus2 […]*“ [5]. Lisaks täiendavatele parameetritele saab formaadi abil eristada samas failis olevaid *changeset*'e ning määrata iga *changeset*'i identifikaator ja autor (vt Joonis 1).

```
--liquibase formatted sql

--changeset artjom:CDS_4315_1 splitStatements:false
ALTER TABLE CDS.CONTRACT DROP COLUMN SENT_TO_COURIER_SIGNING_DTIME;
ALTER TABLE CDS.CONTRACT DROP COLUMN SENT_TO_COURIER_SIGNING_BY;
ALTER TABLE CDS.CONTRACT DROP COLUMN SENT_TO_COURIER_SIGNING_NAME;
```

Joonis 1. Näide SQL-formaadis Liquibase muutumiskomplektist.

Selleks, et Liquibase jooksutaks faile, tuleb defineerida üks peamine fail, ja antud projektis on nende järjekord määratud failis „*db-changelog.xml*“. Liquibase'i jooksutamisel tehakse kindlaks, millised skriptid on juba jooksnud ning millised skriptid on uued. Uued skriptid jooksutatakse rakenduse käivitamise hetkel.

3.2 Nõuded kavandatavale lahendusele

Probleemi lahenduseks tuleb valida sobilik tehnoloogia. Järgmisena on välja toodud peamised nõuded ja kriteeriumid, millele peavad tehnoloogiad vastama ning mida kasutatakse edaspidi ka tehnoloogiate võrdluse etapil.

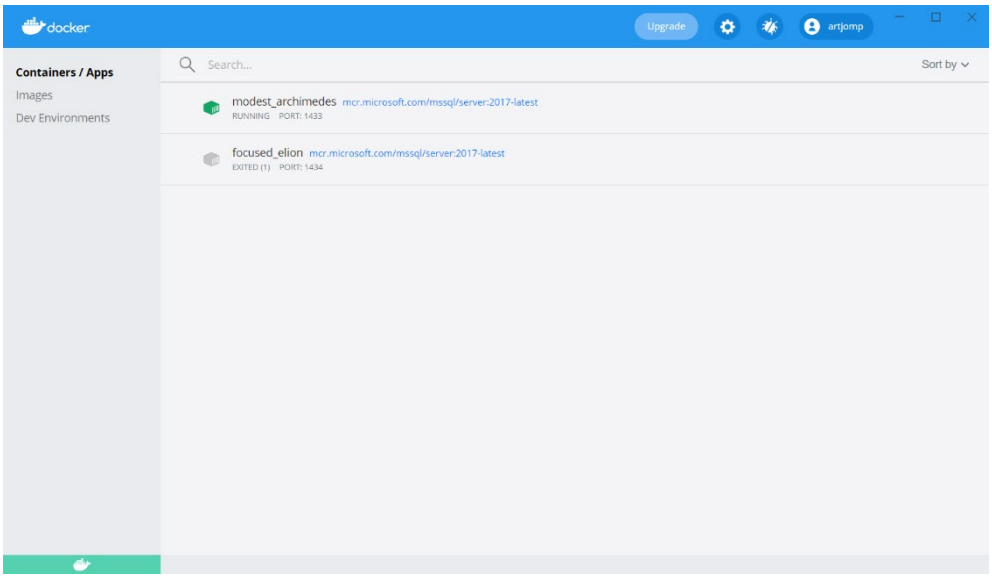
- Turvalisus: lahendus peab olema turvaline ja piisavalt kaitstud – lokaalsed ja ajutised andmebaasid võivad samuti sisaldada delikaatseid andmeid ning informatsiooni andmebaasi struktuuri kohta võib pidada ärisaladuseks. Ehkki veebipõhiste lahenduste kasutamine on prioriteetne, kõikide veebipõhiste lahenduste kasutamine ei ole siiski lubatud – sellisel juhul vajaks veebipõhine lahendus kindlasti ka detailse turvaanalüüsi läbiviimist.
- Keerukus: käesolev lahendus on eelkõige arendaja töö lihtsustamine, seega uus lahendus ei tohi nõuda väga palju lisateadmisi, lõppkasutaja ei peaks juurde õppima asju, mida läheb vaid selle lahenduse kasutamiseks vaja; õppimiskõver ei tohi olla järsk.
- Kiirus ja andmemaht: lahendus ei tohi olla liiga aeglane ega ressursimahukas. Eriti tähtis on see CI-keskkonna puhul, kus mälu ja arvutusressursid on piiratud.
- Mugavus: lahendus ei tohi kaasa tuua lisaajakulusid ega täiendavaid tööülesandeid, eeltöö lahenduse kasutamiseks peab olema minimaalne, kasutamise ajal peaks seis olema jälgitav.
- Multiplatvormilisus: lahendus peab töötama nii Windows kui ka Mac OS peal, kuna ettevõtte arendajad kasutavad mõlema operatsioonisüsteemiga arvuteid.
- Sarnasus päris keskkonnaga: lahendus peab olema võimalikult sarnane kasutuses oleva andmebaasisüsteemiga; vastasel juhul võib tekkida olukord, kus ajutise andmebaasi peal tehtud arendus ei tööta korrektselt püsiva andmebaasiga.
- Hind: hetkeseisuga ei ole eelarvesse plaanitud rahalisi ressursse antud muudatuse peale, seega eelistatakse tasuta lahendust.
- Aktuaalsus ja usaldusväärsus: kasutusele võetav tehnoloogia peab olema autorite poolt toetatud ning ka tulevikus arendatav ja hallatav; lahendust realiseeritakse ühe suurema Eesti ettevõtte näitel ja antud ettevõte kipub olema tehnoloogiate vastu suhteliselt konservatiivne, seega peab tehnoloogia olema stabiilne ning usaldusväärne ning väga uute ja eksperimentaalsete tehnoloogiate kasutamine ei ole soovituslik.

3.3 Taasloodava andmebaasi tekitamise võimalused

Kasutades erinevaid internetiallikaid, autor teostas eelneva analüüsi ning valis kolm peamist tehnoloogiat, mida saab püstitatud probleemi lahendamiseks kasutusele võtta. Käesolevas peatükis kirjeldatakse põhjalikumalt neid võimalusi ning tuuakse välja nende tugevad ja nõrgad küljed.

3.3.1 Konteineriseerimine

Tänapäeval on väga populaarseks tehnoloogiaks osutunud konteineriseerimine: tarkvara koodi kapseldamine ja pakkimine, et tulemusena töötaks see igas keskkonnas ühtlaselt ning järjekindlalt [6]. Üks kõige populaarsemaid tööriistasid selle eesmärgi saavutamiseks on Docker.



Joonis 2. Autori tehtud kuvatõmmis programmist Docker Desktop.

Docker on tööriist rakenduste loomiseks ja jagamiseks, mis teostab virtualiseerimist konteinerite kaudu [7]. Konteinereid luuakse tõmmiste (ehk *image*) põhjal ja neid saab jagada registrites, nt DockerHub [8]. Microsoft pakub andmebaasisüsteemist SQL Server mitu erinevat tõmmist, mida saab lahenduse aluseks võtta [9]. Samuti on sellel tehnoloogial olemas ka mugav kasutajaliides – Docker Desktop, kus saab mugavalt jälgida ja hallata konteinerite tööd (vt Joonis 2) [10].

Docker tehnoloogia abil tehtud lahenduste suurteks eelisteks on:

- isoleeritus: konteinerite töö ei sõltu arendaja lokaalsest masinast, seega on võimalik teha kogu eeltöö ära tõmmise kujul, jagada seda teiste kasutajatega ning olla kindel, et kõigil töötab see konteiner ühtemoodi;
- sarnasus päris keskkonnaga: tänu isoleerituse kontseptile konteineri sees on väga lihtne reprodutseerida päris keskkonda;
- väike maht: reeglina on konteineris vaid üks töötav rakendus/teenus ning konteineris asuvad ainult vajalikud failid ja andmed selle jooksutamiseks;
- populaarsus: CI-keskkonnad, IDE-d toetavad seda tehnoloogiat, sellest leidub palju dokumentatsiooni, lai valik tõmmiseid ning see süsteem on ettevõttesiseselt on juba kasutusel [7].

Peamisteks miinusteks tuuakse välja eelkõige järgmist:

- konteineriseeritud rakenduse kiirus ja jõudlus on madalam;
- järsk õppimiskurv kasutaja jaoks, kes pole varem virtualiseerimise kontseptsiooniga kokku puutunud;
- andmete püsivuse tagamine on suhteliselt keeruline [11, 12].

3.3.2 Mälupõhised andmebaasid

Mälupõhiseks andmebaasiks nimetatakse andmebaasisüsteeme, mis talletavad andmeid muutmälus ehk RAM-is kõrgemate kiiruste tagamiseks [13]. Mälupõhised andmebaasid on väga levinud eriti arenduskeskkondades või väiksemates rakendustes. Kõige levinumad mälupõhised andmebaasid Java-rakenduse keskkonnas kasutamiseks on H2, HSQLDB ja Apache Derby [14].

H2, HyperSQL (ehk HSQLDB) ja Apache Derby on Java keeles kirjutatud mälupõhised andmebaasid, mida saab Java rakendusse manustada; kõik ülalmainitud andmebaasisüsteemid on avatud lähtekoodiga ning neid saab kasutada tasuta [15, 16, 17].

Mälupõhiste andmebaaside peamised tugevad küljed on:

- kiirus: sellised andmebaasid on kõige kiiremad, kuna kasutavad muutmälu;
- maht: manustatud andmebaasid võtavad keskmiselt 1,5–3 MB ruumi [15];
- kasutuselevõtu lihtsus Spring Boot raamistikku kasutatavas rakenduses – selleks on vaja vastavalt muuta parameetreid „*.properties*“ konfiguratsioonifailis [14];

Peamiseks miinuseks on aga see, et ülalmainitud andmebaasisüsteemid kasutavad muud vastavat SQL-dialekti – H2, HSQLDB või Derby Dialect [14]. Hetkeseisuga ei ole rakenduses kasutusel andmebaasiga tööd automatiseeritavaid tööriistu (nt Spring Data JPA) ning seetõttu on paljud suuremad SQL-päringud käsitsi välja kirjutatud. Liquibase muutumiskomplektid on samuti koostatud SQL-päringute kujul. Dialektide kokkusobimatuse korral tähendab see SQL-päringute muutmist ja mõlemas dialektis Liquibase *changeset*'ide koostamist, mis on omakorda suur kaasnev lisatöö. Ehkki H2 ja HSQLDB süsteemidel on MS SQL süntaksi toetus, ei ole autori arvates täielik kokkusobivus garanteeritud [18, 19].

Tabel 1. Mälupõhiste andmebaasisüsteemide kiiruste võrdlus [20].

Näitaja		**H2**	**HSQLDB**	**Apache Derby**
Manustatud	Operatsioonide arv	1 930 995	1 930 995	1 930 995
	Kulunud aeg (ms)	13 673	20 686	105 569
	Kiirus (op. arv / s)	141 226	93 347	18 291
Klient-server	Operatsioonide arv	1 930 995	1 930 995	1 930 995
	Kulunud aeg (ms)	117 049	114 777	244 803
	Kiirus (op. arv / s)	16 497	16 823	7 887

Lähtudes kiiruste analüüsist (vt Tabel 1) [20], manustatud H2 andmebaasi kiiruse näitajad on märkimisväärselt paremad kui alternatiivsetel lahendustel. Samuti toetab Spring Boot raamistik H2 andmebaasi suhteliselt hästi, nt on olemas võimalus mugavalt konsooli sisse lülitada ja seadistada [21]. Seega, mälupõhistest andmebaasisüsteemidest otsustati edasi liikuda H2 andmebaasiga.

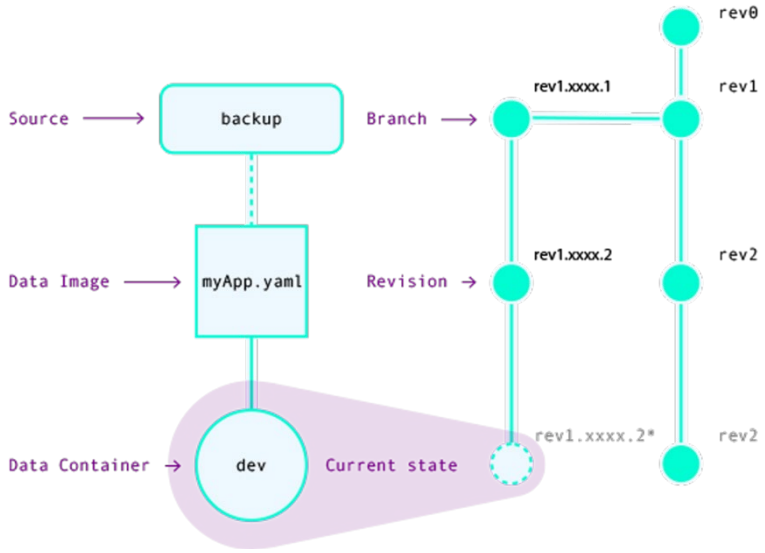
3.3.3 Pilveteenused

Tänapäeval koguvad IT-maailmas aina rohkem populaarsust pilveteenused. Uuringute käigus leidis autor, et Red Gate Software, tuntud andmebaasiteemalisi lahendusi (nt Flyway) pakkuv ettevõte, arendab uut pilveteenust, mis on suunatud lihtsustama töövoogu andmebaasidega arenduses ja CI-keskkonnas – Spawn.

Spawn on tööriist, mis automatiseerib ajutiste andmebaasikoopiate loomist ja haldamist ning võimaldab mugavalt jagada neid koopiaid arenduse meeskonnaliikmete vahel ning kasutada neid arendus- ja testimisprotsessides [22]. Hetkel toetab Spawn juba 5

andmebaasisüsteemi, k.a Microsoft SQL Server. Tõmmise jagamine meeskonnaga töötab hetkel vaid GitHub gruppide kaudu.

Üldiselt saab Spawn tehnoloogia voogu kokku võtta järgmiselt: andmete allikast (*Source*), milleks saab olla tühi andmebaas, skriptid või varukoopia, tehakse tõmmis (*Data Image*), mida saab meeskonnaliikmetega jagada ning mille põhjal luuakse konteiner (*Data Container*); samaaegselt võib andmebaasil olla mitu erinevat seisu (*Revision*) ning neid eristatakse või moodustatakse harude (*Branch*) abil (vt Joonis 3) [23].



Joonis 3. Spawn tehnoloogia kontsepti kokkuvõte [23].

Tehnoloogia suurimaks eeliseks on mugavus: uus andmebaasikoopia olem luuakse ühe käsuga ning see ei võta lokaalses masinas ruumi. Siiski on antud tehnoloogia hetkel vaid kinnise *Beta*-testimise staadiumis ning tegemist on eksperimentaalse tootega, seega sellise lahenduse kasutuselevõtt peab olema kaalutud.

3.4 Tehnoloogiate võrdlus

Peatükis 3.2 tõi autor välja kõige olulisemad nõuded lahendusele ning järgnevalt teostatakse nende järgi võrdlev analüüs, kus osaleb kolm peatükis 3.3 välja valitud tehnoloogiat: andmebaasi konteineriseerimine Dockeri abil, mälupõhine andmebaas H2 ja Spawn tehnoloogia (vt Tabel 2).

Tabel 2. Tehnoloogiate võrdlus peatükis 3.2 välja toodud kriteeriumite järgi.

Kriteerium	**SQL Server konteineris**	**H2 andmebaas**	**Spawn tehnoloogia**
Turvalisus	Loodav andmebaas on ainult lokaalne; tõmmised saavad olema vaid arendaja lokaalsetes masinates või vastavas ettevõtte repositooriumis	Andmebaas on ainult lokaalne, andmebaasi loomise skriptid on juba olemas ettevõtte repositooriumis	Andmebaasi tõmmiseid hoitakse pilveteenuse serverites, mis ei ole ettevõtte vaates soovituslik
Keerukus	Dockeri käskude ning konteineriseerimise põhimõtete tundma õppimine võib esialgu võtta aega, kuid siiski andmebaasisüsteem on sama mis püsival baasil, seega võib eeldada, et selles on arendaja kompetentne	H2 andmebaas toetab SQL Server süntaksit, kuid siiski on neil erinevusi; kasutamise ajal võib esile tulla H2 süsteemi spetsiifilisi vigu; samuti probleeme võib esineda Liquibase migratsioonidega, mida koostatakse just SQL-formaadis	Uue toote kasutamine võib lähimal ajal olla väga keeruline vähese informatsiooni, dokumentatsiooni hõreduse ning ka lähenemise uudsuse tõttu
Kiirus/andmemaht	Konteineri sees asub SQL Server ja kõik vajalikud andmed selle töötamiseks, kiirus on sama mis oleks ka lokaalsel SQL Server andmebaasiserveril	Mälupõhised andmebaasid on reeglina kõige kiiremad ja väiksemahulised	Lokaalses masinas teenus ressursse ja mahtu ei kasuta, kiirus sõltub internetiühendusest ning pilveteenuse serverite võimsusest
Mugavus	Kasutamiseks on vaja masinasse paigaldada Docker, ülejäänud töö saab teha rakenduse käivitamise ajal automaatselt; samuti on mugavamaks haldamiseks olemas ka kasutajaliides Docker Desktop	Baas käivitatakse automaatselt koos rakenduse käivitamisega; mugavamaks haldamiseks saab kasutada H2 konsooli	Kasutuses on väga mugav; võimalik, et teenuse esialgse konfigureerimisega saab vaeva näha

Kriteerium	**SQL Server konteineris**	**H2 andmebaas**	**Spawn tehnoloogia**
Multiplatvormilisus	Dockeri on võimalik kasutada Windows, Mac OS ja Linux süsteemides, Docker Desktop on saadaval Windows ja Mac OS-i kasutajatele	Operatsioonisüsteem ei mõjuta andmebaasi töötamist	Teenuse CLI on võimalik paigaldada Windows, Mac OS ja Linux süsteemidele
Sarnasus päris keskkonnaga	Konteineris jooksev andmebaasiserver on täpselt sama mis päris keskkonnas, seega kokkusobivusega ei saa probleeme tekkida	Ehkki MS SQL süntaksi toetus on H2 andmebaasil olemas, H2 ja SQL Server ei ole samad süsteemid, seega kokkusobivuse probleemide tekkimine on üsna tõenäoline	SQL Server andmebaasisüsteem on täies ulatuses toetatud, kuid vähese kontrolli ja kogemuse puuduse tõttu ei saa olla kindel, et kokkusobivuse probleemid on välistatud
Hind	Docker – tasuta; ehkki SQL Server on litsentseeritav tarkvara, selle isiklik kasutamine tasuta arenduseks on lubatud	Tasuta	Tasuta; siiski ei saa selles kindel olla, kuna tegemist on *Beta*-testimises oleva tarkvaraga
Aktuaalsus ja usaldusväärsus	Docker on tänapäeval väga laialt kasutatav tarkvara, mida pidevalt arendatakse	H2 andmebaasi kasutatakse väga palju Java rakenduste arendamises, see on stabiilne ja kindel tehnoloogia	Arendajaks on üsna usaldusväärne firma, millel on olemas mitu edukat toodet IT-turul; hetkel on toote arendus siiski väga varases etapis

Läbiviidud analüüsi tulemuste põhjal otsustas autor kasutada lahenduseks konteineriseerimise tehnoloogiat, otsuse peamistes argumendiks osutus sarnasus päris keskkonnaga, konteineriseerimise populaarsus, veakindlus ning rakenduse kokkusobivuse probleemide välistatus.

3.5 Lahenduse kavandamine

Kuna uus funktsionaalsus hõlmab ainult arendus- ja testimiskeskkonda ega mõjuta lõppkasutaja kogemust, otsustas autor implementeerida eraldi mooduli nimega „*mod-ephemeral-db*“. Projekti ehitamiseks ja sõltuvuste haldamiseks kasutatakse tööriista Gradle ja selle abil on võimalik võtta moodulit sõltuvuseks ainult teatud keskkonnas – antud juhul oleks selleks arenduskeskkond ning integratsiooni- ja kasutajaliidesetestide keskkond, selline lähenemine võimaldab etapilist üleminekut püsivast andmebaasist taasloodavale. Samuti välditakse mooduliga koodi duplitseerimist.

Ülalmainitud uue mooduli eesmärgiks oleks uute konfiguratsioonide hoidmine ning Docker konteinerite loomise ja haldamise automatiseerimine. Selleks on võimalik kasutada teeki Testcontainers, mis pakub mugavat API Dockeri konteinerite loomiseks ja haldamiseks [24]. Siiski tuleb arendajatele, kes teatud põhjustel ei soovi kasutada rakenduse poolt loodavaid konteinereid või soovivad kasutada nt lokaalset serverit, tagada võimalus mugavalt luua ühendusi ka muude ajutiste baasidega.

4 Lahenduskäik

Käesoleva töö praktilises osas autor kirjeldab taasloodava andmebaasi implementeerimist analüüsi käigus valitud tehnoloogia põhjal. Lugejale tutvustatakse peamisi probleeme, selgitatakse ja põhjendatakse autori lähenemist nende lahendamiseks. Samuti kirjeldatakse töös konfiguratsiooni koostamist arenduse ja automaattestimise keskkonna jaoks ning automatiseerimist arendajate töövoo mugavdamiseks. Viimasena tutvustatakse lugejale rakenduse testitavuse tagamise protsessi taasloodava andmebaasiga.

4.1 Andmebaasi käivitamine lokaalses masinas

Esimeseks ülesandeks osutus andmebaasi loomine uues keskkonnas, ja kuna lahenduse elluviimiseks on vaja teostada palju katsetusi, otsustas autor kohe alustada Docker tõmmise ehitamisest, et andmebaasi taasloomise protsess oleks kiirem.

Selleks, et andmebaasi konteinerit oleks võimalik teiste arendajatega jagada, on vaja luua Docker tõmmis. Nagu oli eelnevalt mainitud, Docker-põhilise lahenduse eeliseks on võimalus teha kogu eeltöö ära, et teised meeskonnaliikmed sellega vaeva ei näeks. Uurimise käigus autor leidis, et kasutuseks valmis andmebaasi tõmmise loomiseks on olemas kaks peamist varianti: manuaalne – luua tõmmis käsitsi muudetud konteineri põhjal, või automaatne – teha eeltöö ära tõmmise loomise skriptis. Mõlema variandi tulemusena saadakse Docker-tõmmis, mida saab edasi üles laadida (nt DockerHub keskkonda või sisemisse repositooriumisse), et teised arendajad sellele ligi saaksid.

Esimese variandi puhul on algoritm järgmine: eelkõige on vaja luua konteiner olemasoleva andmebaasiserveri tõmmise põhjal (käesoleva töö puhul on selleks SQL Server); edasi on vaja töötavas konteineris käsitsi jooksutada vajalikud skriptid, sh andmebaasi loomine, andmetega täitmine; viimase sammuna teha sellest konteinerist tõmmis kasutades käsku "*docker container commit*".

Teine variant on üsna sarnane esimese variandiga, kuid protsess on automatiseeritud: elluviimiseks on aga vaja koostada Dockerfile, kus on ära kirjeldatud vajalikud sammud

vajaliku tõmmise loomiseks. Seejärel on võimalik tõmmist ehitada kasutades käsku "*docker build*".

Ehkki esimene variant on ühekordseks loomiseks lihtsam ning korrektse Dockerfile-skripti loomiseks tuleb rohkem aega panustada, autori arvates sobib siiski teine variant loomiseks paremini, kuna sel juhul on tõmmise loomise algoritmisse palju mugavam muudatusi teha. Samuti saab Dockerfile skripti koostada nii, et andmebaasi skriptide lugemine ning käivitamine oleks üldistatud ja automatiseeritud – sellisel juhul, kui on näiteks vaja muuta või juurde lisada andmebaasi loomise skripte, tervet protsessi ei ole uuesti vaja käsitsi läbi teha.

Dockerfile skripti koostamist alustatakse reeglina lähtetõmmise määramisest [25], ning antud juhul lähtetõmmiseks on Microsoft SQL Server andmebaasiserveri tõmmis, siiski mitte ametlikust Microsoft'i serverist, vaid ettevõttesisesest repositooriumist, kus asuvad Docker'i tõmmised, mis on ettevõttes kasutamiseks lubatud. Järgnevalt tuleb määrata vajalikud keskkonnamuutujad: MSSQL_SA_PASSWORD – andmebaasiserveri admin-kasutaja parooli väärtus, ning ka ACCEPT_LICENSE – kuna SQL Server on litsentseeritav tarkvara, siis enne kasutamist tuleb litsentsiga nõustuda ehk väärtuseks määrata "*Y*" [26].

Oluline osa arendusprotsessist on silumine (*debug*) – tihti tuleb ette olukord, millal on vaja sirvida logisid, et aru saada, mis skripti jooksutamise või programmi kasutuse ajal valesti on läinud. Vaikimisi lähevad SQL Server andmebaasiserveri logid kindlasse kataloogi „*var/opt/mssql/log*" [26], ja selleks, et andmebaasiserveri logidele ligi saada ning säilitada neid mitme jooksutamise vahel, lõi autor konteineris VOLUME käsu abil püsiva andmekogumi.

Järgmine samm on automatiseeritud SQL-skriptide jooksutamine. Probleem seisneb aga selles, et otse Dockerfile sees ei ole võimalik seda mugavalt teha: andmebaasi loomine koosneb mitmest erinevast käsust (serveri käimapanek, ooteaeg ja SQL-skriptide jooksutamine), ent peale Dockeri RUN käsku luuakse tulemusena uus vahepealne tõmmis. RUN käsu sisse saab panna mitu käsku, kuid see pole loetav ega edaspidi mugavalt hallatav. Seega, autor otsustas koostada Shell-skripti, mis paneks SQL Server'i käima, ootaks kuni see lõplikult käima läheks ja siis jooksutaks kõik SQL-skriptid, mis

asuvad konkreetses kataloogis (vt Joonis 4), vahendi „sqlcmd“ abil, määrates lippude kaudu ka vajalikud parameetrid.

```
#!/bin/bash

echo "Starting SQL Server..."
(/opt/mssql/bin/sqlservr &) | grep -m 1 -q "Service Broker manager has
started"
echo "SQL Server started!"

for file in /opt/db-scripts/*.sql
do
    echo "Executing '${file}...'"
    /opt/mssql-tools/bin/sqlcmd -U sa -P $MSSQL_SA_PASSWORD -e
        -i $file
done
```

Joonis 4. Shell-skript andmebaasi loomise SQL-skriptide jooksutamiseks.

Seejärel saab Dockerfile-skriptis luua kataloog (antud juhul „*/opt/db-scripts*“), kopeerida sinna SQL- ja Shell-skriptid ADD/COPY käsu abil ning panna need käima RUN käsuga (vt Joonis 5). Nüüd saab valmis saanud Dockerfile-i järgi tõmmise luua käsuga „*docker build -t <tõmmise nimi>*“. Kui tõmmis on ehitatud, siis saab seda konteinerina käima panna käsuga „*docker run -t <tõmmise nimi>*“. „-p“ argumendiga saab täiendavalt määrata, milline konteineri port on lokaalse masina pordiga vastavuses – SQL Server kuulab vaikimisi pordil 1433; „-v“ argumendiga saab luua konteineri sees- ja väljaspool asuvate kataloogide vahel lingi, mida on tarvis näiteks logide sirvimiseks [27].

```
FROM *sisemine süsteem peidetud*/server:2017-latest

RUN mkdir -p /opt/db-scripts
ADD db/* /opt/db-scripts/
ADD init.sh /opt/db-scripts/

ENV MSSQL_SA_PASSWORD=*parool*
ENV ACCEPT_EULA=Y

EXPOSE 1433/tcp
VOLUME /var/opt/mssql/log

RUN /opt/db-scripts/init.sh
```

Joonis 5. Dockerfile-skripti sisu lokaalse andmebaasi tõmmise loomiseks.

4.2 Taasloodavuse tagamine

Järgmine ülesanne, mida peab lahendama taasloodava andmebaasi kasutusele võtmiseks, on selle taasloodavuse tagamine ning selle edukus koosneb kahest osast:

- andmebaasiserver peab käivituma uues kohas ilma vigadeta ning SQL-skriptide jooksutamine õnnestus;
- andmebaas peab olema rakendusele kasutuskõlblik, st rakendus peab saama käivituda.

Kuna eelnevalt oli otsustatud, et andmebaasi loomisprotsess toimub tõmmise ehitamise ajal, siis võib järeldada, et esimene kriteerium on täidetud, kui tõmmis on edukalt ehitatud, konteinerit käivitades läheb ka MS SQL Server ilma probleemideta käima ning kõik vajalikud lähteandmed (CDS ja CDS_BAK andmebaasid, tabelid ja nende sisu) on loodud.

Teine kriteerium on täidetud, kui rakendus on võimeline looma ühendust konteineris oleva uue andmebaasiserveriga ning saab käivituda ilma takistusteta – kõik Liquibase migratsioonid on õnnestunud, vajalikud andmed rakenduse korrektseks jooksutamiseks on andmebaasides olemas.

4.2.1 SQL-skriptide kohandamine

Sõltuvalt sellest, mis keskkonnas rakendust jooksutatakse, kasutatavatel andmebaasidel võivad olla erinevad nimetused, kasutajanimed jne. Olemasolevates skriptides oli selleks järgmine lahendus: failide alguses deklareeritakse SQL-muutujad võtmesõna „DECLARE“ abi ning vastavad väärtused tuleb määrata samas failis võtmesõnaga „SET“. See pole autori arvates kõige optimaalsem lahendus, kuna muutujate väärtuse määramiseks tuleks iga kord muuta mitu faili.

Vahendi „sqlcmd“ dokumentatsiooni uurides, leidis aga autor tunduvalt parema lahenduse, tänu millele saaks määrata kõik muutujad ühes kohas – skriptimuutujad. Skriptimuutujad (*Script Variables*) on sellised muutujad, mida saab kasutada SQL-lausete või -skriptide sees ning mida on võimalik määrata mitmel erineval viisil, nt keskkonna muutuja, käsurea või sqlcmd „-v“ argumendi kaudu [28].

Autor otsustas muutujaid määrata keskkonnamuutujate kaudu, kuna neid saab mugavalt määrata ka Dockerfile skripti sees ENV käskluse abil. Selleks, et sqlcmd vahend kõik muutujad vajalikkudes kohtades väärtustega asendaks, tehti vastavalt muudatused ka igas SQL-skriptifailis: eemaldati ebavajalikud muutuja deklareerimised ning asendati SQL-muutuja notatsioon ehk @<muutuja nimi> sqlcmd notatsiooniga $(<muutuja nimi>).

Kui muudatused olid tehtud, oli autoril võimalik valideerida andmebaasiserveri lahenduse korrektsust. Tõmmise ehitamine käsu „*docker build*“ abil õnnestus ning tulemusena saadud tõmmis läks andmebaasiserveri konteinerina käima ilma vigadeta. Kasutatava IDE andmebaasi funktsionaalsus võimaldas autorile andmebaasiserveri külge ühendada ja veenduda, et esimene sätestatud taasloodavuse kriteerium on täidetud: mõlemad andmebaasid olid edukalt loodud koos kõigi vajalike tabelite, kasutajate ja andmetega.

4.2.2 Liquibase skriptide järjestamine

Peatükis 3.1.3 tõi autor välja, et rakenduses kasutatakse andmebaasi skeemi versiooni kontrollimiseks lahendust Liquibase.

Projekti *changeset*'id on jagatud kataloogidesse vastavalt tüübile (jooksutamise järjekord on säilitatud): DDL-muudatused, funktsioonid, protseduurid, vaated, trigerid, DML-muudatused. Failide jooksutamise järjekord ei ole siiski määratud: kasutusel on *includeAll* atribuudid, mis tähendab, et jooksutatakse järjest kõik kataloogis olevad failid tähestiku järjekorras failinime järgi. Kõik *changeset*'ide tüübid, v.a DML ja DDL, on seadistud jooksutamiseks peale igat tehtud muudatust vastavas *changeset*'i failis. DML ja DDL skriptid on mõeldud ühekordseks jooksutamiseks ja failide nimed konventsiooni kohaselt algavad Jira keskkonnast saadud ülesande identifikaatoriga „CDS-*XXXX*“, kus *XXXX* on ülesande loomise järjekorranumber, seega ülesande loomise järjekord määrab ka *changeset*'ide jooksutamise järjekorda.

Ülalkirjeldatud lähenemine on väga probleemne, kui tegemist on pikka aega arenduses oleva pärandrakendusega. Peamiseks probleemiks on see, et ülesannete numbrid ei pruugi alati olla kronoloogilises järjekorras, st et mõni varasemalt loodud ülesanne võib olla tehtud hiljem kui uuem ülesanne. Seetõttu rakendus ei saanud käivituda: vanemates *changeset*'ides olid kustutatud mõned andmebaasiobjektid (nt veerud või tabelid), mille olemasolu eeldasid uuemate *changeset*'ide SQL-skriptid.

Teine probleem, mis seoses praeguse lähenemisviisiga esile tuleb ja taasloodavust takistab, on DML- ja DDL-skriptide eristamine. See toob kaasa kronoloogilise konflikti: uuemad DDL-skriptid jooksutatakse enne vanemaid DML-skripte.

Autor jõudis järelduseni, et praegune süstematiseerimine ei ole jätkusuutlik ega taasloodava andmebaasi kasutusele võtmiseks sobilik. Liquibase *changeset*'ide nimetamise ja süstematiseerimise konventsioonis tuleb teha järgmised muudatused:

1. lõpetada DDL- ja DML-skriptide eristamine ning need skriptid tuleb teisaldada ühte kataloogi, nt „*./changesets*";

2. skriptide kronoloogilise järjekorra tagamiseks lisada failinime algusesse loomise kellaaeg, et failinimed vastaks formaadile „<aasta><kuu><päev>_CDS-*XXXX*_<dml/ddl>.sql". Kuupäeva asemele oli kaalutud ka järjekorranumbrit, kuid selline lahendus oleks pigem ajutine ega pruugi olla jätkusuutlik – võib tekkida probleem, et kaks ülesannet jõuavad samaaegselt arendusse ja *changeset*'idele määratakse samad järjekorranumbrid.

Korrastatud skriptid ei tohi olla konfliktis ühises kasutuses olevate andmebaasidega, seetõttu osutus olemasolevate *changeset*'ide muutmine probleemseks, kuna muudetakse nii asukohta kui ka failinime. Liquibase salvestab informatsiooni jooksutatud *changeset*'ide kohta eraldi andmebaasi tabelisse „DATABASECHANGELOG", kus primaarsed võtmed on muudatuse ID, muudatuse autor ning asukohast ja failinimest koosnev kohaviit. Primaarsete võtmete kontroll on viis, kuidas Liquibase piirab *changeset*'ide jooksutamist ning kui üks parameetrist on failis muudetud, siis Liquibase jaoks on tegemist uue *changeset*'iga.

Varem oli mainitud, et kõik Liquibase *changeset*'id on koostatud kasutades Liquibase eriformaati, seega probleemi lahendamiseks asus autor uurima selle dokumentatsiooni. Tulemusena leidis autor, et olukord, kui on vajalik failid ümber tõsta või nimetada, on Liquibase poolt läbi mõeldud – parameeter „*logicalFilePath*" võimaldab hetkekohaviita ümber kirjutada, et andmebaasi minev kohaviit oleks korrektne.

Kuna antud projektis oli plaani teostamise hetkel üle 50 DDL ning üle 140 DML skripti, nende käsitsi muutmine oleks väga aeganõudev protsess, seega autor otsustas seda automatiseerida ning koostas skripti Python keeles, mis loeks kõik failid „*./ddl*" ja „*./dml*"

kaustadest ja kopeeriks need uude kataloogi „*/changesets*“, lisades failinime ette „*git log*“ käsu abil saadud faili esmakordse loomise kuupäeva ning määrates iga *changeset*'i metainformatsiooni parameetri „*logicalFilePath*: <praegune failinimi>“.

Probleemseks aga osutus üks *changeset*, kus oli olemas vastavas formaadis metainformatsioon (k.a autor ja ID), kuid faili alguses puudus rida „*--liquibase formatted sql*“, mida on vaja selleks, et Liquibase interpreteeriks failis olevaid kommentaare kui metainformatsiooni. Ehkki autor lisas vastava rea faili algusesse, probleemi see ei lahendanud, kuna andmebaasi tabelisse „DATABASECHANGELOG“ oli salvestatud vale informatsioon: autoriks oli määratud „*includeAll*“ ning ID-ks – „*raw*“. Autor ühendus püsiva andmebaasiga, võttis sealt salvestatud info selle *changeset*'i kohta ja viis failis oleva informatsiooni sellega vastavusse.

Probleem siiski ei lahenenud, sest failil muutus veel üks parameeter, mida määratakse Liquibase poolt muutumiskomplektide valideerimiseks – faili MD5-räsi. Seda arvutatakse ja salvestatakse selleks, et hiljem saaks kindlaks teha, et peale läks täpselt sama sisuga *changeset*. Liquibase dokumentatsiooni kohaselt on võimalik ka MD5-räsi üle kirjutada: parameetri „*validCheckSum*“ abil saab määrata kõik võimalikud väärtused, mida saab Liquibase õigeks pidada [29]. Kui faili oli vana MD5-räsi väärtus lisatud, sai probleem lahendatud ning rakendus läks edukalt taasloodava andmebaasiga käima.

4.2.3 Liquibase skriptide arhiveerimine

Valideerimise tulemusena teostatud Liquibase probleemi lahendus täies ulatuses kasutusele ei jäänud. Võttes arvesse ettevõtte spetsiifikat, suurt tähelepanu pööratakse andmevarundusele, mida teostavad ülalmainitud trigerid. Hetkelahendust valideerides tuli esile olukord, et kui uus triger luuakse või muudetakse peale andmete lisamist, siis uued väljad jäävad varutabelitesse salvestamata. Trigerite loomine ja muutmine ei saa olla ka enne *changeset*'ide jooksutamist – sel juhul muudatused tabelite struktuuris tuleksid peale hiljem, kui trigerid.

Teine aspekt, mis võib tulevikus arendus- ja testimisprotsessi oluliselt mõjutada, on pidevalt kasvav *changeset*'ide hulk. Hetkeseisuga läheneb DML/DDL failide arv 200-le ning nende jooksutamine baasi loomisel võtab aega ca 20 sekundit lokaalses masinas. Pideva integratsiooni keskkonnas, kus arvutus- ja mäluressursid on piiratud, jooksutamine võib veelgi rohkem aega võtta.

Seega oli otsustatud, et ehkki kellaaeg peab nüüdsest alates failinimes olema, ühist „*./changesets*“ kataloogi kasutusele ei võeta. Selle asemel jätkatakse vana süsteemiga ehk DDL ja DML skriptide eristamisega ning andmebaasi hetkeseisust tehakse uus lähtepunkt (nn null-punkt): selleks tuleb vanad Liquibase skriptid arhiveerida ning andmebaasi loomise skriptid vastavalt uuendada. Arhiveerimisest saab perioodiline protseduur, mis hakkab toimuma orienteeruvalt iga poole aasta tagant.

Andmebaasi loomise skripte otsustati genereerida olemasoleva andmebaasi pealt, milleks osutus test-keskkonna andmebaas. Uusi andmebaasi skripte on saadud tarkvara *SQL Server Management Studio* abil ning genereeriti nii andmebaasi struktuuri skripte kui ka staatilisi andmeid (nt tootekoodid, tüübid jne). Lisaks sellele oli genereeritud skript tabeli „DATABASECHANGELOG“ täitmiseks käesoleva aasta algusest loodud kirjetega; see oli tehtud sellepärast, et on võimalik, et mõned ülesanded on veel arenduses ja faile ei saa veel arhiveerida, kuid skriptid olid andmebaasi peal juba jooksutatud.

Liquibase skriptide arhiveerimist oli võimalik teostada eelnevalt tehtud töö peal: selleks muudeti kataloogi „*./changesets*“ nime „*./archive*“ peale ning faili „*db-changelog.xml*“ tagastati algseisu. Tänu eelnevale *changeset*'ide korrastamisele jääb võimalus vajadusel ka vana andmebaasi loomise skriptide põhjal baasi luua – selleks tuleb *changelog*-failis lisaks olemasolevatele kataloogidele määrata skriptide allikaks „*./archive*“ kataloogi.

4.3 Taasloodava andmebaasi mooduli loomine

Peatükis 3.5 otsustas autor taasloodava andmebaasiga seotud koodi hoidmiseks luua rakendusse eraldi mooduli. Mooduli nimeks oli määratud „*mod-ephemeral-db*“, siis lisati moodul alamprojektina kasutades „*settings.gradle*“ failis „*include*“ meetodit. Viimase asjana lisati uus alamprojekt sõltuvusena arenduskeskkonda, integratsioonitestide ning kasutajaliidesetestide keskkonda failis „*build.gradle*“, kasutades vastavalt „*developmentOnly*“, „*integTestImplementation*“ ja „*uiTestImplementation*“ meetodeid. Alamprojekti „*build.gradle*“ failis tehti vajalik konfiguratsioon ning defineeriti kõik sõltuvused, mida on tarvis selle mooduli korrektseks töötamiseks (vt Joonis 6).

```
plugins {
    id 'java'
}

java {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

compileJava.inputs.files(processResources)

dependencies {
    recommendedVersion platform('org.springframework.boot:spring-boot-
dependencies:2.3.7.RELEASE')
    recommendedVersion platform('org.testcontainers:testcontainers-
bom:1.15.3')

    annotationProcessor 'org.springframework.boot:spring-boot-
configuration-processor'
    annotationProcessor 'org.projectlombok:lombok'
    compileOnly 'org.projectlombok:lombok'

    implementation 'org.springframework.boot:spring-boot-starter'
    implementation 'org.testcontainers:mssqlserver'
    implementation 'org.apache.commons:commons-dbcp2'
}
```

Joonis 6. Alamprojekti „*mod-ephemeral-db*“ Gradle-konfiguratsioonifail.

Eelnevalt oli mainitud, et konteinerite loomise ja haldamise automatiseerimiseks võetakse kasutusele teek Testcontainers. Testcontainers on Java keeles loodud teek, mis on tänapäeval üsna levinud ning seda kasutatakse põhiliselt testimiskeskkonnas, nt integratsioonitestide ajal lühiajaliste sõltuvuste (andmebaaside, veebiserverite jms) loomiseks või kasutajaliidesetestide ajal brauserite loomiseks [24]. Teegil on suur arv mooduleid, mis pakuvad mugavat API laialt levinud konteineritega töötamiseks (nt SQL Server, MySQL, RabbitMQ, WebDriver, jt); vajadusel on võimalik luua ka enda mooduleid. Jooksutamiseks vajab teek vaid Docker keskkonda.

Mugavamaks haldamiseks ning konteineri funktsionaalsuse kapseldamiseks otsustas autor luua klassi *DatabaseContainer*, kus sai ära kirjeldada spetsiifilised parameetrid ja vajalik lisafunktsionaalsus. Ülalmainitud uus klass pärineb klassist *MSSQLServerContainer*, mis omakorda sisaldab vajalikku funktsionaalsust SQL Server konteineriga suhtlemiseks. Rakenduse jaoks sobiva andmebaasi loomiseks ning korrektseks töötamiseks on vajalik eelkõige klassi konstruktoris määrata Docker tõmmise

nimi ja silt (*tag*), seejärel edastada edaspidi vajalikud parameetrid (kasutajanimi, parool ja andmebaasi nimi) ning suurendada mälu limiiti 2 GB peale. Selleks, et Testcontainers teek saaks aru, kuidas tõmmist käsitleda, oli vaja määrata meetodi „*.asCompatibleSubstituteFor()*“ abil, et andmebaasi tõmmis saab asendada MS SQL Server ametlikku tõmmist. Kasutajanime ja andmebaasi nime lisamine ei olnud aga ülemklassis implementeeritud, seega see funktsionaalsus pidi olema lisatud autori poolt.

Peatükis 4.1 oli mainitud, et SQL Server konteineri kasutamiseks tuleb aktsepteerida litsents. Ehkki kohandatud Docker tõmmises on litsents juba aktsepteeritud, tuleb see uuesti Testcontainers teeki kasutades ära teha; selleks saab lisada „*resources*“ kataloogi fail „*container-license-acceptance.txt*“, kus on formaadis „<tõmmise nimi>:<silt>” välja kirjutatud tõmmised, mille litsentsid tuleb aktsepteerida. Samuti saab „*resources*“ kataloogi lisada fail „*testcontainers.properties*“, kus on võimalik võtme-väärtuse paaridena üldisemalt ära konfigureerida teegi tööd, nt „*testcontainers.reuse.enable=true*“ abil autor määras, et loodud konteineri taaskasutus on lubatud, st uut konteinerit ei looda, kui on võimalik kasutada varasemalt loodud konteinerit.

Selleks, et Testcontainers teegi poolt loodava konteineri konfigureerimise muutmine oleks paindlikum ja mugavam, autor otsustas hoida vajalikke parameetreid uue mooduli Spring Boot konfiguratsioonifailides „*application-{profiili nimi}.properties*“, milleks on:

- *testcontainers.datasource.username* – andmebaasi kasutajanimi;
- *testcontainers.datasource.password* – andmebaasi parool;
- *testcontainers.docker.image* – Docker tõmmise nimi;
- *testcontainers.docker.image-tag* – Docker tõmmise silt (versioon).

Teistes konfiguratsioonifailides olid eelnevad ära määratud vajalikud muutujad, mis on sõltuvalt keskkonnast erinevad. Ent taasloodava andmebaasi on mõned väärtused keskkonnast sõltumatud ning võivad määratuist erineda, seega mooduli konfiguratsioonifailides pidi ümber kirjutama kaks muutujat:

- *datasource.name* – andmebaasi nimi;
- *dsCdsLiquibase.dbName* – andmebaasi nimi Liquibase jaoks.

Kuna tegemist on uute muutujatega, mille tähendus ja mõte ei pruugi olla teistele koodi lugejatele olla ühtepidi mõistetav, otsustas autor koostada olemasolevatele (ning ka

hiljemalt lisatud) muutujatele dokumentatsiooni, kasutades teeki Spring Boot Configuration Processor, mis võimaldab IDE-l pakkuda abi ja vihjeid konfiguratsiooniga töötavale arendajale [30]. Dokumentatsiooni koostatakse JSON-formaadis ning see asub failis „*META-INF/additional-spring-configuration-metadata.json“* (vt Joonis 7).

```
{
  "properties": [
    {
      "name": "sqlserver.container",
      "type": "java.lang.Boolean",
      "defaultValue": "true",
      "values": [
        {
          "value": true,
          "description": "App will connect to the database that will be
created in the container."
        },
        {
          "value": false,
          "description": "App will connect to the existing local server."
        }
      ],
      "description": "Choose whether to run local database in container."
    }
  ]
}
```

Joonis 7. Näide *Spring Boot Configuration Processor* kasutusest.

Taasloodava andmebaasi kasutamine saab olema pisut erinev sõltuvalt hetkekeskkonnast, seega uue funktsionaalsuse edaspidine konfigureerimine on jagatud kaheks vastavalt keskkondadele: arenduskeskkond ja automaattestide keskkond. Järgnevates peatükkides on lähemalt kirjeldatud taasloodava andmebaasi kasutuselevõtt igas keskkonnas.

4.3.1 Arenduskeskkonna konfigureerimine

Eelkõige asus autor seadistama arenduskeskkonda ning selle konfiguratsiooni peamiseks sihiks oleks see, et nii taasloodava andmebaasi käivitamine kui ka lokaalse ja püsiva andmebaasiga ühendamine oleks teistele arendajatele võimalikult mugav. Selleks otsustas autor kasutada profileerimist – moodulis oli loodud konfiguratsiooniklass *DatabaseConfig* ning selle profiilideks oli määratud „*autotest*“ ja „*local*“, st et Spring Boot kasutab seda konfiguratsiooniklassi, kui vähemalt üks ülalmainitust profiilist on aktiivsete profiilide hulgas.

Rakendus saab andmebaasiga ühendust luua tänu Spring raamistiku sõltuvuste süstimise (ingl *Dependency Injection*) mehhanismile: konfiguratsiooniklassides on ära defineeritud „*cdsDataSource*“ nimega *DataSource* tüüpi *bean* – objekt andmebaasi ühenduste saamiseks, mida nõuavad sõltuvusena teised rakenduse komponendid ja sõltuvused. Autor lõi taasloodava andmebaasi moodulis meetodi, mis tagastab *DataSource* objekti ning teistes konfiguratsioonides oli vastavat tüüpi *bean*'idele lisatud annotatsioon *@Profile(„!autotest & !local“)*, mis väldib *bean*'i registreerimist kui kumbki profiil on aktiveeritud, või *bean*'id olid üldse konfiguratsiooniklassidest eemaldatud (nt integratsiooni- ja UI-testide konfiguratsiooni puhul).

Kuna ühendust peab saama tekitada nii ajutise kui ka lokaalse andmebaasiga, siis *DataSource bean*'i tagastav meetod vajab sisendit vastavalt nõuetele. Selleks oli loodud liides *ConnectionData* kolme meetodiga:

- *getJdbcUrl()* – peab tagastama andmebaasi URL-i;
- *getUsername()* – peab tagastama andmebaasi kasutajanime;
- *getPassword()* – peab tagastama andmebaasi parooli.

Seda liidest implementeerib kaks klassi, mis on ülalmainitud andmete allikad: *DatabaseContainer* – konteinerit esitav klass, kus olid need meetodid juba implementeeritud, ja *SqlServerProperties* – klass, mis hoidis konfiguratsioonifailis „*application-local.properties*“ määratud väärtusi olemasoleva andmebaasi külge ühendamiseks. Mõlemad klassid olid registreeritud *bean*'idena. *DatabaseContainer* objekti tagastav meetod lisaks vajab sisendiks klassi *ContainerSqlServerProperties*, kus on vajalikud muutujad Testcontainers teegi jaoks, ning paneb lisaks „*.start()*“ meetodiga konteineri käima. Vaikimisi on *bean*'id Spring raamistikus *singleton* skoobiga, seega rakenduse jooksutamise ajaks luuakse alati vaid üks andmebaasikonteinerile vastav objekt ning konteinerite arvu käsitsi kontrollida ei ole vajadust.

Et rakendus saaks õiget *bean*'i vastavalt vajadusele valida, autor defineeris konfiguratsioonifailis muutuja „*sqlserver.container*“ ning kasutas mõlema *bean*'i peal annotatsiooni *@ConditionalOnProperty* – kui muutuja väärtus on „true“ või puudub, registreeritakse *bean*'ina vaid *DatabaseContainer*, kui on „false“ – *SqlServerProperties* (vt Lisa 2 – *DatabaseConfig.java* klassi sisu).

4.3.2 Automaattestimise keskkonna konfigureerimine

Automaattestimise keskkonna puhul on oluline eelkõige teha valik, kuidas hakatakse testimise ajal andmebaasi taaslooma. Testcontainers teek on tihedalt integreeritud testimise teegiga JUnit 5, mis on kasutusel ka antud projektis, ning Testcontainers JUnit moodul toetab kahte tüüpi taasloomist:

- uus konteiner iga test-meetodi jaoks;
- üks konteiner on jagatud samas test-klassis olevate test-meetodite vahel [31].

Kuna automaattestid kasutavad andmebaasi lähteandmete leidmiseks ning funktsionaalsust otseselt ei kontrolli, pole otseselt vajadust mitu korda andmebaasi taasluua, vaid piisab ühest andmebaasikonteinerist kogu testimisprotsessi jooksul. Samuti võtaks baasi mitmekordne taasloomine rohkem aega ja ressursse, mis võib CI-keskkonnas probleemiks olla. Seega on autoril mõistlik taaskasutada „*mod-ephemeral-db*“ moodulis olevat implementatsiooni.

Automaattestimises klassi *DatabaseContainer* kasutamiseks on vaja teha ka mõned täiendused. Testcontainers kasutab konteinerite haldamiseks konteinerit Ryuk, millel peab olema piisavalt privileege konteinerite kustutamiseks, kuid selliste konteinerite käivitamine ei ole lubatud kasutatavas CI-keskkonnas Bamboo. Selleks, et konteinerid ei jääks tööle peale automaattestimise protsessi lõppemist, lisati kood konteineri peatumiseks JVM välja lülitamisel (vt Lisa 3 – *DatabaseContainer.java* klassi sisu).

4.4 Andmebaasi täitmine lähteandmetega

Peatükis 2.3 oli mainitud, et katsetuskeskkonna puhul on arendajate jaoks äärmiselt tähtis test-andmete olemasolu. Automaattestide jaoks eelistatakse andmebaasi vaid vajalike andmetega, nt parameetrid ja klassifikaatorid.

Pidades nõu teiste arendajatega meeskonnast, autor otsustas, et kõige optimaalsem variant on võtta kasutusele kaks Dockeri tõmmist. Üks tõmmis on n-ö „puhas“, tühja andmebaasi koopia ning sisaldab vaid neid andmeid, mida on vaja rakenduse korrektseks töötamiseks. See tõmmis oli lisatud LHV sisemisse repositooriumisse, kuhu on koondatud kõik teised tõmmised, ning tänu sellele saab seda tõmmist hiljem kasutada ka CI-keskkonnas andmebaasi loomiseks.

Teine Dockeri tõmmis oli jäetud koos peatükis 4.1 koostatud shell-skriptiga projekti Git-repositooriumisse. Sealt saab iga arendaja selle enda masinas käivitada, lisades andmebaasi loomise skriptide kataloogi vajalikus koguses muid SQL-skriptifaile. Vaikimisi oli autori poolt kataloogi lisatud teatud andmete hulka lisav skriptfail.

Siiski vajavad automaattestid hulka andmeid, millega võib alati arvestada, ent uute testide loomise või vanade testide muudatustega võib kaasneda vajadus selle hulga muutmiseks. Samuti tuleb veenduda, et andmete lisamine hakkab toimuma vaid ühekordselt. Kuna andmebaasi tõmmise uuendamine repositooriumis võib nõuda parasjagu aega ja spetsiifilisi teadmisi, otsustas autor hakata andmeid lisama Liquibase abil, kasutades kontekstide mehhanismi.

Liquibase kontekst (ingl *context*) on *changeset*'ile lisatav parameeter, mille abil saab kontrollida, milliseid skripte teatud jooksutamisel Liquibase poolt käima pannakse [32]. Selleks tuleb lisada igale *changeset*'ile väljendi (ingl *expression*) kujul konteksti parameeter, nt „*!autotest*", „*master or local*" jms, ning seejärel määrata rakenduse käivitamisel kontekst – Spring Boot võimaldab seda teha ka konfiguratsioonifailis.

Lahendus oli implementeeritud nii, et kõikide *changeset*'ide konteksti parameetrina oli lisatud „*master*"-kontekst ja peamisse konfiguratsioonifaili „*application.properties*" oli lisatud võti-väärtus paar „*spring.liquibase.contexts=master*". Taasloodava andmebaasi moodulis olevatesse „*local*" ja „*autotest*" profiili konfiguratsioonifailidesse oli määratud kontekstiks vastavalt „*master,local*" ja „*master,autotest*", mis nüüdsest võimaldab luua migratsioone, mida hakatakse jooksutama vaid teatud juhtudel.

Automaattestide jaoks oli lisatud *changeset* nimega *test_data.sql* konteksti parameetriga „*master and (autotest or local)*", mida hakatakse jooksutama nii lokaalse kui automaattestide andmebaasi puhul. Migratsioonifail koosnes esialgu mõnest ettevõtte-, kaupmehe- või kasutajaobjekti lisavast SQL-lausest, kuid hiljem sai oluliselt täiendatud ka teiste test-andmetega. Samuti jättis autor välja „*autotest*" kontekstist trigerite jooksutamise, kuna andmevarundust hetkel automaattestid ei kontrolli – seetõttu muutus automaattestide käivitamine kiiremaks.

4.5 Rakenduse testitavuse tagamine

Kui konfiguratsioon sai valmis taasloodava andmebaasi kasutamiseks automaattestimises, tuli autoril tagada rakenduse korrektne testitavus ning see osutus suureks järeltööks. Töö teostamise hetkel oli projektis 333 integratsioonitesti ja 10 UI-testi; eelnevalt oli mainitud, et automaattestid kasutavad andmebaasi lähteandmete leidmiseks, ning taasloodavas andmebaasis sobivate andmete puuduse tõttu kukkus läbi ligi 100 integratsioonitesti ning kõik UI-testid.

Autor alustas testide parandamise tööd UI-testidest. Mõned seda tüüpi automaattestid on endale mõningaid andmeid tekitanud *@Before*-annotatsiooniga märgitud meetodis andmebaasiga päringute kaudu suhtlevate abimeetoditega, kuid enamus teste teatud määral küsib sobivaid andmeid baasist. Kuna lisatavate andmete hulk ühe UI-testi jaoks pole meeletult suur, autor otsustas andmeid lisada Java koodi ja abimeetodite kaudu.

Paljude integratsioonitestide puhul oli probleemiks see, et ajaloolistel põhjustel vajasid testid jooksutamiseks käivat rakendust – funktsionaalsust kontrolliti test-keskkonnas jooksva rakenduse peal. Sellisel juhul ei ole võimalust kasutada taasloodavat andmebaasi, seetõttu pidi autor tegelema integratsioonitestide migreerimisega *MockMvc* peale, mille abil on võimalik simuleerida rakenduse poole päringute tegemist nii, nagu see toimuks rakendust kasutades brauseri kaudu [33].

Kuna projektis on integratsioonitestide arv suhteliselt suur, oli autori ülesandeks määrata kõige tõhusam ja kompaktsem, kuid ka piisavalt mugav viis lähteandmete lisamiseks. Suurtel pärandrakendustel on omapärane keeruline, paljude sõltuvustega andmestruktuur ning vajalike andmete lisamine nõuab reeglina palju eeltööd, seetõttu otsustas autor kasutada integratsioonitestides lisamiseks SQL-formaadis skripte. Kõikidel antud projektiga töötavatel meeskonnaliikmetel on vajalikud oskused ja teadmised SQL-skriptide koostamiseks ning üleüldse nõuab arendamine ja automaattestide koostamine tänapäeval häid teadmisi andmebaaside valdkonnas.

Spring Boot raamistikus saab SQL-skriptide jooksutamiseks enne test-meetodit kasutada *@Sql*-annotatsiooni, kusjuures ühe annotatsiooni sees saab määrata ka mitu skripti, mis võimaldab skriptide paremat struktureerimist ning suuremat taaskasutatavust [34]. Seejärel saab testi alguses abimeetodiga küsida informatsiooni viimasena lisatud teatud tüüpi objekti kohta. *@Sql*-annotatsiooni kasutavaid teste märkis autor lisaks

annotatsiooniga *@Transactional*, mis tähendab, et kõik selle testi raames teostatud päringud andmebaasi poole pannakse transaktsiooni sisse ja testi lõppemisel vaikimisi teostatakse tagasivõtmine (ingl *rollback*) – niiviisi ei jää lisatud andmed baasi, mis väldib võimalikke konflikte (vt Joonis 8) [35]. Kokku oli koostatud üle 20 SQL-skripti, mis lisavad test-andmeid ning mida saab mugavalt kombineerida omavahel; kaks autori poolt koostatud ning omavahel kombineeritavat SQL-skripti on lisatud näitena (vt Lisa 4 – Näited lähteandmeid lisavatest SQL-skriptidest).

Kõiki teste ei olnud siiski võimalik märkida annotatsiooniga *@Transactional*, kuna rakenduse koodis on ka annotatsiooniga *@Transactional(propagation = NEVER)* märgitud meetodeid ja klasse, mis tähendab, et jooksvat transaktsiooni ei tohi olla. Selliste meetodite jaoks lisas autor andmeid ülalmainitud faili *test_data.sql*, mis lisab andmeid alguses, enne testide käivitamist.

```
@Test
@Sql({"/sql/cof/application_cof_approved.sql",
"/sql/cof/contract_cof_signed.sql"})
@Transactional
public void testContractData() throws Exception {
    // given
    BigInteger contId = testCofApplicationDao.getLastCofContractId();
    request.setContId(contId);
    var contractModel = cofContractService.getContractDetails(contId);

    // when
    var response = testWsRequest(REQUEST_URL, request,
                                 CofContractDataResponse.class);

    // then
    assertEquals(
            contractModel.getCofApplicationModel().getApplId().longValue(),
            response.getCofApplication().getApplicationId().longValue());
    assertEquals(
            contractModel.getContract().getContId().longValue(),
            response.getContract().getContId().longValue());
    assertEquals(
            contractModel.getContract().getStatusCode().toString(),
            response.getContract().getStatusCode());
}
```

Joonis 8. SQL-skripte andmete lisamiseks kasutav integratsioonitest.

Töö viimane etapp peaks olema lahenduse töölepanek ning katsetamine pideva integratsiooni keskkonnas, kuid sellega seoses tuli välja üks takistus, millega ei olnud autor võimeline midagi tegema. Eelnevalt oli mainitud, et hetkel on CI-keskkonnana kasutusel lahendus Bamboo, kuid lähimal ajal on plaanitud täielik üleminek CI/CD-keskkonnale GitLab. Praeguses Bamboo keskkonnas ei ole korrektselt seadistatud Docker, mis on implementeeritud lahenduse jaoks vajalik, ent Bamboo haldamine ei ole hetkeseisuga enam prioriteetne. Seega lahendust on võimalik katsetada CI-keskkonnas vaid siis, kui käesolev projekt migreeritakse uuele keskkonnale.

5 Tulemused

Käesoleva bakalaureusetöö peamise tulemusena said peatükkides 2.3 ja 2.4 kirjeldatud probleemid lahendatud; peamine töö eesmärk sai täidetud – nüüd on rakenduse andmebaasi võimalik taasluua uues keskkonnas. Töö raames saadud lahendust ning tehtud järeldusi saab jagada põhiliselt kolmeks kategooriaks:

- muudatused töös andmebaasiga ja selle versioneerimisega;
- arendusprotsessi mugavdamine ning parendamine;
- automaattestimise protsessi muutmine stabiilsemaks ja autonoomsemaks.

Muudatused töös andmebaasiga seisnevad selles, et praegusest peab Liquibase migratsioonide failinimes ette lisama kuupäeva ning aeg-ajalt (autori poolt pakutud, iga poole aasta tagant) tuleb kõik vanad migratsioonid arhiveerida ja taasloodava andmebaasi loomise skriptid vastavalt sellele uuendada. Hetkeseisuga on uus kokkulepe jõus üle kahe kuu ning siiamaani ei ole Liquibase *changeset*'ide ja andmebaasi taasloomisega konflikte ega probleeme esile tulnud.

Arenduses on nüüdsest võimalik kasutada taasloodavat andmebaasi, lihtsalt lisades rakenduse käivitamisel Spring Boot aktiivsete profiilide hulka profiil „*local*". Arendajal on võimalik mugavalt valida, kas ühendada rakendus ajutise baasiga, mida luuakse automaatselt, või olemasoleva lokaalse andmebaasiga, mis töötab nt Docker keskkonnas. Samuti saab arendaja valida, kas kasutada rakendust test-andmetega andmebaasi või tühja andmebaasi peal. Kõike seda saab mugavalt määrata Spring Boot konfiguratsioonifailis.

Automaattestimise protsess muutus palju stabiilsemaks, kuna nüüdsest ei võta testid andmebaasist juhuslikke andmeid, vaid igale testile on garanteeritud lähteandmete olemasolu. Kõik automaattestid on nüüd paremini isoleeritud ning enam ei tekita test-keskkonna andmebaasi igapäevaselt suurt hulka uusi andmeid. Siiski autoril hetkel puudub antud projektiga võimalus lahenduse katsetamiseks CI-keskkonnas.

Enamus arendajaid autori meeskonnast on uut funktsionaalsust proovinud ning jäid implementatsiooniga rahule, seega saab väita, et kasutajatesti tulemus oli positiivne.

Tõmmise loomisega probleeme ei tekkinud, taasloodava andmebaasi kasutamine arenduses oli sujuv ja mugav, seda nii ajutise kui lokaalse baasi puhul. Teised meeskonnaliikmed kiitsid hästi koostatud kasutusjuhendit ja dokumentatsiooni, õigesti valitud tehnoloogiat, mille tõttu saab kasutada sama andmebaasisüsteemi, mis on kasutusel ka päris keskkonnas, ning pidasid õigeks otsust hoida taasloodava andmebaasi konfiguratsiooni eraldi moodulis – niiviisi on garanteeritud, et päris keskkonnas ei kaasne sellega seoses mingeid probleeme. Autor isiklikult on mitu korda ajutist andmebaasist kasu tundnud, kui tegeles andmebaasiga seotud funktsionaalsuse arendamisega.

5.1 Lahenduse universaalsuse valideerimine

Selleks, et valideerida teostatud lahenduse universaalsust, otsustas autor katsetada seda ka teise tema pädevuses oleva rakenduse peal – diilerportaal. Diilerportaal (edaspidi DPORT) on CDS süsteemiga võrreldes oluliselt väiksem rakendus, mida kasutavad LHV äripartnerid, pangaga seotud teenuste osutamiseks (järelmaksu lepingu sõlmimine, pangakaartide tellimine jt). Väikese suuruse tõttu on DPORT rakendus, kus teostatakse värskendused ning uuendused alati esimesena ja seetõttu on kasutatavad tehnoloogiad selles rakenduses alati kõige aktuaalsemad.

Eelkõige jooksutas autor enda veidi kohandatud peatükis 4.2.2 mainitud Python skripti Liquibase migratsioonifailide õigesse järjekorda panemiseks. Seejärel kohandas autor peatükis 4.1 näidatud Dockerfile-skripti, määrates sobivad muutujate väärtused, ning lisas tõmmise LHV sisemisse repositooriumisse. Viimase sammuna kopeeris autor taasloodava andmebaasi mooduli „*mod-ephemeral-db*“ uude rakendusse ning tegi mõned spetsiifilised muudatused konfiguratsioonis: nimetas ümber *DataSource bean*’i „*dportDataSource*“-ks ja viis konfiguratsioonifailide muutujad projektiga vastavusse.

Valideerimise ajal suuri probleeme ei tekkinud: kogu funktsionaalsus (ühendus lokaalse andmebaasiga, ajutise andmebaasi käivitamine jm) toimis ka uues rakenduses ilma vigadeta. Hetkeseisuga kasutavad kõik DPORT-i integratsioonitestid *MockMvc* lahendust, seega autor ei tegelenud testide muutmisega ning automaatteste sai kohe jooksutada taasloodava andmebaasi peal. 241st automaattestist läks valdav enamus läbi taasloodava andmebaasiga, ligi 10 integratsioonitesti kukkus läbi lähteandmete puuduse tõttu – nendele lisas autor *@Sql*-annotatsiooni kaudu uued lähteandmed.

```
...
.docker:
  services:
    - *sisemine süsteem peidetud*/docker:19.03.8-2-dind
  variables:
    DOCKER_HOST: "tcp://localhost:2375"
    DOCKER_TLS_CERTDIR: ""
    DOCKER_DRIVER: overlay2

.build:
  extends: .docker
...
```

Joonis 9. Vajalik konfiguratsioon Docker'i kasutamiseks Gitlab CI/CD keskkonnas.

Kuna DPORT'i rakendus on juba migreeritud Gitlab'i peale, sai autor kontrollida lahenduse korrektsust ka CI-keskkonnas. Selleks tegi ta Gitlab'i konfiguratsioonifailis „*.gitlab-ci.yml*" vajalikud muudatused, et Docker tehnoloogia oleks automaattestide jooksutamise ajal kättesaadaval (vt Joonis 9), ning koostas lihtsa integratsioonitesti andmebaasiga ühenduse kontrollimaks. Jooksutamisel läks andmebaasikonteiner tööle ning tulemusena läksid kõik integratsioonitestid läbi. Seega võib väita, et lahendus töötab korrektselt CI-keskkonnas ning et CDS-i projektiga ei tohiks seoses selle implemantatsiooniga keerulisi või lahendamatuid probleeme tekkida.

5.2 Edasised tegevused

Ehkki automaattestimises on realiseeritud lahendus autori arvates kõige optimaalsem oma päris keskkonnaga sarnasuse tõttu, arenduses kasutusele võetud lahendus ei ole siiski kõige kiirem ja mugavam. Autor on veendunud, et manustatud mälupõhiste andmebaaside kasutamine arendusprotsessis on parem kui andmebaasi jooksutamine konteineris, ent rakenduse omapära ja eriomaduste tõttu pidi autor sellest plaanist loobuma.

Autor leiab, et selleks, et mälupõhised andmebaasid kasutusele võtta, tuleb eelkõige võtta kasutusele mõned kaasaegsed lahendused andmekihi automatiseerimiseks. Java rakenduse puhul on heaks näiteks peatükis 3.1 välja toodud teegid Spring Data JPA ja Hibernate – tänu andmebaasiga suhtlemisprotsessi automatiseeritamisele väheneb vajadus andmebaasisüsteemi spetsiifiliste lahenduste kasutamine (nt SQL-päringute väljakirjutamine või SQL Server dialektis päringute koostamine) ning rakenduse

veakindlus muude andmebaasisüsteemidega suhtlemise juhul suureneb. Samuti on autor veendunud, et Liquibase muutumiskomplektide koostamine SQL-formaadis ei ole kõige optimaalsem variant – nende asemel võinuks kasutada mõnda teist Liquibase poolt pakutavat formaati *changeset*'ide koostamiseks (nt XML, JSON või YAML), mis oleks andmebaasisüsteemist ja tulenevatest eripäradest võimalikult sõltumatu.

Lõputöö raames kasutusele võetud teek Testcontainers teeb andmekihti valideeritavate integratsioonitestide koostamise ja jooksutamise lihtsamaks [24]. Hetkeseisuga on projektis CDS vaid 9 andmekihi funktsionaalsust kontrollivat integratsioonitesti, kuid isoleeritud ja alati kindlas olekus andmebaasi kasutamine automaattestimises võimaldab palju mugavamalt andmekihi funktsionaalsust valideerida ning seetõttu kasvab ka koodi kaetus testidega.

6 Kokkuvõte

Käesoleva töö raames oli põhjalikult uuritud taasloodava andmebaasi kasutuselevõttu, mis on arendus- ja testimisprotsessi parendamise oluline aspekt ning üks võimalik viis andmebaasiga töö tõhustamiseks. Selle töö eesmärk oli võtta kasutusele taasloodav andmebaas Java pärandrakenduse arenduses ja automaattestimises, lahendades kõik probleemid ja takistused.

Töö analüüsimise osas uuriti, millised võimalused ja tehnoloogiad on kiiruse, turvalisuse ja mugavuse poolest kõige sobilikumad taasloodava andmebaasi loomiseks; selgitati välja nende eripärad, kaasnevad potentsiaalsed takistused ja probleemid lähtudes ettevõtte vajadustest, olemasolevatest ökosüsteemidest, turvaaspektidest jm sätestatud piirangutest. Selle osa tulemusena kõige sobilikumaks lähenemiseks antud projekti jaoks osutus Docker tehnoloogial põhinev lahendus.

Lõputöö praktilises osas võeti taasloodav andmebaas analüütilise osa käigus valitud tehnoloogia abil nii arenduses kui ka automaattestimises ning lahendati ära probleemid, mis on varasemalt takistanud andmebaasi taasloomist. Tulemusena olid lõputöö eesmärgid täies ulatuses saavutatud: taasloodav andmebaas on muutunud antud pärandrakenduse arendusprotsessi mugavamaks ja efektiivsemaks, automaattestimine on tänu uuele lähenemisele palju veakindlam, stabiilsem ning autonoomsem.

Bakalaureusetöö raames langetatud otsused ja järeldused olid tehtud eelkõige lähtudes ettevõtte ja rakenduse spetsiifikast ning sellest tulenevatest nõuetest ja piirangutest. Ehkki iga projekt on arhitektuuri, struktuuri ning kasutatavate tehnoloogiate poolest unikaalne, on autor veendunud, et töös kirjeldatud lähenemine, probleemid ja nende lahendused on igati kasulikud ning taaskasutatavad ka teistes pärandrakendustes.

Kasutatud kirjandus

[1] GeeksForGeeks, „SQL | DDL, DQL, DML, DCL and TCL Commands,“ 7. aprill 2021. [Võrgumaterjal]. Loetud aadressil: https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/. [Kasutatud 28. aprill 2021].

[2] AS LHV Group, „Ettevõttest,“ [Võrgumaterjal]. Loetud aadressil: https://www.lhv.ee/et/ettevottest. [Kasutatud 28. aprill 2021].

[3] Microsoft Corporation, „SQL Server 2017 on Windows and Linux,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://www.microsoft.com/en-us/sql-server/sql-server-2017. [Kasutatud 7. mai 2021].

[4] Microsoft Corporation, „Transact-SQL Reference (Database Engine),“ 29. aprill 2020. [Võrgumaterjal]. Loetud aadressil: https://docs.microsoft.com/en-us/sql/t-sql/language-reference. [Kasutatud 4. mai 2021].

[5] Liquibase, „changelogs in SQL Format,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://docs.liquibase.com/concepts/basic/sql-format.html. [Kasutatud 14. aprill 2021].

[6] IBM Cloud Education, „Containerization,“ IBM, 15. mai 2019. [Võrgumaterjal]. Loetud aadressil: https://www.ibm.com/cloud/learn/containerization. [Kasutatud 17. aprill 2021].

[7] Docker Inc., „Docker overview,“ [Võrgumaterjal]. Loetud aadressil: https://docs.docker.com/get-started/overview/. [Kasutatud 18. aprill 2021].

[8] Docker Inc., „Docker Hub,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://hub.docker.com/. [Kasutatud 1. mai 2021].

[9] Docker Inc., „Microsoft SQL Server,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://hub.docker.com/_/microsoft-mssql-server. [Kasutatud 1. mai 2021].

[10] Docker Inc., „Docker Desktop for Mac and Windows,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://www.docker.com/products/docker-desktop. [Kasutatud 30. aprill 2021].

[11] C. Tozzi, „Docker Downsides: Container Cons to Consider before Adopting Docker,“ 26. mai 2017. [Võrgumaterjal]. Loetud aadressil: https://www.channelfutures.com/open-source/docker-downsides-container-cons-to-consider-before-adopting-docker. [Kasutatud 18. aprill 2021].

[12] Iron.io Blog, „Docker Containers: The Pros and Cons of Docker,“ 28. august 2020. [Võrgumaterjal]. Loetud aadressil: https://blog.iron.io/docker-containers-the-pros-and-cons-of-docker/. [Kasutatud 18. aprill 2021].

[13] OmniSci, Inc., „In-Memory Database,“ [Võrgumaterjal]. Loetud aadressil: https://www.omnisci.com/technical-glossary/in-memory-database. [Kasutatud 18. aprill 2021].

[14] Baeldung, „List of In-Memory Databases,“ 20. mai 2020. [Võrgumaterjal]. Loetud aadressil: https://www.baeldung.com/java-in-memory-databases. [Kasutatud 18. aprill 2021].

[15] H2, „H2 Database Engine,“ [Võrgumaterjal]. Loetud aadressil: https://www.h2database.com/. [Kasutatud 29. aprill 2021].

[16] The HSQL Development Group, „HSQLDB - 100% Java Database,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://hsqldb.org. [Kasutatud 29. aprill 2021].

[17] The Apache Software Foundation, „Apache Derby,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://db.apache.org/derby/. [Kasutatud 29. aprill 2021].

[18] H2, „Features,“ 2021. [Võrgumaterjal]. Loetud aadressil: http://www.h2database.com/html/features.html. [Kasutatud 29. aprill 2021].

[19] F. Toussi, „Chapter 13. Compatibility With Other DBMS,“ 21. märts 2021. [Võrgumaterjal]. Loetud aadressil: https://hsqldb.org/doc/2.0/guide/compatibility-chapt.html#coc_compatibility_mssql. [Kasutatud 29. aprill 2021].

[20] H2, „Performance,“ [Võrgumaterjal]. Loetud aadressil: http://www.h2database.com/html/performance.html. [Kasutatud 18. aprill 2021].

[21] Baeldung, „Spring Boot With H2 Database,“ 30. jaanuar 2021. [Võrgumaterjal]. Loetud aadressil: https://www.baeldung.com/spring-boot-h2-database. [Kasutatud 18. aprill 2021].

[22] Red Gate Software, Ltd., „Welcome to Spawn,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://www.spawn.cc/docs/getting-started. [Kasutatud 18. aprill 2021].

[23] Red Gate Software, Ltd., „Spawn: Summary,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://www.spawn.cc/docs/concepts-summary. [Kasutatud 18. aprill 2021].

[24] R. North, „Testcontainers,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://www.testcontainers.org/. [Kasutatud 27. aprill 2021].

[25] Docker, Inc., „Dockerfile reference,“ [Võrgumaterjal]. Loetud aadressil: https://docs.docker.com/engine/reference/builder/. [Kasutatud 29. märts 2021].

[26] Microsoft Corporation, „Quickstart: Run SQL Server container images with Docker,“ 7. september 2020. [Võrgumaterjal]. Loetud aadressil: https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker. [Kasutatud 31. märts 2020].

[27] Microsoft Corporation, „Configure and customize SQL Server Docker containers: Mount a host directory as data volume,“ 22. märts 2021. [Võrgumaterjal]. Loetud aadressil: https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-docker-container-configure?view=sql-server-ver15#mount-a-host-directory-as-data-volume. [Kasutatud 4. aprill 2021].

[28] Microsoft Corporation, „sqlcmd - Use with Scripting Variables,“ 9. august 2016. [Võrgumaterjal]. Loetud aadressil: https://docs.microsoft.com/en-us/sql/ssms/scripting/sqlcmd-use-with-scripting-variables. [Kasutatud 4. aprill 2021].

[29] Liquibase, „changeset - Available sub-tags,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://docs.liquibase.com/concepts/basic/changeset.html. [Kasutatud 14. aprill 2021].

[30] Spring.io, „Configuration Metadata,“ 15. aprill 2021. [Võrgumaterjal]. Loetud aadressil: https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-configuration-metadata.html. [Kasutatud 27. aprill 2021].

[31] Testcontainers, „Jupiter / JUnit 5,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://www.testcontainers.org/test_framework_integration/junit_5/. [Kasutatud 1. mai 2021].

[32] Liquibase, „Contexts,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://docs.liquibase.com/concepts/advanced/contexts.html. [Kasutatud 30. aprill 2021].

[33] VMware, Inc., „Testing the Web Layer,“ 2021. [Võrgumaterjal]. Loetud aadressil: https://spring.io/guides/gs/testing-web/. [Kasutatud 1. mai 2021].

[34] S. Brannen, „Annotation Type Sql,“ [Võrgumaterjal]. Loetud aadressil: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/jdbc/Sql.html. [Kasutatud 4. mai 2021].

[35] C. Sampaleanu, „Annotation Type Transactional,“ [Võrgumaterjal]. Loetud aadressil: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html. [Kasutatud 4. mai 2021].

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

Mina, Artjom Pahhomov

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Taasloodava andmebaasi kasutuselevõtt Java pärandrakenduse arenduses ja automaattestimises AS-i LHV Pank näitel", mille juhendaja on Ago Luberg:
   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.

17.05.2021

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

Lisa 2 – *DatabaseConfig.java* klassi sisu

```
import ee.lhv.cds.db.CdsDatabaseContainer;
import ee.lhv.cds.db.ConnectionData;
import ee.lhv.cds.properties.ContainerSqlServerProperties;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;
import javax.sql.DataSource;

@Configuration
@Profile({"autotest", "local"})
public class DatabaseConfig {
    @Value("${cds.base.datasource.driverClassName}")
    private String driverClassName;

    @Primary
    @Bean(name = "cdsDataSource")
    public DataSource getCdsDataSource(@Qualifier("connectionData")
ConnectionData connectionData) {
        final BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName(driverClassName);
        dataSource.setUsername(connectionData.getUsername());
        dataSource.setPassword(connectionData.getPassword());
        dataSource.setUrl(connectionData.getJdbcUrl());
        return dataSource;
    }

    @Bean(name = "connectionData")
    @SuppressWarnings("java:S1452")
    @ConditionalOnProperty(prefix = "sqlserver", name = "container",
havingValue = "true", matchIfMissing = true)
    public CdsDatabaseContainer<?>
sqlServerContainer(ContainerSqlServerProperties properties) {
        var container = new CdsDatabaseContainer<>(properties);
        container.start();
        return container;
    }
}
```

Lisa 3 – *DatabaseContainer.java* klassi sisu

```
import ee.lhv.cds.properties.ContainerSqlServerProperties;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.extern.log4j.Log4j2;
import org.testcontainers.containers.MSSQLServerContainer;
import org.testcontainers.utility.DockerImageName;
import static java.util.Objects.requireNonNull;

@Log4j2
@EqualsAndHashCode(callSuper = true)
public class DatabaseContainer<T extends MSSQLServerContainer<T>> extends
MSSQLServerContainer<T> implements ConnectionData {

    private static final Long MEMORY_LIMIT = 1024L * 1024L * 1024L * 2L;

    @Getter
    private String username;

    public DatabaseContainer(ContainerSqlServerProperties properties) {
        super(DockerImageName.parse(properties.getDockerImage())
                .asCompatibleSubstituteFor(MSSQLServerContainer.IMAGE)
                .withTag(properties.getDockerImageTag()));
        withUsername(properties.getUsername())
                .withPassword(properties.getPassword())
                .withDatabaseName(properties.getDatabaseName())
                .withReuse(true);
        setupMemoryLimit();
        addShutdownHook();
    }

    @Override
    public void start() {
        log.info("Container is starting...");
        super.start();
    }

    @Override
    public void stop() {
        log.info("Container is stopping...");
        super.stop();
    }
```

```
    protected void setupMemoryLimit() {
        this.withCreateContainerCmdModifier(cmd ->
requireNonNull(cmd.getHostConfig()).withMemory(MEMORY_LIMIT));
    }

    private void addShutdownHook() {
        Runtime.getRuntime().addShutdownHook(new Thread(this::stop));
    }

    @Override
    public T withUsername(String username) {
        this.username = username;
        return self();
    }

    @Override
    public T withDatabaseName(String dbName) {
        return withUrlParam("databaseName", dbName);
    }
}
```

Lisa 4 – Näited lähteandmeid lisavatest SQL-skriptidest

Fail *sql/cof/application_cof_approved.sql*:

```
DECLARE
    @APPL_ID NUMERIC,
    @RUSE_ID NUMERIC,
    @APDE_ID NUMERIC

BEGIN

INSERT INTO CDS.APPLICATION (PROD_ID, CATEGORY_CODE, CORP_PRIVATE,
STATUS_CODE, CHANNEL_CODE, DEOR_ID, DEBR_ID, DEUS_ID)
VALUES (CDS.fn_getProdId('CF', 'COF'), 'COF', 'P', 'DECIDED', 'W', 10000,
10100, 10200)

SET @APPL_ID = SCOPE_IDENTITY()

INSERT INTO CDS.APPLICATION_PERSON (APPL_ID, PERSON_ROLE, FIRST_NAME,
LAST_NAME, USER_ID, ID_CODE, ID_CODE_ISSUER, GENDER)
VALUES (@APPL_ID, 'P_APL', 'Alo', 'Seep', 42055, '36011114243', 'EE', 'M')

INSERT INTO CDS.APPLICATION_COF (APPL_ID, COF_TYPE_CODE,
CONSUMER_LOAN_ALLOWED, CONSUMER_FINANCE_ALLOWED)
VALUES (@APPL_ID, 'CF', 'Y', 'Y')

INSERT INTO CDS.RULE_SESSION (APPL_ID, IS_VALID)
VALUES (@APPL_ID, 1)

SET @RUSE_ID = SCOPE_IDENTITY()

INSERT INTO CDS.APPLICATION_DECISION (APPL_ID, RUSE_ID, DECISION_NUMBER,
DECISION_CODE)
VALUES (@APPL_ID, @RUSE_ID, @APPL_ID, 'APPROVED')

SET @APDE_ID = SCOPE_IDENTITY()

INSERT INTO CDS.APPLICATION_DECISION_COF (APDE_ID, CONSUMER_LOAN_ALLOWED,
CONSUMER_FINANCE_ALLOWED, INTEREST_BASE_PERCENT, INTEREST_MARGIN_PERCENT,
MAX_REPAYMENT_PERIOD, MIN_DOWN_PAYMENT_PERCENT)
VALUES (@APDE_ID, 'Y', 'Y', 19.90, 9.90, 72, 0.00)

END;
```

Fail *sql/cof/contract_cof_signed.sql*:

```
DECLARE
    @APPL_ID NUMERIC,
    @APDE_ID NUMERIC,
    @CONT_ID NUMERIC,
    @COPS_ID NUMERIC

BEGIN

SET @APPL_ID = IDENT_CURRENT('CDS.APPLICATION')
SET @APDE_ID = IDENT_CURRENT('CDS.APPLICATION_DECISION')

INSERT INTO CDS.CONTRACT (APPL_ID, APDE_ID, CONTRACT_NUMBER, STATUS_CODE,
DEOR_ID, DEBR_ID, DEUS_ID, INTEREST_BASE_PERCENT, INTEREST_MARGIN_PERCENT,
PROD_ID, PAYMENT_DAY, INVOICE_DELIVERY)
VALUES (@APPL_ID, @APDE_ID, 100000123, 'CONTRACT_SIGNED', 10000, 10100,
10200, 19.90, 9.90, CDS.fn_getProdId('CF', 'COF'), 10, 'EMAIL')

SET @CONT_ID = SCOPE_IDENTITY()

INSERT INTO CDS.CONTRACT_COF (CONT_ID, DOWN_PAYMENT_PERCENT,
FACTORING_AMOUNT, REPAYMENT_PERIOD, DOWN_PAYMENT_AMOUNT)
VALUES (@CONT_ID, 0.00, 601.00, 23, 0.00)

INSERT INTO CDS.CONTRACT_PAYMENT_SCHEDULE (CONT_ID)
VALUES (@CONT_ID)

SET @COPS_ID = SCOPE_IDENTITY()

INSERT INTO CDS.PAYMENT_SCHEDULE_ROW (COPS_ID, ENTRY_TYPE, PERIOD,
PAYMENT_DATE, RESIDUAL_AMOUNT)
VALUES (@COPS_ID, 'Downpayment', 0, '2021-04-13 11:11:11.000', 15111.00),
       (@COPS_ID, 'Installment', 1, '2021-06-10 11:11:11.000', 15111.00),
       (@COPS_ID, 'Installment', 2, '2021-07-10 11:11:11.000', 14901.12),
       (@COPS_ID, 'Installment', 3, '2021-08-10 11:11:11.000', 14691.24),
       (@COPS_ID, 'Installment', 4, '2021-09-10 11:11:11.000', 14481.36),
       (@COPS_ID, 'Installment', 5, '2021-10-10 11:11:11.000', 14271.48),
       (@COPS_ID, 'Installment', 6, '2021-11-10 11:11:11.000', 14061.60),
       (@COPS_ID, 'Installment', 7, '2021-12-10 11:11:11.000', 13851.72),
       (@COPS_ID, 'Installment', 8, '2022-01-10 11:11:11.000', 13641.84),
       (@COPS_ID, 'Installment', 9, '2022-02-10 11:11:11.000', 13431.96),
       (@COPS_ID, 'Installment', 10, '2022-03-10 11:11:11.000', 13222.08),
       (@COPS_ID, 'Installment', 11, '2022-04-10 11:11:11.000', 13012.20),
       (@COPS_ID, 'Installment', 12, '2022-05-10 11:11:11.000', 12802.32),
       (@COPS_ID, 'Installment', 13, '2022-06-10 11:11:11.000', 12592.44),
       (@COPS_ID, 'Installment', 14, '2022-07-10 11:11:11.000', 12382.56),
       (@COPS_ID, 'Installment', 15, '2022-08-10 11:11:11.000', 12172.68),
       (@COPS_ID, 'Installment', 16, '2022-09-10 11:11:11.000', 11962.80),
       (@COPS_ID, 'Installment', 17, '2022-10-10 11:11:11.000', 11752.92),
       (@COPS_ID, 'Installment', 18, '2022-11-10 11:11:11.000', 11543.04),
```

```
        (@COPS_ID, 'Installment', 19, '2022-12-10 11:11:11.000', 11333.16),
        (@COPS_ID, 'Installment', 20, '2023-01-10 11:11:11.000', 11123.28),
        (@COPS_ID, 'Installment', 21, '2023-02-10 11:11:11.000', 10913.40),
        (@COPS_ID, 'Installment', 22, '2023-03-10 11:11:11.000', 10703.52),
        (@COPS_ID, 'Installment', 23, '2023-04-10 11:11:11.000', 10493.64)

END;
```